NDSS Symposium 2018

VulDeePecker: A Deep Learning-Based System for Vulnerability Detection

Zhen Li¹, Deqing Zou¹, Shouhuai Xu², Xinyu Ou¹, Hai Jin¹, Sujuan Wang¹, Zhijun Deng¹, Yuyi Zhong¹

¹Huazhong University of Science and Technology (HUST), Wuhan, China
²University of Texas at San Antonio (UTSA), San Antonio, USA
Automatic Software Vulnerability Detection

- Automatic detection of software vulnerabilities is an important research problem
- Static vulnerability detection tools and studies

<table>
<thead>
<tr>
<th>Flawfinder</th>
<th>RATS</th>
<th>CHECKMARX</th>
</tr>
</thead>
<tbody>
<tr>
<td>COVERITY</td>
<td>FORTIFY</td>
<td>QoBOT</td>
</tr>
<tr>
<td>ReDeBug</td>
<td>VUDDY</td>
<td>VulDeePecker (ACSAC’16)</td>
</tr>
<tr>
<td></td>
<td>(SP’17)</td>
<td>...</td>
</tr>
</tbody>
</table>
Drawbacks of Existing Approaches

✧ First, imposing intense labor of human experts
  ✔ Define features

✧ Second, incurring high false negative rates
  ✔ Two most recent vulnerability detection systems
    • VUDDY (SP’17): false negative rate = 18.2% for Apache HTTPD 2.4.23
    • VulPecker (ACSAC’16): false negative rate = 38% with respect to 455 vulnerability samples
Given the source code of a target program, how can we determine whether or not the target program is vulnerable and if so, where are the vulnerabilities?

Without asking human experts to manually define features

Without incurring a high false negative rate or false positive rate
Our Main Contribution

**Vulnerability Deep Pecker (VulDeePecker):**

A deep learning-based system for automatically detecting vulnerabilities in programs (source code)
Outline

✧ Guiding Principles
✧ Design of VulDeePecker
✧ Experiments and Results
✧ Limitations
✧ Conclusion
| Q1: How to represent software programs for deep learning-based vulnerability detection? |
| Q2: What is the appropriate granularity for deep learning-based vulnerability detection? |
| Q3: How to select a specific neural network for vulnerability detection? |
Q1: How to represent software programs for deep learning-based vulnerability detection?

Preserve the semantic relationships between the programs’ elements (e.g., data-flow and control-flow information).
Q2: What is the appropriate granularity for deep learning-based vulnerability detection?

Represented at a finer granularity than treating a program or a function as a unit.
Q3: How to select a specific neural network for vulnerability detection?

Neural networks that can cope with contexts may be suitable for vulnerability detection.

- CNN
- DBN
- RNN
- DNN
- RNN
- LSTM
- Traditional RNN
- Unidirectional LSTM
- Bidirectional LSTM
- GRU

This paper
Outline

- Guiding Principles
- **Design of VulDeePecker**
- Experiments and Results
- Limitations
- Conclusion
Overview of VulDeePecker

Learning phase:
- Input: Training programs
  - Step I: generating code gadgets from training programs
  - Step II: generating ground truth labels for code gadgets
  - Step III: transforming code gadgets into vectors for training programs
  - Step IV: training BLSTM neural network

Detection phase:
- Target programs
  - Step V: generating code gadgets from target programs (similar to Step I)
  - Step VI: transforming code gadgets into vectors for target programs (similar to Step III)
  - Step VII: classifying the code gadgets in vector representation

Output: Code gadgets are vulnerable or not
The Concept of Code Gadget

✧ A unit for vulnerability detection
✧ A number of program statements that are semantically related to each other in terms of data dependency or control dependency
✧ **Example**: vulnerabilities related to library/API function calls
Step I: Generating Code Gadgets

1 void test(char *str)
2 {
3     int MAXSIZE=40;
4     char buf[MAXSIZE];
5     if(!buf)
6         return;
7         strcpy(buf, str); /*string copy*/
8 }
9
10 int main(int argc, char **argv)
11 {
12     char *userstr;
13     if(argc > 1) {
14         userstr = argv[1];
15         test(userstr);
16     }
17     return 0;
18 }

Program source code

(1) Extract library/API function calls

(2) Generate slices for arguments of library/API function calls

(3) Assemble slices into code gadgets

A code gadget corresponding to

```
13 main(int argc, char **argv)
15 char *userstr;
18 userstr = argv[1];
19 test(userstr);
2 test(char *str)
4 int MAXSIZE=40;
5 char buf[MAXSIZE];
9 strcpy(buf, str); /*string copy*/
```
Each code gadget is labeled as “1” (i.e., vulnerable) or “0” (i.e., not vulnerable).
Step III: Transforming Code Gadgets into Vectors

✧ Transform code gadgets into their symbolic representations
✧ Encode the symbolic representations into vectors

- Transform code gadgets into their symbolic representations
- Encode the symbolic representations into vectors

Input: code gadget (from Step II.1)

13 main(int argc, char **argv)  
15 char *userstr;  
18 userstr = argv[1];  
19 test(userstr);  
2 test(char *str)  
4 int MAXSIZE=40;  
5 char buf[MAXSIZE];  
9 strcpy(buf, str); /*string copy*/

(1) Remove non-ASCII characters and comments

13 main(int argc, char **argv)  
15 char *userstr;  
18 userstr = argv[1];  
19 test(userstr);  
2 test(char *str)  
4 int MAXSIZE=40;  
5 char buf[MAXSIZE];  
9 strcpy(buf, str);

(2) Map user-defined variables

13 main(int argc, char **argv)  
15 char *VAR1;  
18 VAR1 = argv[1];  
19 test(VAR1);  
2 test(char *VAR2)  
4 int VAR3=40;  
5 char VAR4[VAR3];  
9 strcpy(VAR5, VAR2);

(3) Map user-defined functions

13 main(int argc, char **argv)  
15 char *VAR1;  
18 VAR1 = argv[1];  
19 FUN1(VAR1);  
2 FUN1(char *VAR2)  
4 int VAR3=40;  
5 char VAR4[VAR3];  
9 strcpy(VAR5, VAR2);

strncpy(VAR5, VAR2);

“strncpy”, “(” “VAR5”, “,” “VAR2”, “)”, and “,”

Vector of symbolic representation 

<table>
<thead>
<tr>
<th>Token</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy</td>
<td>$v_1$</td>
</tr>
<tr>
<td>(</td>
<td>$v_2$</td>
</tr>
<tr>
<td>VAR5</td>
<td>$v_3$</td>
</tr>
<tr>
<td>,</td>
<td>$v_4$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

7 tokens
Step IV: Training the BLSTM Neural Network

- Training process for learning the BLSTM neural network is standard.
Steps V-VII: Detection Phase

Input:
- Target programs
- Trained BLSTM neural network

Output:
- Code gadgets are vulnerable or not

Flow:
1. Step V: generating code gadgets from target programs (similar to Step I)
2. Step VI: transforming code gadgets into vectors for target programs (similar to Step III)
3. Step VII: classifying the code gadgets in vector representation
Outline

- Guiding Principles
- Design of VulDeePecker
- Experiments and Results
- Limitations
- Conclusion
Research Questions

RQ1: Can VulDeePecker deal with multiple types of vulnerabilities at the same time?

RQ2: Can human intelligence (other than defining features) improve the effectiveness of VulDeePecker?

RQ3: How effective is VulDeePecker when compared with other approaches?

✧ Metrics for evaluation

✓ False positive rate (FPR), false negative rate (FNR), recall, precision, F-measure
Preparing Input to VulDeePecker

✧ Programs collection for answering the RQs
  ✓ Two sources of vulnerability data
    • 19 C/C++ open source products which vulnerabilities are described in NVD, and C/C++ test cases in SARD
  ✓ Collect 520 open source software program files and 8,122 test cases for the buffer error vulnerability (i.e., CWE-119), and 320 open source software program files and 1,729 test cases for the resource management error vulnerability (i.e., CWE-399)

✧ Training programs vs. target programs
  ✓ Randomly choose 80% of the programs we collect as training programs and the rest 20% as target programs
Learning BLSTM Neural Networks

.paginator

 Duchess for answering the RQs

 ✔ Code Gadget Database (CGD): 61,638 code gadgets

 ✔ Six datasets of CGD

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Code gadgets</th>
<th>#Vulnerable code gadgets</th>
<th>#Not vulnerable code gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE-ALL</td>
<td>35,753</td>
<td>9,440</td>
<td>29,313</td>
</tr>
<tr>
<td>RM-ALL</td>
<td>21,885</td>
<td>7,285</td>
<td>14,600</td>
</tr>
<tr>
<td>HY-ALL</td>
<td>61,638</td>
<td>17,725</td>
<td>43,913</td>
</tr>
<tr>
<td>BE-SEL</td>
<td>26,720</td>
<td>8,119</td>
<td>18,601</td>
</tr>
<tr>
<td>RM-SEL</td>
<td>16,198</td>
<td>6,573</td>
<td>9,625</td>
</tr>
<tr>
<td>HY-SEL</td>
<td>42,918</td>
<td>14,692</td>
<td>28,226</td>
</tr>
</tbody>
</table>

Table I. Datasets for answering the RQs

BE: Buffer error vulnerabilities
RM: Resource management vulnerabilities
HY: Hybrid of the above two types of vulnerabilities
ALL: All library/API function calls
SEL: Manually selected library/API function calls
RQ1: Can VulDeePecker deal with multiple types of vulnerabilities at the same time?

✦ Insight: VulDeePecker can detect multiple types of vulnerabilities, but the effectiveness is sensitive to the amount of data (which is common to deep learning).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>FPR(%)</th>
<th>FNR(%)</th>
<th>TPR(%)</th>
<th>P(%)</th>
<th>F1(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE-ALL</td>
<td>2.9</td>
<td>18.0</td>
<td>82.0</td>
<td>91.7</td>
<td>86.6</td>
</tr>
<tr>
<td>RM-ALL</td>
<td>2.8</td>
<td>4.7</td>
<td>95.3</td>
<td>94.6</td>
<td>95.0</td>
</tr>
<tr>
<td>HY-ALL</td>
<td>5.1</td>
<td>16.1</td>
<td>83.9</td>
<td>86.9</td>
<td>85.4</td>
</tr>
</tbody>
</table>

RM: 16 function calls related to vulnerabilities
BE: 124 function calls related to vulnerabilities
RQ2: Can human intelligence (other than defining features) improve the effectiveness of VulDeePecker?

✧ Insight: Human expertise can be used to select function calls to improve the effectiveness of VulDeePecker.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>FPR(%)</th>
<th>FNR(%)</th>
<th>TPR(%)</th>
<th>P(%)</th>
<th>F1(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HY-ALL</td>
<td>5.1</td>
<td>16.1</td>
<td>83.9</td>
<td>86.9</td>
<td>85.4</td>
</tr>
<tr>
<td>HY-SEL</td>
<td>4.9</td>
<td>6.1</td>
<td>93.9</td>
<td>91.9</td>
<td>92.9</td>
</tr>
</tbody>
</table>
Insight: A deep learning-based vulnerability detection system can be more effective by taking advantage of the data-flow information.
RQ3: How effective is VulDeePecker when compared with other approaches?

 לפני כל שיטה הוחלט שלאכול ניסיונות ידניים

✧ Insight: VulDeePecker is more effective than code similarity-based approaches
VulDeePecker detected 4 vulnerabilities, which were not reported in the NVD, but were “silently” patched by the vendors.

These vulnerabilities are missed by most of the other vulnerability detection systems mentioned above.

<table>
<thead>
<tr>
<th>Target product</th>
<th>CVE ID</th>
<th>Vulnerable product published in the NVD</th>
<th>Vulnerability publish time</th>
<th>Vulnerable file in target product</th>
<th>Library/API function call</th>
<th>1st patched version of target product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libav 10.1</td>
<td>CVE-2013-0851</td>
<td>FFmpeg</td>
<td>12/07/2013</td>
<td>libavcodec/eamad.c</td>
<td>memset</td>
<td>Libav 10.3</td>
</tr>
<tr>
<td>Seamonkey 2.31</td>
<td>CVE-2015-4517</td>
<td>Firefox</td>
<td>09/24/2015</td>
<td>../dom/system/gonk/NetworkUtils.cpp</td>
<td>snprintf</td>
<td>Seamonkey 2.38</td>
</tr>
<tr>
<td></td>
<td>CVE-2015-4513</td>
<td>Firefox</td>
<td>11/05/2015</td>
<td>../netwerk/protocol/Http2Stream.cpp</td>
<td>memset</td>
<td>Seamonkey 2.39</td>
</tr>
<tr>
<td>Xen 4.6.0</td>
<td>CVE-2016-9104</td>
<td>Qemu</td>
<td>12/09/2016</td>
<td>tools/qemu-xen/hw/9pts/virtio-9p.c</td>
<td>memcpy</td>
<td>Xen 4.9.0</td>
</tr>
</tbody>
</table>
Outline

 plaza Guiding Principles plaza Design of VulDeePecker plaza Experiments and Results plaza Limitations plaza Conclusion
Limitations and Open Problems

✿ Present design
  ✓ Assuming source code is available
  ✓ Only dealing with C/C++ programs
  ✓ Only dealing with vulnerabilities related to library/API function calls
  ✓ Only accommodating data-flow information, but not control-flow information
  ✓ Using some heuristics

✿ Present implementation
  ✓ Limit to the BLSTM neural network

✿ Present evaluation
  ✓ The dataset only contains vulnerabilities about buffer errors and resource management errors
Outline

- Guiding Principles
- Design of VulDeePecker
- Experiments and Results
- Limitations
- Conclusion
We initiate the study of using deep learning for vulnerability detection, and discuss some preliminary guiding principles.

We present VulDeePecker, and evaluate it from 3 perspectives.

We present the first dataset for evaluating deep learning-based vulnerability detection systems.

https://github.com/CGCL-codes/VulDeePecker
New Results (after finishing the paper; in submission)

✧ Cope with **all kinds of vulnerabilities** (including library/API function calls related ones)

✧ Accommodate both **data dependency and control dependency**

✧ Detect 7 (potential) 0-day vulnerabilities and 8 silently patched vulnerabilities from 4 software products

✧ Some deep neural networks are more powerful than others
Takeaways

✧ The first deep learning-based vulnerability detection system using a finer-granularity unit code gadget
✧ Guiding principles for deep learning-based vulnerability detection
✧ The first dataset for evaluating deep learning-based vulnerability detection systems
Thanks!

lizhen_hust@hust.edu.cn

Data available at:
https://github.com/CGCL-codes/VulDeePecker