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.
Introduction

- As announced by the 2021 report, 98% of codebases contain open source components
- Meanwhile, 84% of code-bases have at least one open-source vulnerability
- 60% of them contain high-risk vulnerabilities
- By exploiting the OSS vulnerabilities reported in the vulnerability databases (e.g., NVD), attackers can perform “N-day” attacks against unpatched software systems
Problem

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

- 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)

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

Source Code Changes

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

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

What is added?

What is deleted?

What context statements are related?

Which statements decide the (un)safe operation?
Patch Code Property Graph (PatchCPG)

- A joint graph encodes rich patch structural information.

What is added?

What is deleted?

What context statements are related?

Where the value comes from?

Which statements decide the (un)safe operation?
Patch Code Property Graph (PatchCPG)

- A joint graph encodes rich patch structural information.

What is added?

What is deleted?

Where the value comes from?

What context statements are related?

How each statement looks like?

Which statements decide the (un)safe operation?

What is added?
Reducing Noisy Information by Slicing

---

Too many statements

```
6 TEE Result syscall_asm_verify(unsigned long state,
    const struct utee_attribute *us_params, size_t
    num_params, const void *data, size_t data_len,
    const void *sig, size_t sig_len)
7 {
    TEE_Result res;
    TEE_Attribute *param = NULL;
    struct user_ta_ctx *utc;
    ...
    res = tee_mmu_check_access_rights(utc,
        TEE_MEMORY_ACCESS_READ | TEE_MEMORY_ACCESS_ANY_OWNER, (addr_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_sizes))
        return TEE_ERROR_OVERFLOW;
    params = malloc(alloc_size);
    if (!params)
        return TEE_ERROR_OUT_OF_MEMORY;
    res = copy_mem_ctx(utc, us_params, num_params, params);
    if (res != TEE_SUCCESS)
        goto out;
    ...
    free(params);
    return res;
}
```

slicing

```
6 TEE Result syscall_asm_verify(unsigned long state,
    const struct utee_attribute *us_params, size_t
    num_params, const void *data, size_t data_len,
    const void *sig, size_t sig_len)
7 {
    TEE_Result res;
    TEE_Attribute *param = NULL;
    struct user_ta_ctx *utc;
    ...
    res = tee_mmu_check_access_rights(utc,
        TEE_MEMORY_ACCESS_READ | TEE_MEMORY_ACCESS_ANY_OWNER, (addr_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_mem_ctx(utc, us_params, num_params, params);
    if (res != TEE_SUCCESS)
        goto 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.
### TABLE I: The involved tokens or sub-tokens of the control flow features, the operator features, and the API features.

<table>
<thead>
<tr>
<th>Features</th>
<th>Matched Tokens or Sub-tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>if, switch</td>
</tr>
<tr>
<td>loop</td>
<td>for, while</td>
</tr>
<tr>
<td>jump</td>
<td>return, break, continue, goto, throw, assert</td>
</tr>
<tr>
<td>arithmetic</td>
<td>+, -, +, -, *, /, =, +=, -=, *=, /=, %=</td>
</tr>
<tr>
<td>relational</td>
<td>==, !=, &gt;=, &lt;=, &gt;, &lt;</td>
</tr>
<tr>
<td>logical</td>
<td>&amp;&amp;,</td>
</tr>
<tr>
<td>bitwise</td>
<td>&amp;,</td>
</tr>
<tr>
<td>memory API</td>
<td>alloc, free, mem, copy, new, open, close, delete, create, release, sizeof, remove, clear, dequeue, enqueue, detach, attach</td>
</tr>
<tr>
<td>string API</td>
<td>str, string</td>
</tr>
<tr>
<td>lock API</td>
<td>lock, mutex, spin</td>
</tr>
<tr>
<td>system API</td>
<td>init, register, disable, enable, put, get, up, down, inc, dec, add, sub, set, map, stop, start, prepare, suspend, resume, connect</td>
</tr>
</tbody>
</table>

† Operator * is determined as dereference operator or arithmetic operator.
* Operator & is determined as address-of operator or bitwise operator.
PatchGNN with Multi-Attribute Graph Convolution

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

where

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

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

$\tilde{D}_k = diag(\sum_j A_{i,j})$

Each dimension of edge embeddings $\rightarrow$ one convolution channel
PatchGNN with Multi-Attribute Graph Convolution
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%↓

<table>
<thead>
<tr>
<th>Method</th>
<th>Dataset</th>
<th>General Metrics</th>
<th>Special Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Accuracy</td>
<td>F1-score</td>
</tr>
<tr>
<td>TwinRNN</td>
<td>PatchDB</td>
<td>69.60%</td>
<td>0.461</td>
</tr>
<tr>
<td>[1][2]</td>
<td>SPI-DB</td>
<td>56.37%</td>
<td>0.512</td>
</tr>
<tr>
<td>GraphSPD</td>
<td>PatchDB</td>
<td>80.39%</td>
<td>0.557</td>
</tr>
<tr>
<td></td>
<td>SPI-DB</td>
<td>63.04%</td>
<td>0.503</td>
</tr>
</tbody>
</table>

Compared with Vulnerability Detection Solutions

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

<table>
<thead>
<tr>
<th>Method</th>
<th># Vul&lt;sub&gt;pre-patch&lt;/sub&gt;</th>
<th># Vul&lt;sub&gt;post-patch&lt;/sub&gt;</th>
<th># Patch&lt;sub&gt;security&lt;/sub&gt;</th>
<th>TP Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cppcheck[3]</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0.54%</td>
</tr>
<tr>
<td>flawfinder[4]</td>
<td>109</td>
<td>108</td>
<td>1</td>
<td>0.27%</td>
</tr>
<tr>
<td>ReDeBug[5]</td>
<td>29</td>
<td>29</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>VUDDY[6]</td>
<td>22</td>
<td>16</td>
<td>21</td>
<td>5.71%</td>
</tr>
<tr>
<td>VulDeePecker[7]</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0.82%</td>
</tr>
<tr>
<td>GraphSPD</td>
<td>-</td>
<td>-</td>
<td>53</td>
<td>14.40%</td>
</tr>
</tbody>
</table>

Case Study on False Negetives


```c
commit 247d30a7dba6684ccee4508424f35fd58465e535
if (!s1->current_frame.data[0]
|| s->width != s1->width
|| s->height != s1->height) {
  if (s != s1)
    copy_fields(s, s1, golden_frame, current_frame);
  + copy_fields(s, s1, golden_frame, keyframe);
  return -1;
}
```

Listing 5: A patch with similar patterns (CVE-2018-16393).

```c
commit 360e95d45ac4123255a4c796db96337f332160ad
if (priv->cac_id_len) {
  serial->len=MIN(priv->cac_id_len, SC_MAX_SERIALNR);
  - memcpy(serial->val, priv->cac_id, priv->cac_id_len);
  + memcpy(serial->val, priv->cac_id, serial->len);
  SC_RETURN(card->ctx, SC_DEBUG_NORMAL, SC_SUCCESS);
}
```
Case Study on OSS Repos

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

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

- **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.
Acknowledgments