THE UNIVERSITY OF TEXAS AT DALLAS

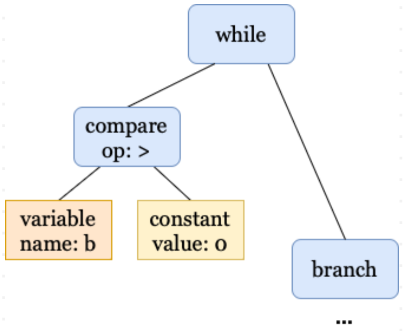
(Partial) Program Dependence Learning

Aashish Yadavally and Tien N. Nguyen
Computer Science Department
The University of Texas at Dallas

Wenbo Wang and Shaohua Wang
Department of Informatics
New Jersey Institute of Technology

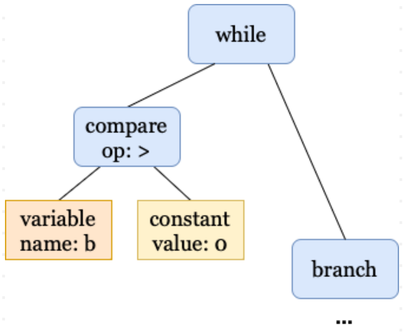
Background

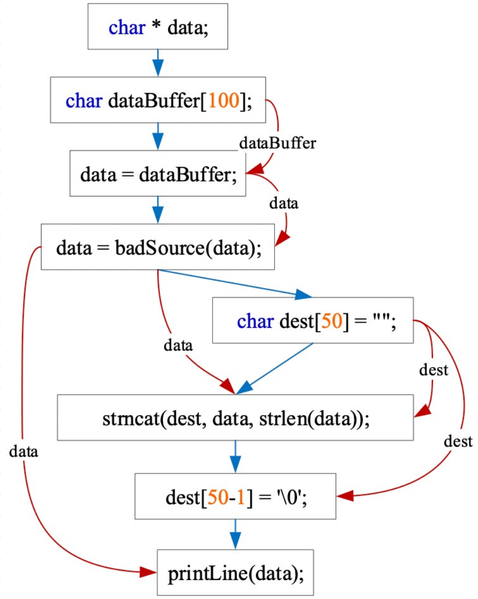
❖ Given complete program units, one can utilize different tools to help build program representations such as abstract syntax trees (ASTs)



Background

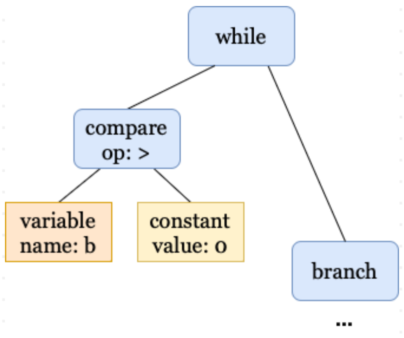
❖ Given complete program units, one can utilize different tools to help build program representations such as abstract syntax trees (ASTs), control-flow and program dependence graphs (CFGs & PDGs), etc.

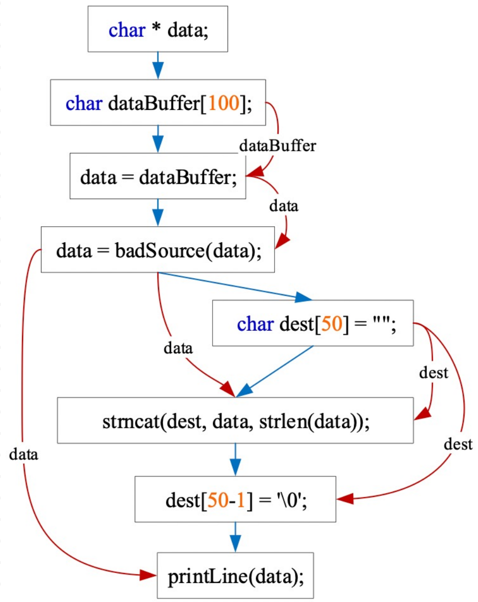




Background

❖ Given complete program units, one can utilize different tools to help build program representations such as abstract syntax trees (ASTs), control-flow and program dependence graphs (CFGs & PDGs), etc.





❖ In contrast, analyzing code fragments from online forums is difficult as they are often incomplete, unparseable, contain declaration/reference ambiguity, and are interspersed between user comments.

Let us call this ***partial program dependence analysis***.

Why is this useful?

Why is this useful?

An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples

Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh

Abstract—Software developers share programming solutions in Q&A sites like Stack Overflow, Stack Exchange, Android forum, and so on. The reuse of crowd-sourced code snippets can facilitate rapid prototyping. However, recent research shows that the shared code snippets may be of low quality and can even contain vulnerabilities. This paper aims to understand the nature and the prevalence of security vulnerabilities in crowd-sourced code examples. To achieve this goal, we investigate security vulnerabilities in the C++ code snippets shared on Stack Overflow over a period of 10 years. In collaborative sessions involving multiple human coders, we manually assessed each code snippet for security vulnerabilities following CWE (Common Weakness Enumeration) guidelines. From the 72,483 reviewed code snippets used in at least one project hosted on GitHub, we found a total of 99 vulnerable code snippets categorized into 31 types. Many of the investigated code snippets are still not corrected on Stack Overflow. The 99 vulnerable code snippets found in Stack Overflow were reused in a total of 2859 GitHub projects. To help improve the quality of code snippets shared on Stack Overflow, we developed a browser extension that allows Stack Overflow users to be notified for vulnerabilities in code snippets when they see them on the platform.

Index Terms—Stack Overflow, Software Security, C++, SOTorrent, Vulnerability Migration, GitHub, Vulnerability Evolution

Why is this useful?

- Some existing approaches build CFG/PDGs for partial code via ***manual intervention*** (e.g., by wrapping around method signature) in a best-effort manner, often at the cost of several misses.

Why is this useful?

- Some existing approaches build CFG/PDGs for partial code via ***manual intervention*** (e.g., by wrapping around method signature) in a best-effort manner, often at the cost of several misses.

```
1 string subTag(string s, string a, string b) {
2   std::string lower_s;
3   std::transform(s.begin(), s.end(), lower_s.begin(), ::
        tolower);
4   std::transform(a.begin(), a.end(), a.begin(), ::tolower);
5   auto position = lower_s.find(a);
6                              ...
```

In the above code listing, due to the unknown data type `string` on line 2 and API element `transform` on lines 3 and 4, traditional program analysis tools ignore all CFG/PDG edges to/from these statements. However, this listing is vulnerable on line 3, automated detection of which is not possible due to the missing edges.

Motivating Examples

Observation 1

Partial program dependence analysis is desirable for the software engineering tasks in which completely analyzable code is not available.

```cpp
std :: shared ptr<FILE> pipe(popen(cmd,"r"), pclose);
if (! pipe) return "ERROR";
char buffer[128];
std :: string result = "";
while (! feof (pipe.get() ) ) {
  if ( fgets (buffer , 128, pipe.get() ) != NULL)
    result += buffer;
}
```

Illustration 1: Incomplete S/O code snippet to execute a command within a C++ program, that is prone to OS Command Injection.

Motivating Examples

Observation 1

Partial program dependence analysis is desirable for the software engineering tasks in which completely analyzable code is not available.

```
std :: shared_ptr<FILE> pipe(popen(cmd,"r"), pclose);
if (! pipe) return "ERROR";
char buffer[128];
std :: string result = "";
while (! feof (pipe.get() ) ) {
  if ( fgets (buffer , 128, pipe.get() ) != NULL)
    result += buffer;
}
```

Illustration 1: Incomplete S/O code snippet to execute a command within a C++ program, that is prone to OS Command Injection.

```
#include <cstdio>
#include <iostream>
#include <memory>
#include <stdexcept>
#include <string>
#include <array>

std::string exec(const char* cmd) {
  std::array<char, 128> buffer;
  std::string result;
  std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(cmd,
                                                   "r"), pclose);
  if (!pipe) {
    throw std::runtime_error("popen() failed!");
  }
  while (fgets(buffer.data(), buffer.size(),
               pipe.get()) != nullptr) {
    result += buffer.data();
  }
  return result;
}
```

Illustration 2: Complete code example with the same POSIX elements as in Illustration 1. Here, similar statement blocks are highlighted with same color.

Observation 2

Inter-statement dependence analysis of partial code can be derived from the patterns learned from such analyses of entire programs in existing code corpora.

Key Ideas

- PDGs are ***repetitive***!

> Nguyen *et al.* [1] reported that among 17.5M PDGs with 1.6B PDG subgraphs, 14.3% of the PDGs have all of their subgraphs repeated across different projects. Furthermore, in 15.6% of the PDGs, at least 90% of their subgraphs are likely to have appeared before in other projects.

[1] Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2016. A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). Association for Computing Machinery, New York, NY, USA, 362–373.

Key Ideas

- PDGs are ***repetitive***!
- We can leverage a pattern learning-based approach to **partial program dependence analysis**.

[1] Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2016. A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). Association for Computing Machinery, New York, NY, USA, 362–373.

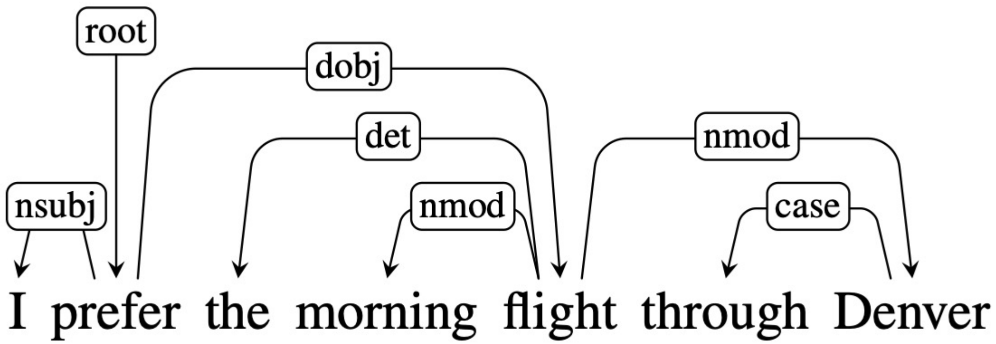
Key Ideas

- PDGs are ***repetitive***!
- We can leverage a pattern learning-based approach to **partial program dependence analysis**.
- Is this even possible?

[1] Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2016. A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). Association for Computing Machinery, New York, NY, USA, 362–373.

Key Ideas

- PDGs are ***repetitive***!
- We can leverage a pattern learning-based approach to **partial program dependence analysis**.
- Is this even possible?



[1] Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2016. A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). Association for Computing Machinery, New York, NY, USA, 362–373.

General Overview

```
private boolean isValidUntil ( Until annotation) {
    if (annotation != null ) {
        double annotationVersion = annotation.value();
        if (annotationVersion <= version) {
            return false ;
        }
    }
    return true ;
}
```

Code Snippet

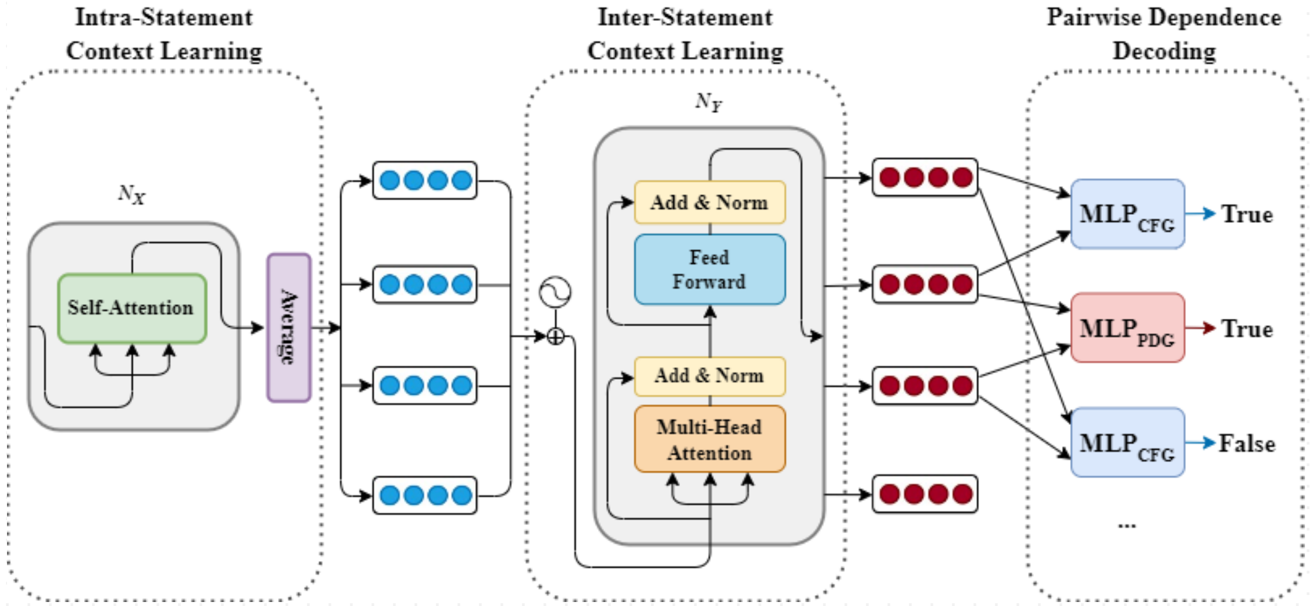
Training Process

- From ***complete*** code corpora, build CFG/PDGs to utilize as ground-truth.
- Train a model to predict the presence of a CFG/PDG edge between two statements in a specific code context.

Inference

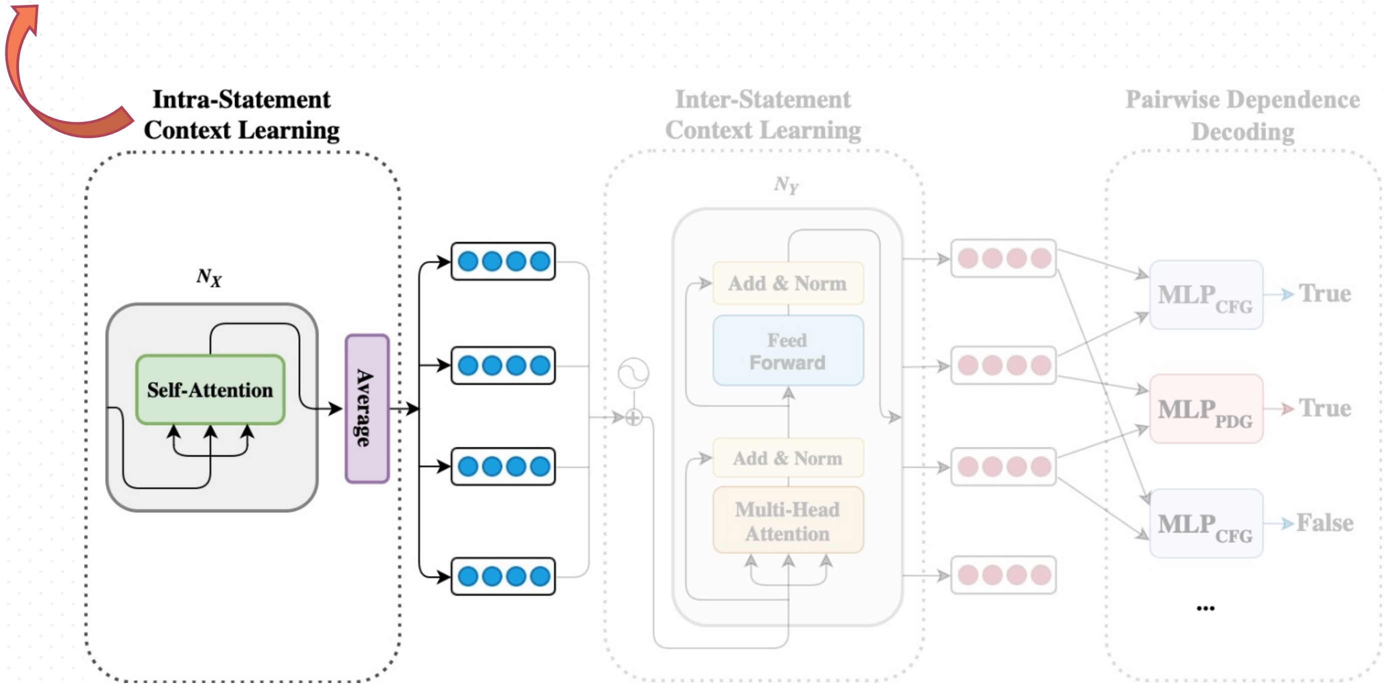
- Given ***partial/complete*** code snippet, predict the presence of a CFG/PDG edge between each statement pair.
- Combination of all such CFG/PDG edges can be realized as a CFG/PDG of the code snippet.

NeuralPDA: Neural Network-Based Program Dependence Analysis

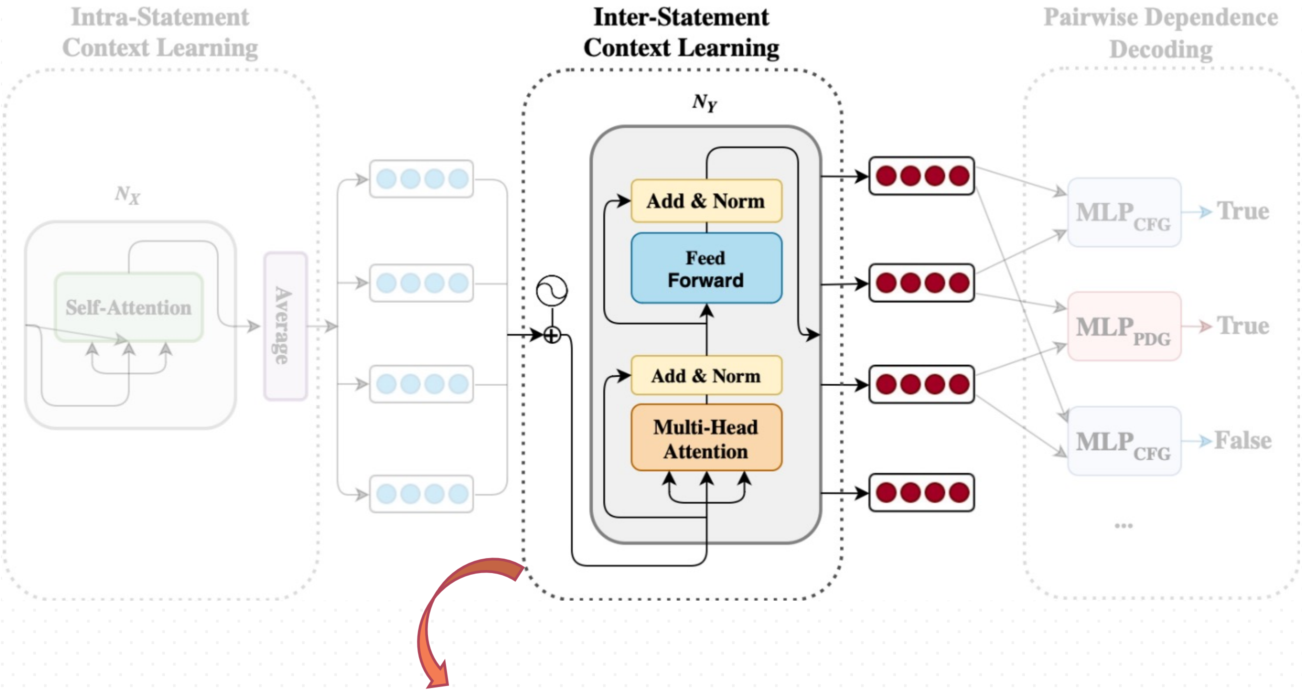


NeuralPDA: Neural Network-Based Program Dependence Analysis

Helps ***relay the syntactic and semantic relationships*** between the tokens in a statement to other statements in the code snippet.



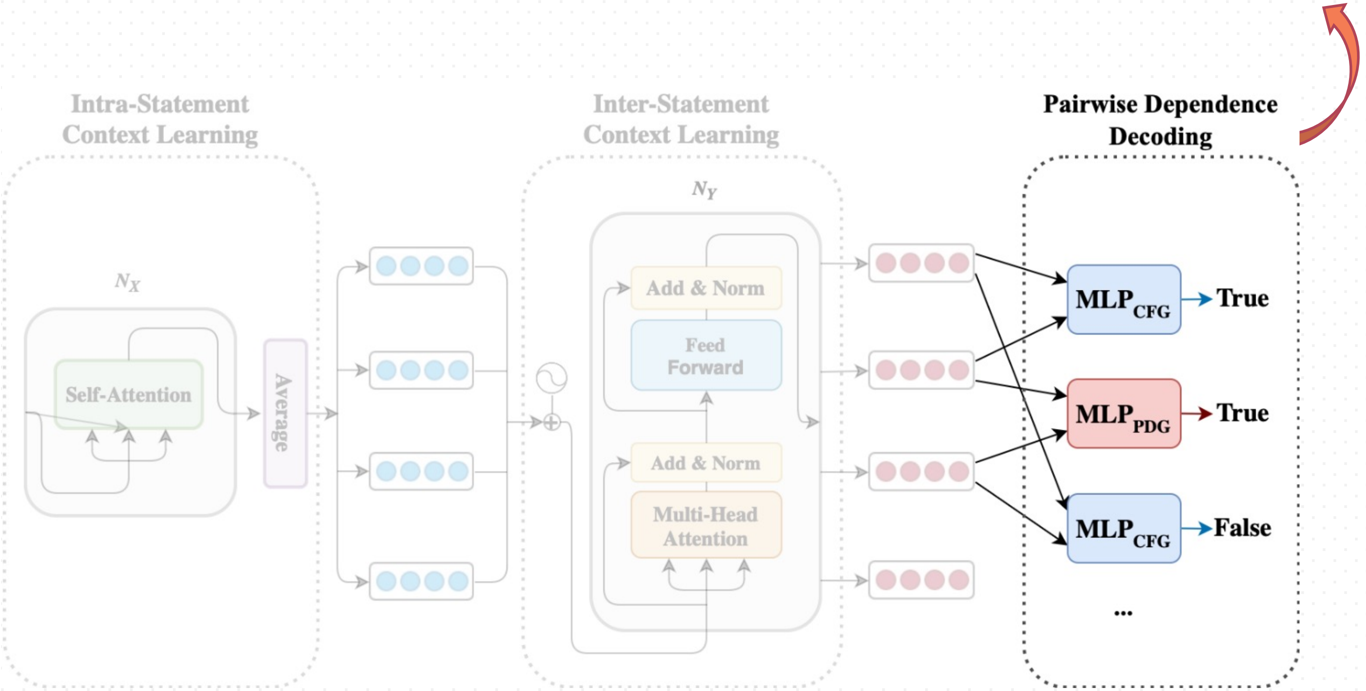
NEURALPDA: Neural Network-Based Program Dependence Analysis



The goal of this component is to ***learn latent representations*** for each statement that model the inter-statement dependencies.

NEURALPDA: Neural Network-Based Program Dependence Analysis

The combination of all the CFG/PDG edges extracted via such an arc-factored approach is ***realized as the CFG/PDG*** for the given program.



Empirical Evaluation (Intrinsic)

P/L	Graph	Accuracy	Precision	Recall	F1-Score
Java	*CFG*	99.79	98.31	98.58	98.44
	PDG	98.87	89.89	87.53	88.70
	Overall	99.33	94.75	93.83	**94.29**
C/C++	*CFG*	99.50	96.76	96.56	96.66
	PDG	98.55	83.55	90.01	86.66
	Overall	99.02	91.10	93.87	**92.46**

Table 1: Performance of NeuralPDA on **complete** code from Java and C/C++ ***(Intrinsic Evaluation)***

Empirical Evaluation (Intrinsic)

P/L	Graph	Accuracy	Precision	Recall	F1-Score
Java	*CFG*	99.79	98.31	98.58	98.44
	PDG	98.87	89.89	87.53	88.70
	Overall	99.33	94.75	93.83	**94.29**
C/C++	*CFG*	99.50	96.76	96.56	96.66
	PDG	98.55	83.55	90.01	86.66
	Overall	99.02	91.10	93.87	**92.46**

Table 1: Performance of NeuralPDA on complete code from Java and C/C++ ***(Intrinsic Evaluation)***

k	F1-Score (in %)					
	Java			*C/C++*		
	CFG	*PDG*	*Overall*	*CFG*	*PDG*	*Overall*
3	99.70	93.24	97.17	98.39	91.10	96.01
4	99.41	92.51	96.61	98.33	90.18	95.28
5	99.26	91.23	95.96	97.91	89.10	94.37
6	99.04	90.36	95.40	97.14	87.91	93.31
7	98.75	89.45	94.82	96.69	86.45	92.39
8	98.44	88.70	**94.29**	96.66	86.66	**92.46**

Table 2: Performance of NeuralPDA on partial code from Java and C/C++ ***(Intrinsic Evaluation)***

Empirical Evaluation (Extrinsic)

PDG from NEURALPDA

$$0 < VD\{PDG^{*}\} \leq VD\{PDG^{\#}\}$$

PDG from Program Analysis tool

Empirical Evaluation (Extrinsic)

PDG from NEURALPDA

$$0 < VD\{PDG^*\} \leq VD\{PDG^\#\}$$

PDG from Program Analysis tool

The PDGs predicted by NEURALPDA approximates the performance of those generated by program analysis tools for ***vulnerability detection on complete code*** *by* ***98.98%*** *.*

Empirical Evaluation (Extrinsic)

PDG from NeuralPDA

$$0 < VD\{PDG^{*}\} \leq VD\{PDG^{\#}\}$$

PDG from Program Analysis tool

*The PDGs predicted by NeuralPDA approximates the performance of those generated by program analysis tools for **vulnerability detection on complete code** by **98.98%** .*

*For the **vulnerability detection task on partial code**, the PDGs predicted by NeuralPDA helps an automated tool discover **14** real-world vulnerable code fragments.*

Empirical Evaluation (Qualitative)

Table 3: Performance of NeuralPDA on **different types of CFG/PDG edges** for Java (left) and C/C++ (right) code.

Graph	Edge Type	%C
CFG	*sequential*	99.54
	if-else	95.52
PDG	*data dependence*	82.78
	control dependence	96.33

Graph	Edge Type	%C
CFG	*sequential*	98.91
	if-else	**
PDG	*data dependence*	88.21
	control dependence	94.65

Building CFG/PDG with NeuralPDA: An Illustration

```java
private boolean isValidUntil ( Until annotation) {
    if (annotation != null ) {
        double annotationVersion = annotation.value();
        if (annotationVersion <= version) {
            return false ;
        }
    }
    return true ;
}
```

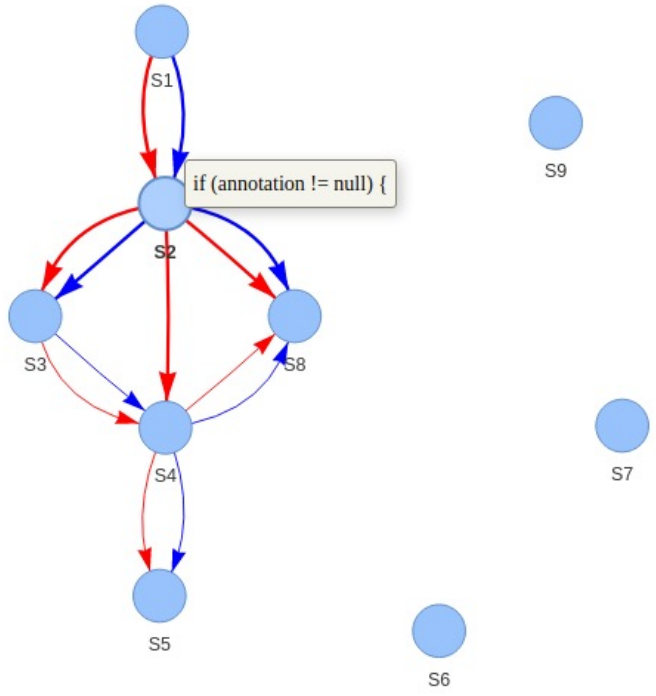


Figure 1: Java code listing (left) and its corresponding **CFG**/**PDG** (right) predicted by NeuralPDA

Key Takeaways

- NeuralPDA is the first neural network tool to predict program dependencies in complete as well as partial programs, which are accurate as well as **380×** faster to generate.
- This work leads to a direction for improving program analysis (PA) for partial programs by combining pattern learning-based approaches with top-down PA techniques.

Conclusion

Why is this useful?

- Some existing approaches build CFG/PDGs for partial code via ***manual intervention*** (e.g., by wrapping around method signature) in a best-effort manner, often at the cost of several misses.

```
1 string subTag(string s, string a, string b) {
2   std::string lower_s;
3   std::transform(s.begin(), s.end(), lower_s.begin(), ::
        tolower);
4   std::transform(a.begin(), a.end(), a.begin(), ::tolower);
5   auto position = lower_s.find(a);
6                          ...
```

In the above code listing, due to the unknown data type `string` on line 2 and API element `transform` on lines 3 and 4, traditional program analysis tools ignore all CFG/PDG edges to/from these statements. However, this listing is vulnerable on line 3, automated detection of which is not possible due to the missing edges.

lusion

Why is this useful?

❖ Some existing approaches build CFG/PDGs for partial code via ***manual intervention*** (e.g., by wrapping around method signature) in a best-effort manner, often at the cost of several misses.

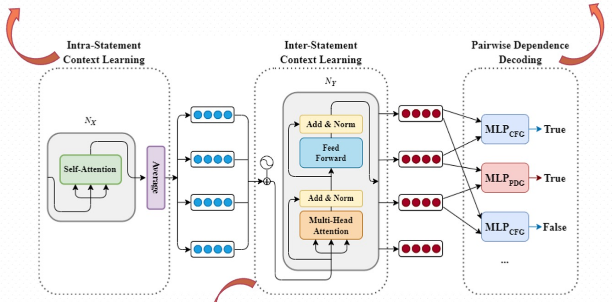
```
1 string subTag(string s, string a, string b) {
2   std::string lower_s;
3   std::transform(s.begin(), s.end(), lower_s.begin(), ::
        tolower);
4   std::transform(a.begin(), a.end(), a.begin(), ::tolower);
5   auto position = lower_s.find(a);
6                           ...
```

In the above code listing, due to the unknown data type `string` on line 2 and API element `transform` on lines 3 and 4, traditional program analysis tools ignore all CFG/PDG edges to/from these statements. However, this listing is vulnerable on line 3, automated detection of which is not possible due to the missing edges.

NeuralPDA: Neural Network-Based Program Dependence Analysis

Helps ***relay the syntactic and semantic relationships*** between the tokens in a statement to other statements in the code snippet.

The combination of all the CFG/PDG edges extracted via such an arc-factored approach is ***realized as the CFG/PDG*** for the given program.



The goal of this component is to ***learn latent representations*** for each statement that model the inter-statement dependencies.

Why is this useful?

- Some existing approaches build CFG/PDGs for partial code via ***manual intervention*** (e.g., by wrapping around method signature) in a best-effort manner, often at the cost of several misses.

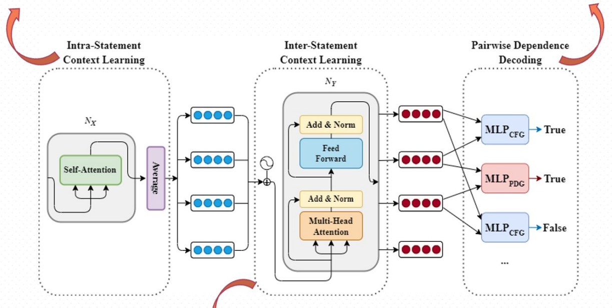
```
1 string subTag(string s, string a, string b) {
2   std::string lower_s;
3   std::transform(s.begin(), s.end(), lower_s.begin(), ::
        tolower);
4   std::transform(a.begin(), a.end(), a.begin(), ::tolower);
5   auto position = lower_s.find(a);
6                          ...
```

In the above code listing, due to the unknown data type `string` on line 2 and API element `transform` on lines 3 and 4, traditional program analysis tools ignore all CFG/PDG edges to/from these statements. However, this listing is vulnerable on line 3, automated detection of which is not possible due to the missing edges.

NeuralPDA: Neural Network-Based Program Dependence Analysis

Helps ***relay the syntactic and semantic relationships*** between the tokens in a statement to other statements in the code snippet.

The combination of all the CFG/PDG edges extracted via such an arc-factored approach is ***realized as the CFG/PDG*** for the given program.



The goal of this component is to ***learn latent representations*** for each statement that model the inter-statement dependencies.

Empirical Evaluation

P/L	Graph	Accuracy	Precision	Recall	F1-Score
Java	*CFG*	99.79	98.31	98.58	98.44
	PDG	98.87	89.89	87.53	88.70
	Overall	99.33	94.75	93.83	**94.29**
C/C++	*CFG*	99.50	96.76	96.56	96.66
	PDG	98.55	83.55	90.01	86.66
	Overall	99.02	91.10	93.87	**92.46**

Table 1: Performance of NeuralPDA on **complete** code from Java and C/C++ ***(Intrinsic Evaluation)***

k	F1-Score (in %)					
	Java			*C/C++*		
	CFG	*PDG*	*Overall*	*CFG*	*PDG*	*Overall*
3	99.70	93.24	97.17	98.39	91.10	96.01
4	99.41	92.51	96.61	98.33	90.18	95.28
5	99.26	91.23	95.96	97.91	89.10	94.37
6	99.04	90.36	95.40	97.14	87.91	93.31
7	98.75	89.45	94.82	96.69	86.45	92.39
8	98.44	88.70	**94.29**	96.66	86.66	**92.46**

Table 2: Performance of NeuralPDA on **partial** code from Java and C/C++ ***(Intrinsic Evaluation)***

*The PDGs predicted by NeuralPDA approximates the performance of those generated by program analysis tools for **vulnerability detection on complete code** by **98.98%** **(Extrinsic Evaluation)***

*For the **vulnerability detection task on partial code**, the PDGs predicted by NeuralPDA helps an automated tool discover **14** real-world vulnerable code fragments **(Extrinsic Evaluation)***

Scan QR code to access the replication package, data, and supplementary material.

Why is this useful?

- Some existing approaches build CFG/PDGs for partial code via ***manual intervention*** (e.g., by wrapping around method signature) in a best-effort manner, often at the cost of several misses.

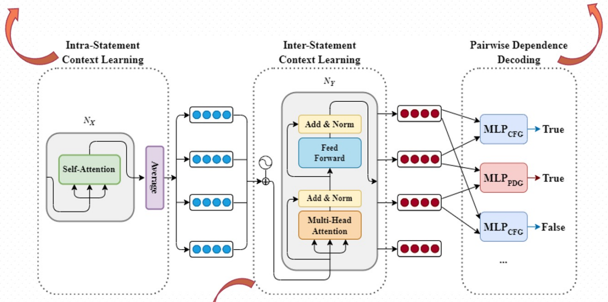
```
1 string subTag(string s, string a, string b) {
2   std::string lower_s;
3   std::transform(s.begin(), s.end(), lower_s.begin(), ::
        tolower);
4   std::transform(a.begin(), a.end(), ::tolower);
5   auto position = lower_s.find(a);
6                               ...
```

In the above code listing, due to the unknown data type `string` on line 2 and API element `transform` on lines 3 and 4, traditional program analysis tools ignore all CFG/PDG edges to/from these statements. However, this listing is vulnerable on line 3, automated detection of which is not possible due to the missing edges.

NeuralPDA: Neural Network-Based Program Dependence Analysis

Helps ***relay the syntactic and semantic relationships*** between the tokens in a statement to other statements in the code snippet.

The combination of all the CFG/PDG edges extracted via such an arc-factored approach is ***realized as the CFG/PDG*** for the given program.



The goal of this component is to ***learn latent representations*** for each statement that model the inter-statement dependencies.

Empirical Evaluation

P/L	Graph	Accuracy	Precision	Recall	F1-Score
Java	*CFG*	99.79	98.31	98.58	98.44
	PDG	98.87	89.89	87.53	88.70
	Overall	99.33	94.75	93.83	**94.29**
C/C++	*CFG*	99.50	96.76	96.56	96.66
	PDG	98.55	83.55	90.01	86.66
	Overall	99.02	91.10	93.87	**92.46**

Table 1: Performance of NeuralPDA on **complete** code from Java and C/C++ ***(Intrinsic Evaluation)***

k	F1-Score (in %)					
	Java			*C/C++*		
	CFG	*PDG*	*Overall*	*CFG*	*PDG*	*Overall*
3	99.70	93.24	97.17	98.39	91.10	96.01
4	99.41	92.51	96.61	98.33	90.18	95.28
5	99.26	91.23	95.96	97.91	89.10	94.37
6	99.04	90.36	95.40	97.14	87.91	93.31
7	98.75	89.45	94.82	96.69	86.45	92.39
8	98.44	88.70	**94.29**	96.66	86.66	**92.46**

Table 2: Performance of NeuralPDA on **partial** code from Java and C/C++ ***(Intrinsic Evaluation)***

*The PDGs predicted by NeuralPDA approximates the performance of those generated by program analysis tools for **vulnerability detection on complete code** by **98.98%** **(Extrinsic Evaluation)***

*For the **vulnerability detection task on partial code**, the PDGs predicted by NeuralPDA helps an automated tool discover **14** real-world vulnerable code fragments **(Extrinsic Evaluation)***

Key Takeaways

- NeuralPDA is the first neural network tool to predict program dependencies in complete as well as partial programs, which are accurate as well as **380×** faster to generate.
- This work leads to a direction for improving program analysis (PA) for partial programs by combining pattern learning-based approaches with top-down PA techniques.

THE UNIVERSITY OF TEXAS AT DALLAS

Scan QR code to access the replication package, data, and supplementary material.

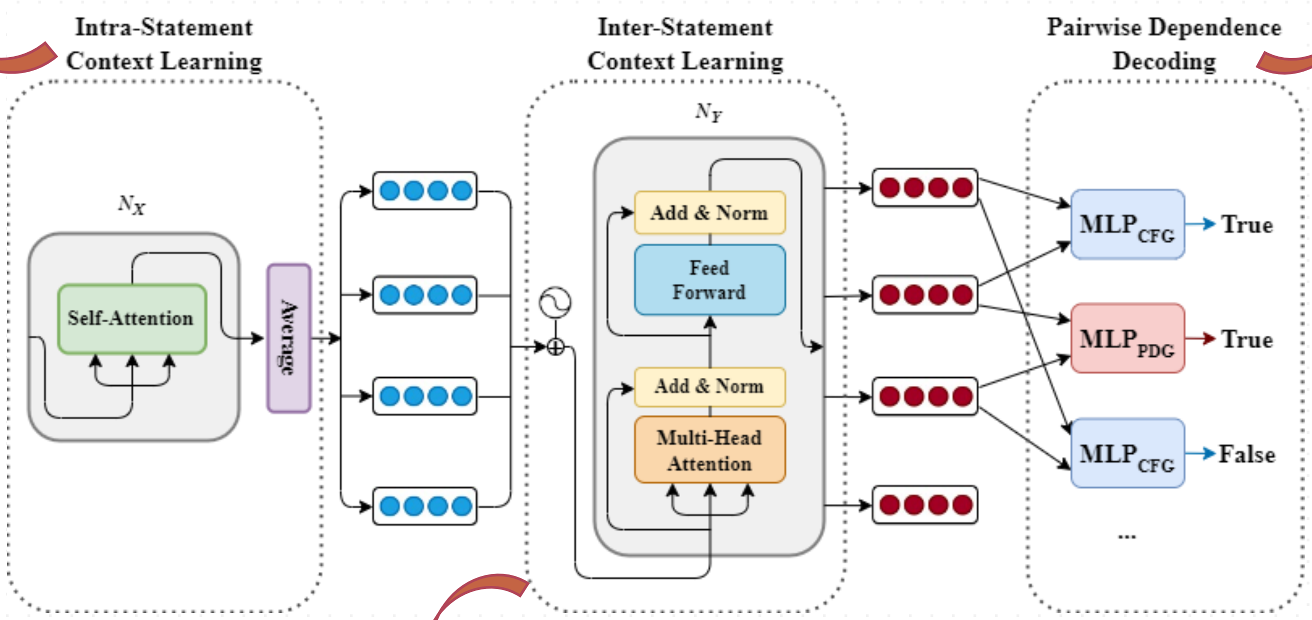
Conclusion

EXTRA SLIDES

NeuralPDA: Neural Network-Based Program Dependence Analysis

Helps ***relay the syntactic and semantic relationships*** between the tokens in a statement to other statements in the code snippet.

The combination of all the CFG/PDG edges extracted via such an arc-factored approach is ***realized as the CFG/PDG*** for the given program.



The goal of this component is to ***learn latent representations*** for each statement that model the inter-statement dependencies.

Building CFG/PDG with NEURALPDA: An Illustration

```java
List<Game> allGames = gameMapper.getAllGamesByLeague(league);
for (Game game : allGames) {
    game.getTeam1().setGame(game);
    game.getTeam2().setGame(game);
}
Collections.sort(allGames, new GameComparator());
```

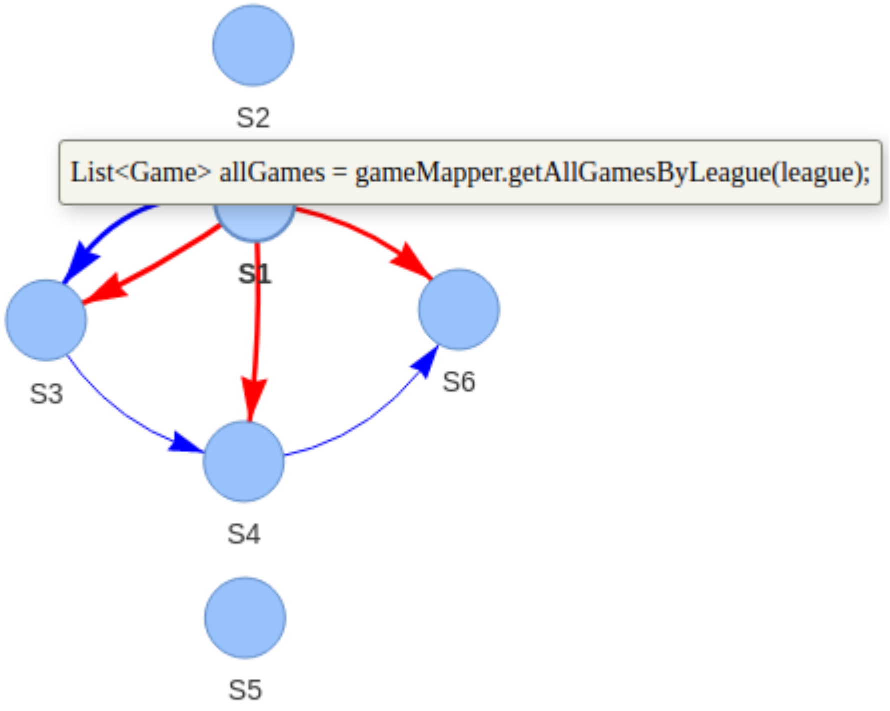


Figure 1: Partial Java code listing (left) and its corresponding **CFG**/**PDG** (right) predicted by NEURALPDA

Empirical Evaluation

P/L	Graph	Accuracy	Precision	Recall	F1-Score
Java	*CFG*	99.79	98.31	98.58	98.44
	PDG	98.87	89.89	87.53	88.70
	Overall	99.33	94.75	93.83	**94.29**
C/C++	*CFG*	99.50	96.76	96.56	96.66
	PDG	98.55	83.55	90.01	86.66
	Overall	99.02	91.10	93.87	**92.46**

Table 1: Performance of NeuralPDA on complete code from Java and C/C++ ***(Intrinsic Evaluation)***

k	F1-Score (in %)					
	Java			***C/C++***		
	CFG	*PDG*	*Overall*	*CFG*	*PDG*	*Overall*
3	99.70	93.24	97.17	98.39	91.10	96.01
4	99.41	92.51	96.61	98.33	90.18	95.28
5	99.26	91.23	95.96	97.91	89.10	94.37
6	99.04	90.36	95.40	97.14	87.91	93.31
7	98.75	89.45	94.82	96.69	86.45	92.39
8	98.44	88.70	**94.29**	96.66	86.66	**92.46**

Table 2: Performance of NeuralPDA on partial code from Java and C/C++ ***(Intrinsic Evaluation)***

*The PDGs predicted by NeuralPDA approximates the performance of those generated by program analysis tools for **vulnerability detection on complete code** by **98.98%** **(Extrinsic Evaluation)***

*For the **vulnerability detection task on partial code**, the PDGs predicted by NeuralPDA helps an automated tool discover **14** real-world vulnerable code fragments **(Extrinsic Evaluation)***

Conclusion

- NeuralPDA is the first neural network tool to predict program dependencies in complete as well as partial programs, which are accurate as well as **<u>380×</u>** faster to generate.

- This work leads to a direction for improving program analysis (PA) for partial programs by combining pattern learning-based approaches with top-down PA techniques.

All code, data, and supplementary material are available through this QR code.