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
- Manifestation distance
- Level of program representation
1 short concat(char *a, char *b, char **out) {
2     short al = strlen(a);
3     short bl = strlen(b);
4     *out = (char *) malloc(al+bl);
5     if (al)
6         memcpy(*out, a, al);
7     if (bl)
8         memcpy(*out+al, b, bl);
9     return al + bl;
}
### Prior Works

**Table: Prior Works**

<table>
<thead>
<tr>
<th>Year</th>
<th>Cite</th>
<th>Name</th>
<th>Source Code</th>
<th>Structure</th>
<th>IR</th>
<th>Function</th>
<th>Control-flow</th>
<th>Data-flow</th>
<th>Generic</th>
<th>Manifestation</th>
<th>Pol</th>
<th>Region</th>
<th>Segred</th>
<th>One-hot Encoding</th>
<th>Node</th>
<th>Doc2Vec</th>
<th>Explicit Features</th>
<th>Input</th>
<th>Model</th>
<th>Utilizes Edge Type</th>
<th>Detection level</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018</td>
<td>[28]</td>
<td>Russle'18</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CNN, RF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2018</td>
<td>[23]</td>
<td>VulDeepecker</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BiLSTM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td>[40]</td>
<td>µVulDeePecker</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BiLSTM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td>[39]</td>
<td>Deviga</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GCN, DNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td>[14]</td>
<td>VGDetector</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GCN, DNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2019</td>
<td>[31]</td>
<td>NW-LCS</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LCS Scores</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020</td>
<td>[19]</td>
<td>Li'20</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020</td>
<td>[38]</td>
<td>Zagane'20</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020</td>
<td>[32]</td>
<td>Funded</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GCN, GRU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2020</td>
<td>[22]</td>
<td>SySeVR</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BiRNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2021</td>
<td>[20]</td>
<td>Li'21</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CNN + RNN, DNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2021</td>
<td>[21]</td>
<td>VulDeePlacer</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BiRNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2021</td>
<td>[13]</td>
<td>DeepWuHong</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GCN, DNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2021</td>
<td>[35]</td>
<td>Wu'21</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GCN, DNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2021</td>
<td>[9]</td>
<td>RGN4N4VD</td>
<td>Linear</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GCN, DNN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VulChecker</td>
<td>Linear</td>
<td>CPG</td>
<td>CPG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GNN (S2V)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: The table above summarizes prior works in vulnerability analysis. Each row represents a different approach, with columns indicating different features and indicators of how the approaches utilize these features.*
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
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

Source program

Resulting slice

Slicing Criterion

indirectly relevant

Slice
Program Slicing (cont.)

```java
public class SimpleExample {
    static int add(int a, int b) {
        return (a + b);
    }

    public static void main(final String[] arg) {
        int i = 1;
        int sum = 0;
        while (i < 11) {
            sum = add(sum, i);
            i = add(i, 1);
        }
        System.out.println("sum = " + sum);
        System.out.println("i = " + i);
    }
}
```

Slicing Criterion

```java
public class SimpleExample {
    static int add(int a, int b) {
        return (a + b);
    }

    public static void main(final String[] arg) {
        int i = 1;
        int sum = 0;
        while (i < 11) {
            sum = add(sum, i);
            i = add(i, 1);
        }
        System.out.println("sum = " + sum);
        System.out.println("i = " + i);
    }
}
```

Slicing Criterion
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).

$$G := (V, E, q, r)$$

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

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

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

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

Table 2: Summary of Features used in $G'_i$

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has static value?</td>
<td>Bool</td>
<td>1</td>
</tr>
<tr>
<td>Static value</td>
<td>Num</td>
<td>1</td>
</tr>
<tr>
<td>Operation {+, *, %, ...}</td>
<td>Num</td>
<td>54</td>
</tr>
<tr>
<td>Basic function {malloc, read, ...}</td>
<td>Num</td>
<td>1228</td>
</tr>
<tr>
<td>Part of IF clause</td>
<td>Categ</td>
<td>1</td>
</tr>
<tr>
<td>Number of data dependents</td>
<td>Categ</td>
<td>1</td>
</tr>
<tr>
<td>Number of control dependents</td>
<td>Categ</td>
<td>1</td>
</tr>
<tr>
<td>Betweenness centrality measure</td>
<td>Categ</td>
<td>1</td>
</tr>
<tr>
<td>Distance to $m_i$</td>
<td>Categ</td>
<td>1</td>
</tr>
<tr>
<td>Distance to nearest $r$</td>
<td>Categ</td>
<td>1</td>
</tr>
<tr>
<td>Operation of nearest $r$</td>
<td>Categ</td>
<td>54</td>
</tr>
<tr>
<td>Output dtype {int, float, ...}</td>
<td>Num</td>
<td>6</td>
</tr>
<tr>
<td>Node tag {r, m, none}</td>
<td>Num</td>
<td>2</td>
</tr>
<tr>
<td>Output type {float, pointer ...}</td>
<td>Categ</td>
<td>6</td>
</tr>
<tr>
<td>Edge type {CFG, DFG}</td>
<td>Categ</td>
<td>2</td>
</tr>
</tbody>
</table>

Total 1352
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.)

![Diagram of ePDG augmentation]

Figure 3: An illustration of an ePDG from the wild $G^{(w)}$ being augmented with a synthetic vulnerability trace from Juliet $G^{(J)}_i$. 
Overview
Evaluation
Table 3: Baseline comparison against a commercial SAST tool in detecting CVEs in the wild.

<table>
<thead>
<tr>
<th>CWE</th>
<th>VulChecker @ FPR 0.05</th>
<th>VulChecker @ FPR 0.1</th>
<th>Helix QAC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lines</td>
<td>CVEs</td>
<td>Lines</td>
</tr>
<tr>
<td></td>
<td>TP</td>
<td>FP</td>
<td>TP</td>
</tr>
<tr>
<td>190</td>
<td>9</td>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>121</td>
<td>7</td>
<td>33</td>
<td>7</td>
</tr>
<tr>
<td>122</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>415</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>416</td>
<td>4</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>24</td>
<td>100</td>
<td>17</td>
</tr>
</tbody>
</table>
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
