44th IEEE Symposium on Security and Privacy, 2023

GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics

Shu Wang[1*], Xinda Wang[1*], Kun Sun[1], Sushil Jajodia[1], Haining Wang[2], Qi Li[3]

[1] Center for Secure Information Systems, George Mason University
[2] Virginia Tech [3] Tsinghua University

* Equal Contribution

Background

- A large volume of OSS security patches (e.g., GitHub commits fixing vulnerabilities) are **silently released**.

```
From 7f9822a48213dd2feca845dbbb6bcb8beb9550de
Subject: [PATCH] Add blinding to a DSA signature

This is based on side channel attacks demonstrated by (NCC Group)
for ECDSA which are likely to be able to be applied to DSA.
```

Not report to NVD

```
From 41bdc78544b8a93a9c6814b8bbbfef966272abbe
Subject: [PATCH] x86/tls: Validate TLS entries to protect espfix

Installing a 16-bit RW data segment into the GDT defeats espfix.
AFAICT this will not affect glibc, Wine, or dosemu at all.
```

Not provide explicit description

- Average users need to **timely detect** and apply security patches before being exploited by armored attackers.

Previous Solutions and Limitations

An OSS Patch

Natural Language Documentation (Commit Message/Changelog)

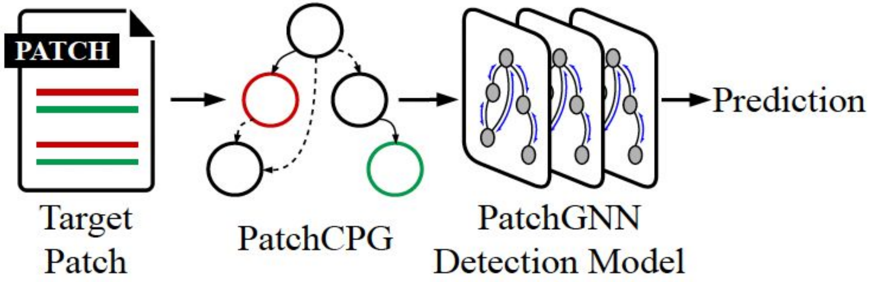
Source Code Changes

- **Mining security keywords**
  ✗ Requiring well-maintained doc.

- **Regarding code as sequential data**
  ✗ Losing important semantics.

- **Our solution: representing code as graph**
  ✓ Retaining rich patch structural info.

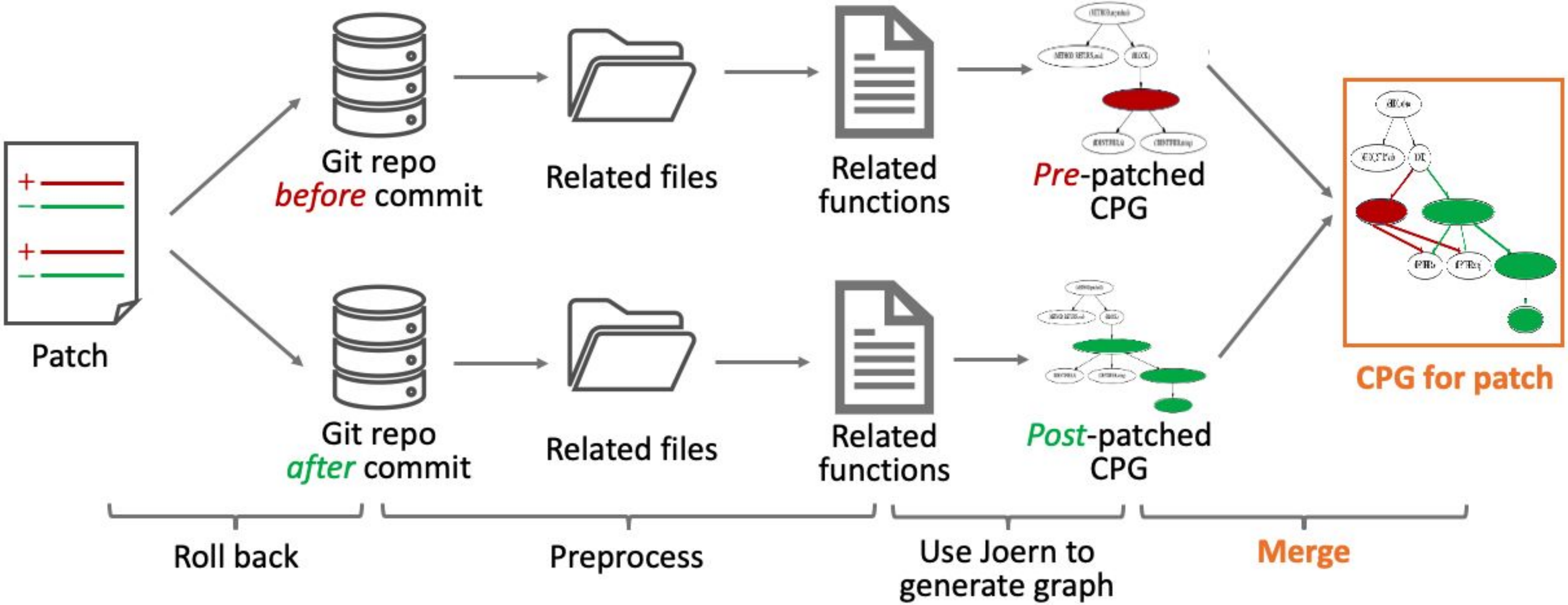
Our Solution: A Graph-Based Security Patch Detection System



- PatchCPG: a **new graph representation** of inherent code change structures.
    - Syntax and semantics: AST + control & data dependency graph.
    - Changes and relations with context: pre-patch + post-patch graph.
- PatchGNN: a **tailored GNN model** to capture diverse patch structural information.

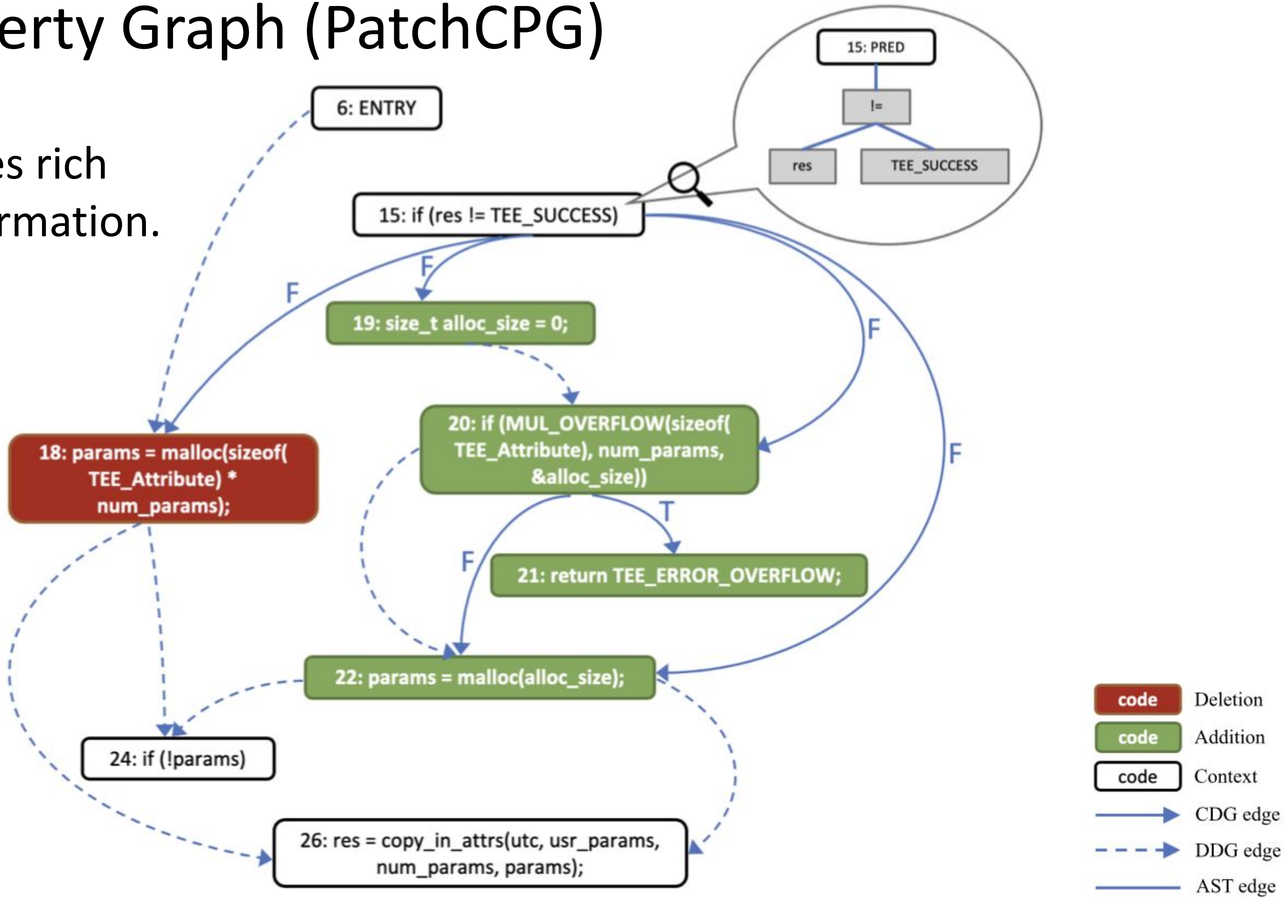
PatchCPG: From Patch to Graph

- Challenge: how to construct PatchCPG?



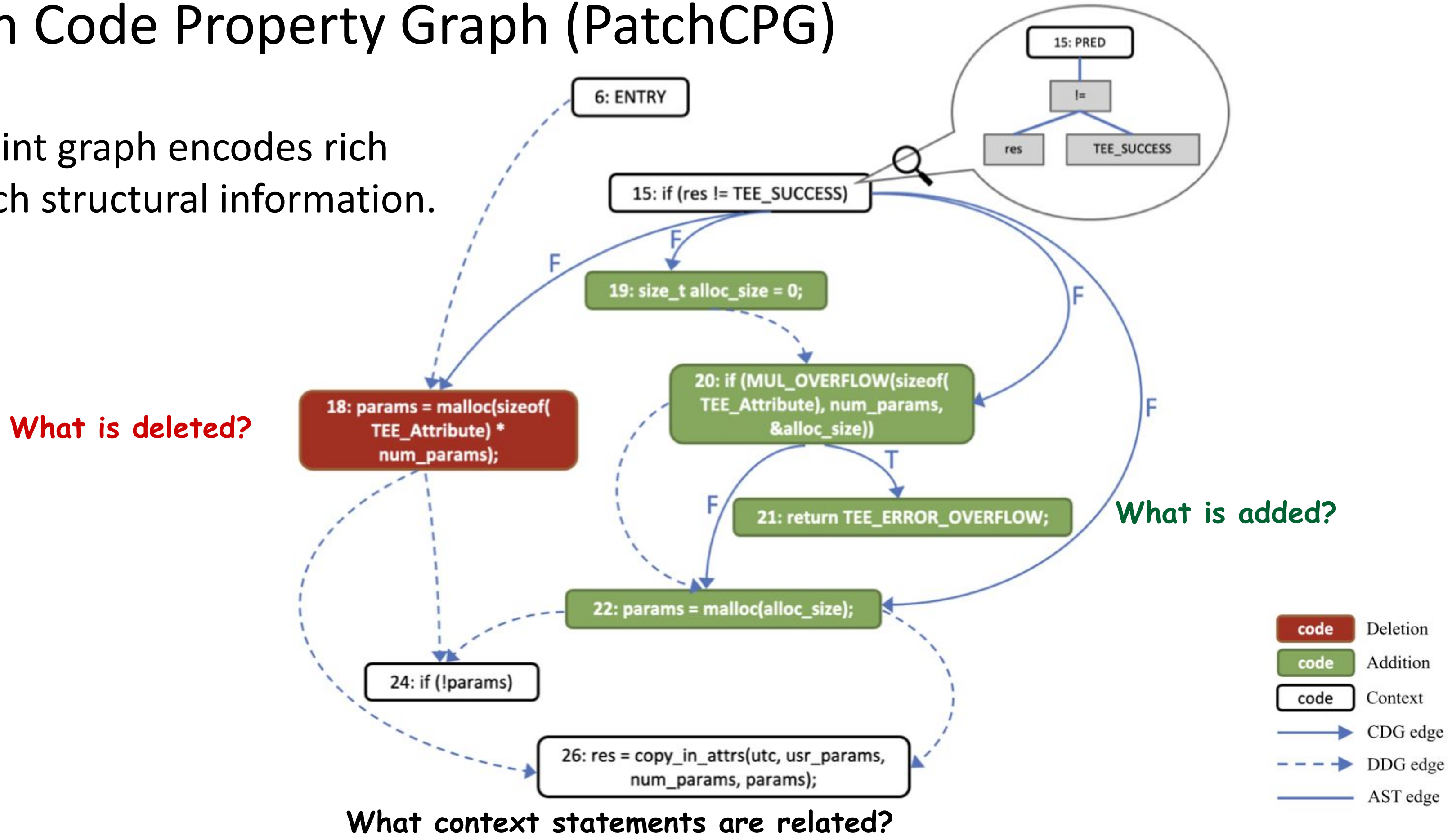
Patch Code Property Graph (PatchCPG)

- A joint graph encodes rich patch structural information.



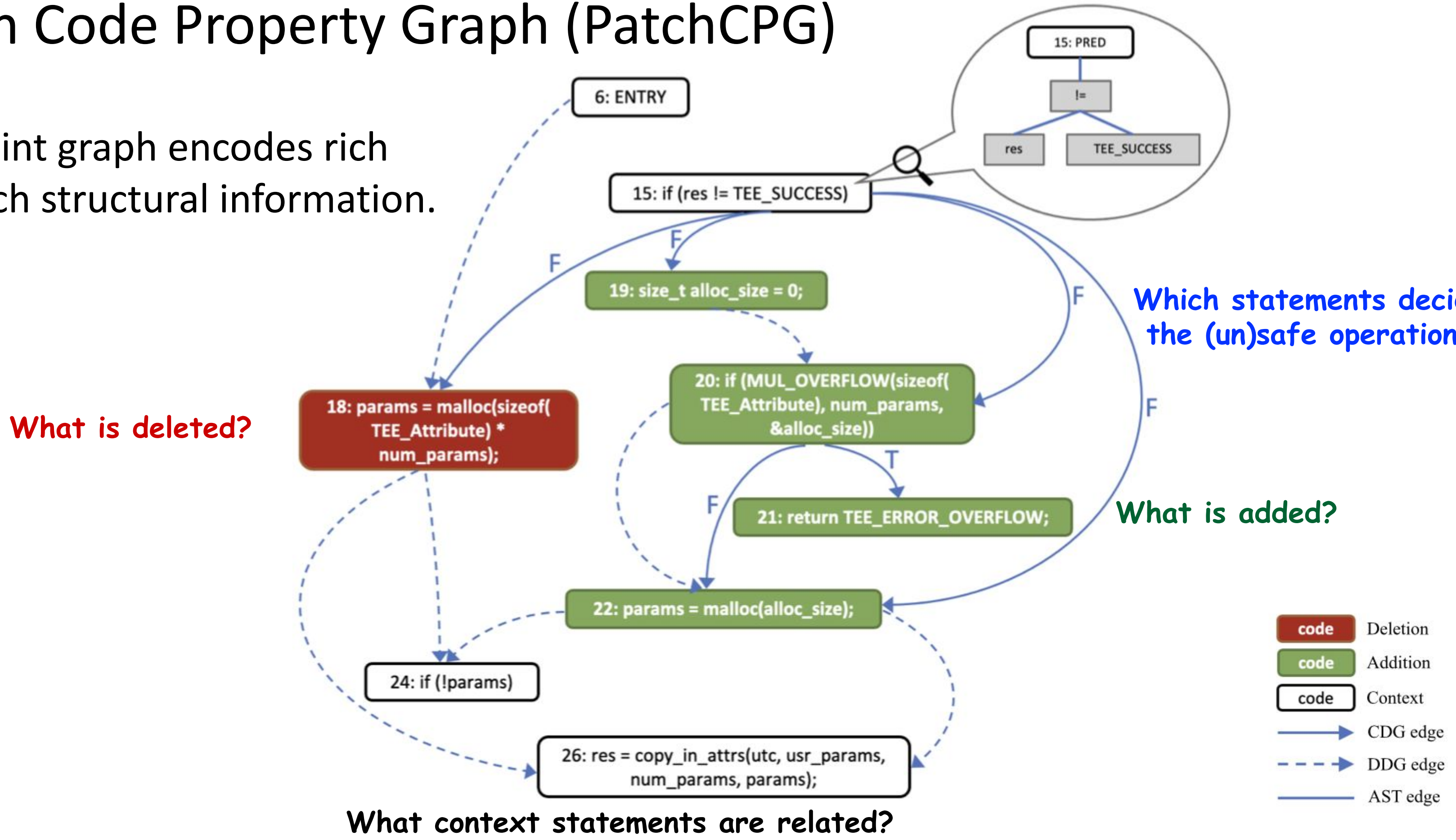
Patch Code Property Graph (PatchCPG)

- A joint graph encodes rich patch structural information.



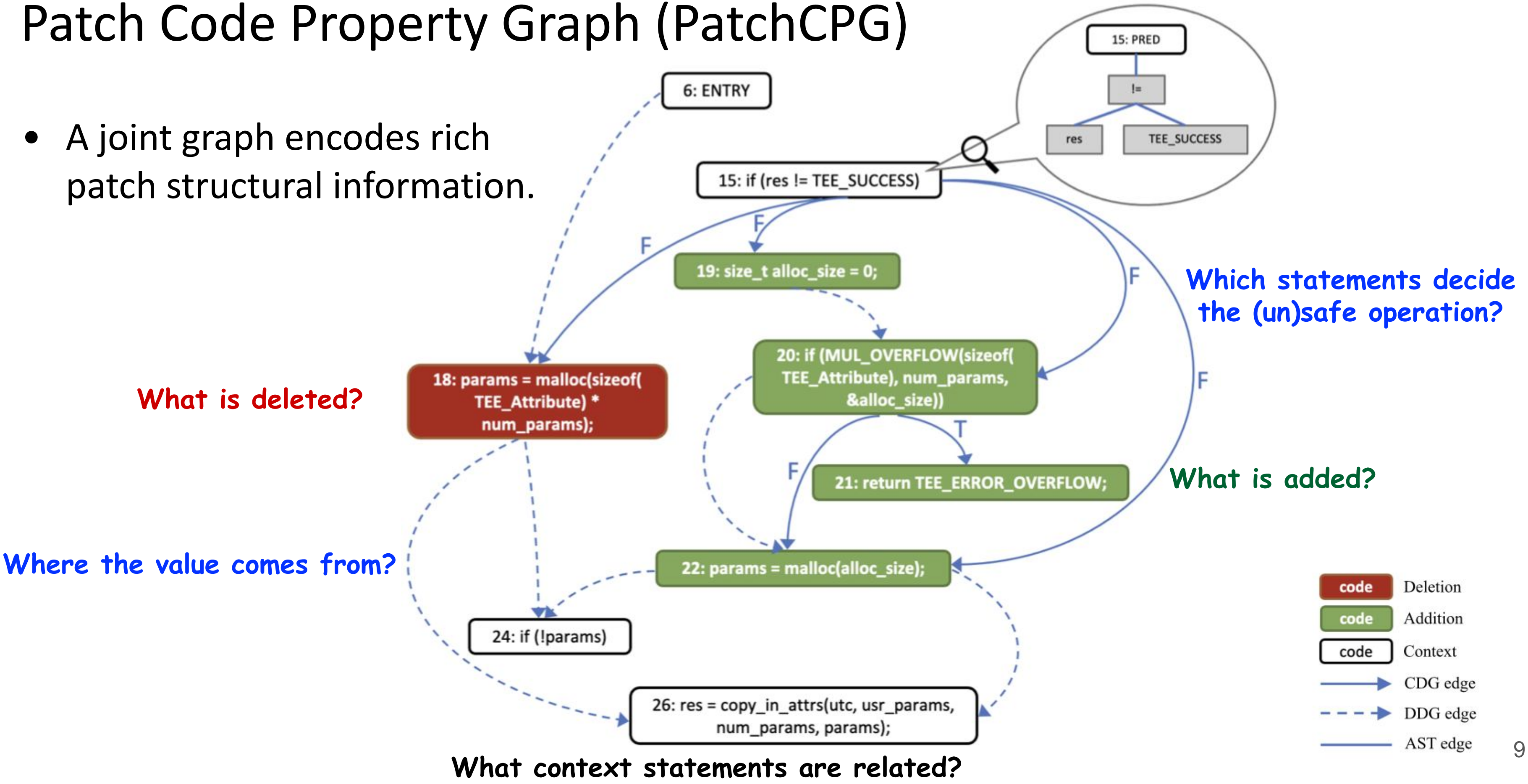
Patch Code Property Graph (PatchCPG)

- A joint graph encodes rich patch structural information.



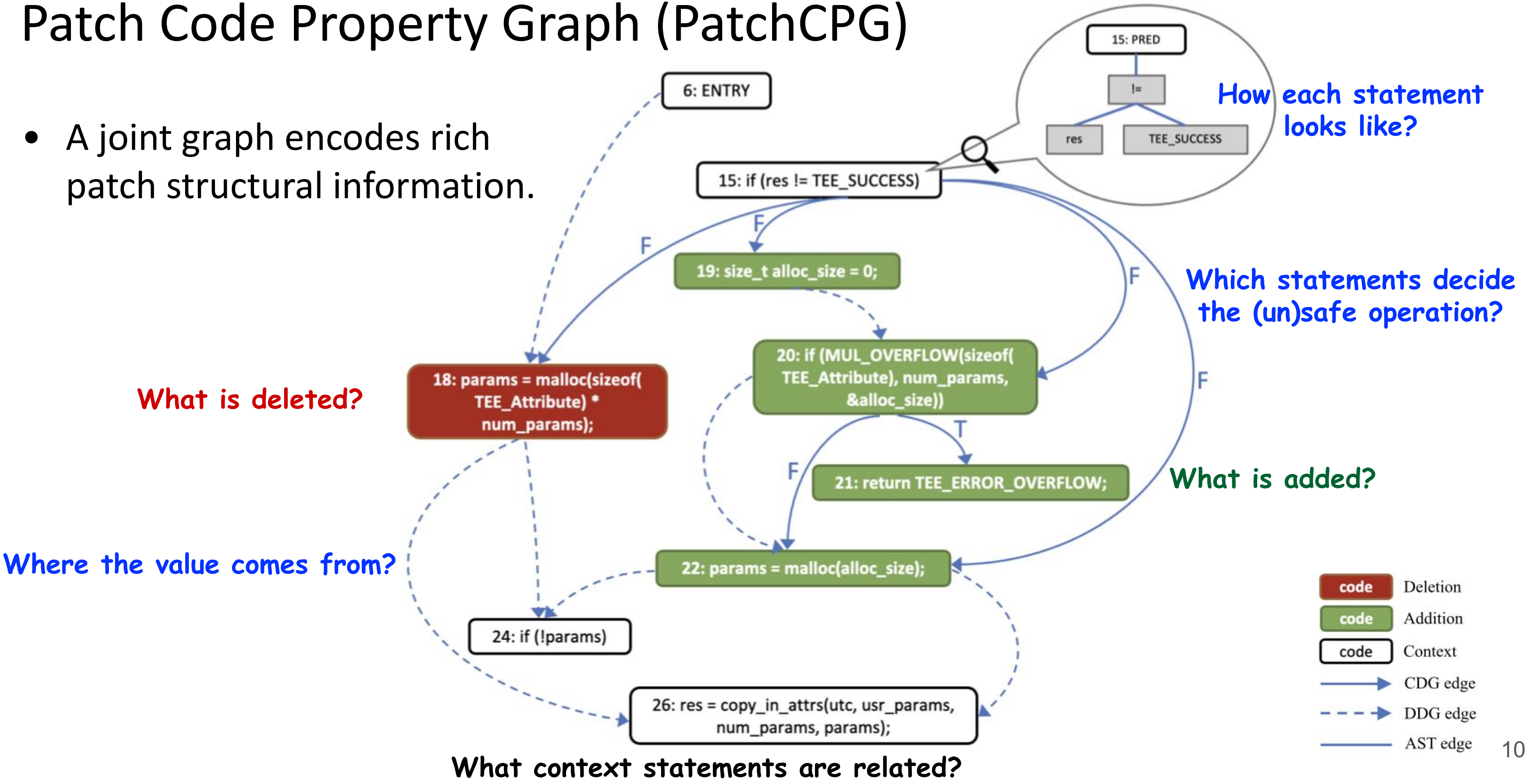
Patch Code Property Graph (PatchCPG)

- A joint graph encodes rich patch structural information.



Patch Code Property Graph (PatchCPG)

- A joint graph encodes rich patch structural information.



Reducing Noisy Information by Slicing

```
TEE_Result syscall_asymm_verify(unsigned long state,
      const struct utee_attribute *usr_params, size_t
      num_params, const void *data, size_t data_len,
      const void *sig, size_t sig_len)
{
     TEE_Result res;
     TEE_Attribute *params = NULL;
     struct user_ta_ctx *utc;

     ...

     res = tee_mmu_check_access_rights(utc,
      TEE_MEMORY_ACCESS_READ |
      TEE_MEMORY_ACCESS_ANY_OWNER, (uaddr_t)sig, sig_len)
      ;
     if (res != TEE_SUCCESS)
         return res;

-    params = malloc(sizeof(TEE_Attribute) * num_params);
+    size_t alloc_size = 0;
+    if (MUL_OVERFLOW(sizeof(TEE_Attribute), num_params,
      &alloc_size))
+         return TEE_ERROR_OVERFLOW;
+    params = malloc(alloc_size);

     if (!params)
         return TEE_ERROR_OUT_OF_MEMORY;
     res = copy_in_attrs(utc, usr_params, num_params,
      params);
     if (res != TEE_SUCCESS)
         goto out;

     ...

out:
     free(params);
     return res;
}
```

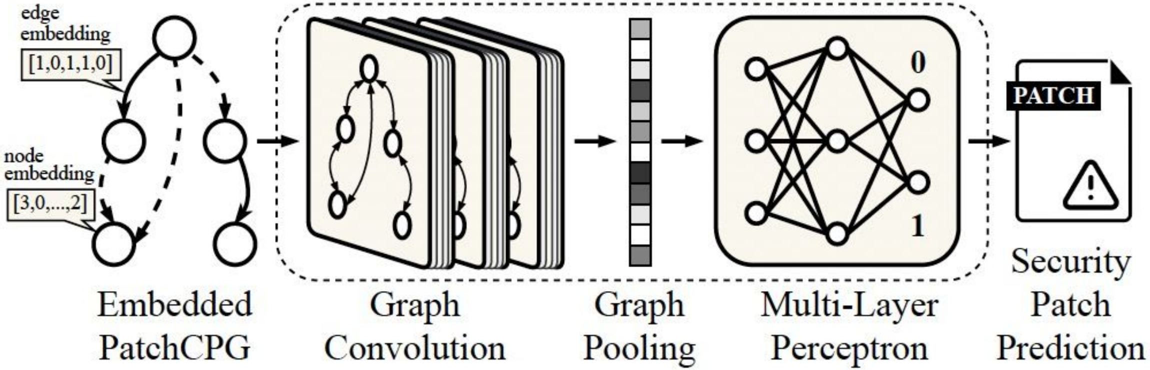
Too many statements

slicing


```
TEE_Result syscall_asymm_verify(unsigned long state,
      const struct utee_attribute *usr_params, size_t
      num_params, const void *data, size_t data_len,
      const void *sig, size_t sig_len)
{
     TEE_Result res;
     TEE_Attribute *params = NULL;
     struct user_ta_ctx *utc;

     ...

     res = tee_mmu_check_access_rights(utc,
      TEE_MEMORY_ACCESS_READ |
      TEE_MEMORY_ACCESS_ANY_OWNER, (uaddr_t)sig, sig_len)
      ;
     if (res != TEE_SUCCESS)
         return res;

-    params = malloc(sizeof(TEE_Attribute) * num_params);
+    size_t alloc_size = 0;
+    if (MUL_OVERFLOW(sizeof(TEE_Attribute), num_params,
      &alloc_size))
+         return TEE_ERROR_OVERFLOW;
+    params = malloc(alloc_size);

     if (!params)
         return TEE_ERROR_OUT_OF_MEMORY;
     res = copy_in_attrs(utc, usr_params, num_params,
      params);
     if (res != TEE_SUCCESS)
         goto out;

     ...

out:
     free(params);
     return res;
}
```


Only retain most relevant contexts

PatchGNN: Detect Security Patches from PatchCPGs

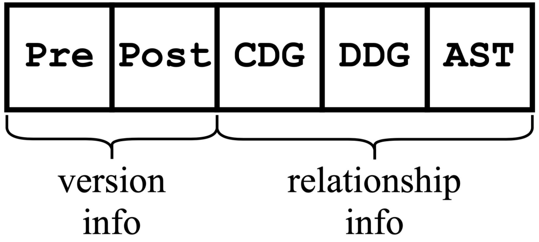


- Challenge 1: how to embed the PatchCPGs?
- Challenge 2: how to learn multiple attributes (CDG/DDG/AST/pre/post)?

PatchCPG Embeddings

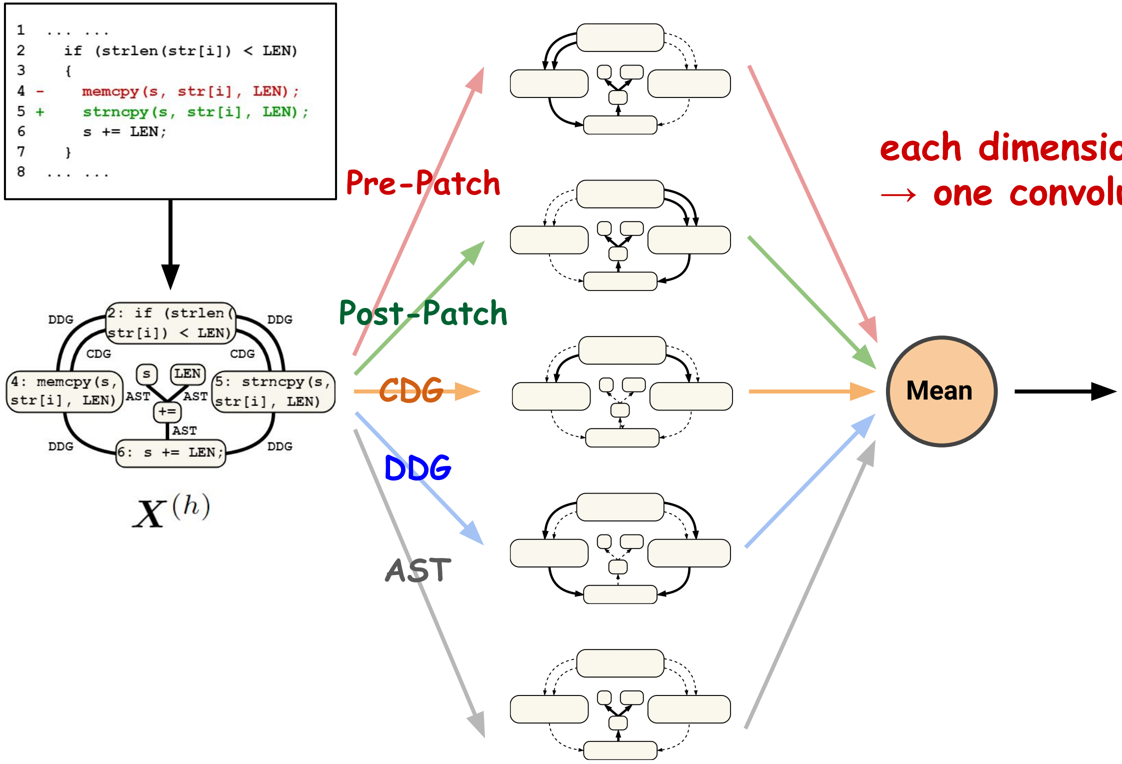
- Node Embedding
  - 20-dimensional vulnerability features.
    - code snippet metadata
    - identifier and literal features
    - control flow features
    - operator features
    - API features

- Edge Embedding
  - 5-dimensional binary vector.



e.g., [1,1,0,1,0] means the edge is a context edge of data dependency.

PatchGNN with Multi-Attribute Graph Convolution



each dimension of edge embeddings
→ one convolution channel

$$X^{(h+1)} = \frac{1}{5}\sum_{k=1}^{5} \sigma(\tilde{L}_k \cdot X^{(h)} \cdot W_k^{(h)})$$

where

$$\tilde{L}_k = \tilde{D}_k^{-1/2} \cdot \tilde{A}_k \cdot \tilde{D}_k^{-1/2}$$

$$\tilde{A}_k = A \odot M^{(k)} + I$$

$$\tilde{D}_k = diag(\textstyle\sum_j A_{\cdot j})$$

Implementation & Evaluation

Implementation

- **5K new LoC** in Scala and Python on top of *Joern* parser and *PyTorch* library.

Datasets:

- PatchDB: 12K security patches from 300+ GitHub repos.
- SPI-DB: 10K security patches from FFmpeg and QEMU.

Evaluation:

- Compared with sequential-based patch detector.
- Compared with vulnerability detection methods.
- Case study on four popular OSS repos.

Compared with Sequential-based Solution

- Accuracy 10.8%↑
- F-1 score: 0.096↑
- Precision: 28.82%↑
- False Positive Rate: 14.62%↓

Method	Dataset	General Metrics		Special Metrics	
		Accuracy	F1-score	Precision	FP Rate
TwinRNN [1][2]	PatchDB	69.60%	0.461	48.45%	19.67%
	SPI-DB	56.37%	0.512	49.07%	41.57%
GraphSPD	PatchDB	80.39%	0.557	77.27%	5.05%
	SPI-DB	63.04%	0.503	63.96%	19.16%

[1] PatchRNN: A Deep Learning-Based System for Security Patch Identification.
[2] SPI: Automated Identification of Security Patches via Commits.

Compared with Vulnerability Detection Solutions

- **2.5 - 50x** detection rate of vulnerability detectors.

Method	# $Vul_{pre\text{-}patch}$	# $Vul_{post\text{-}patch}$	# $Patch_{security}$	TP Rate
Cppcheck[3]	3	1	2	0.54%
flawfinder[4]	109	108	1	0.27%
ReDeBug[5]	29	29	0	0.00%
VUDDY[6]	22	16	21	5.71%
VulDeePecker[7]	3	0	3	0.82%
GraphSPD	-	-	53	14.40%

[3] Cppcheck. https://cppcheck.sourceforge.io.
[4] flawfinder. https://dwheeler.com/flawfinder/.
[5] Redebug: finding unpatched code clones in entire os distributions.
[6] VUDDY: A scalable approach for vulnerable code clone discovery.
[7] VulDeePecker: A deep learning- based system for vulnerability detection.

Case Study on OSS Repos

- **NGINX**: detect 21 security patches (Precision: 78%).

Changes w/	CVE	Total Commits	Valid Commits	Detected S.P.	Confirmed S.P.	Precision
1.19.x	3	180	127	7	6	86%
1.17.x	3	134	82	4	3	75%
1.15.x	1	203	120	7	4	57%
1.13.x	1	270	157	9	8	89%
Sum.	8	787	486	27	21	78%

- **Xen**: detect 29 security patches (Precision: 55%).
- **OpenSSL**: detect 45 security patches (Precision: 66%).
- **ImageMagick**: detect 6 security patches (Precision: 46.2%).

Conclusion

- Silent security patches can be leveraged by attackers to launch N-day attacks.

- GraphSPD presents patches as graphs and identifies security patches with graph learning, achieving higher accuracy and fewer false alarms.

- GraphSPD can be extended to other programming languages.

Thank you!

Contacts: {swang47, xwang44}@gmu.edu

Website: https://sunlab-gmu.github.io/GraphSPD/