Key Technologies of Vulnerability Detection Based on Source Code

Deqing Zou

Huazhong University of Science and Technology
Information security incidents triggered by software emerge in endlessly.

- 2013.6 PRISM
- 2015.1 Ghost
- 2016.5 ImageTragick
- 2016.10 US DDOS
- 2014.4 Heartbleed
- 2015.7 Parasitic Beast
- 2016.10 Dirty Cow
- 2017.5 WannaCry
Information security incidents triggered by software emerge in endlessly.

Software vulnerabilities are the root of most of the information security incidents.

A software vulnerability is an instance of a flaw, caused by a mistake in the design, development, or configuration of software such that it can be exploited to violate some explicit or implicit security policy.

- Allow an attacker to execute dangerous commands
- Allow an attacker to access sensitive data
- Allow an attacker to pose as another entity
- Allow an attacker to conduct a denial of service
Background

- **Information security incidents** triggered by software emerge in endlessly.
- **Software vulnerabilities** are the root of most of the information security incidents.
- **Large amounts of open source code and vulnerability databases** provide a great opportunity for effective vulnerability detection.
Background

- Information security incidents triggered by software emerge in endlessly.
- Software vulnerabilities are the root of most of the information security incidents.
- Large amounts of open source code and vulnerability databases provide a great opportunity for effective vulnerability detection.
- Static vulnerability detection has been widely used.

Open source tools and commercial products:

- Flawfinder
- CHECKMATIC
- Sourcebrella

Incomplete feature, high false positives and false negatives.
Vulnerability Detection Based on Source Code

Input
- Source code (GitHub, SourceForge, etc.)
- Published vulnerabilities (NVD, CNVD, SARD, etc.)

Vulnerability detection
- Code similarity-based
- Pattern-based

Lexical and syntax analysis

Output
- Vulnerabilities (CVE-2017-9991, CVE-2017-7863, CVE-2016-6164, CVE-2016-7502, etc.)
- Vulnerability verification
- Detect a type of vulnerabilities
- Detect similar vulnerable code
- Buffer overflow
- Memory corruption
- SQL injection
- DoS
- ...
1. Similar Vulnerable Code Detection

Background

- Open Source Software (OSS) projects have linear to quadratic growth patterns

The number of registered open source projects in SourceForge increased from 136 K to 430 K between October 2009 and March 2014.

For Github, 10 millionth repository had been created in December 2013. It had over 85 million projects in March 2017.

- As OSS programs are widely used as codebase in software development (e.g., libraries), code clone is becoming one of the major causes of software vulnerabilities.
What is code clone?

- Code fragment that is copied and pasted within or between software systems.

Types of code clones:

- Type 1: Exact clones
- Type 2: Renamed clones
- Type 3: Near miss clones
- Type 4: Semantic clones
Problem Description

- A vulnerability may exist in multiple software programs because of code reuse

**Similar vulnerable code detection:**
Given a vulnerability and the source code of a target program, can we automatically detect whether the program contains the vulnerability or not? If so, where is it?
Process of Vulnerable Code Detection

Input

- Published vulnerable code
- Target programs

Select the code-fragment level

Represent target programs

Represent the vulnerable code

Search similar vulnerable code

Output

Locations of vulnerabilities

Input

Output
Three Components

- **Code-fragment level**
  - Patch-without-context
  - Slice
  - Patch-with-context
  - Function
  - File/component

- **Comparison method**
  - Vector comparison
  - Approximate/exact matching

- **Code representation**
  - Text-based
  - Metric-based
  - Token-based
  - Tree-based
  - Graph-based
An Example---VUDDY

- Three components
  - Code-fragment level: Function
  - Code representation: Token-based
  - Comparison method: Exact matching (Hash)

- An open service

- VUDDY is only used for

SP’17- VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery
Two Challenges

Challenge 1: No single code-similarity algorithm fits all vulnerabilities. It is not known which code-similarity algorithms would be effective for which vulnerabilities.

- Each vulnerability has its own characteristics

Challenge 2: No readily available datasets for the research.
For Challenge 1: Vulnerability Feature

- Vulnerability feature = basic features + patch features
  - **Basic features**: Obtained from NVD

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Basic features</td>
</tr>
<tr>
<td>1-1</td>
<td>CVE-ID</td>
</tr>
<tr>
<td>1-2</td>
<td>CWE-ID</td>
</tr>
<tr>
<td>1-3</td>
<td>Product vendor</td>
</tr>
<tr>
<td>1-4</td>
<td>Product affected</td>
</tr>
<tr>
<td>1-5</td>
<td>Vulnerability severity</td>
</tr>
</tbody>
</table>
For Challenge 1: Vulnerability Feature

- **Vulnerability feature = basic features + patch features**
  - **Basic features**: Obtained from NVD
  - **Patch features**: Extracted from diff

```
diff --git a/arch/x86/crypto/aesni-intel glue.c b/arch/x86/crypto/aesni-intel glue.c
index 947c6bf..54f60ab 100644
--- a/arch/x86/crypto/aesni-intel glue.c
+++ b/arch/x86/crypto/aesni-intel glue.c
@@ -1155,7 +1155,7 @@ static int __driver_rfc4106_decrypt(struct aead_request *req)
     src = kmalloc(req->cryptlen + req->assoclen, GFP_ATOMIC);
     if (!src)
         return -ENOMEM;
-    assoc = (src + req->cryptlen + auth_tag_len);
+    assoc = (src + req->cryptlen);
    scatterwalk_map_and_copy(src, req->src, 0, req->cryptlen, 0);
    scatterwalk_map_and_copy(assoc, req->assoc, 0,
                                req->assoclen, 0);
@@ -1180,7 +1180,7 @@ static int __driver_rfc4106_decrypt(struct aead_request *req)
                 scatterwalk_done(&src_sg_walk, 0, 0);
                 scatterwalk_done(&assoc_sg_walk, 0, 0);
             } else {
                 scatterwalk_map_and_copy(dst, req->dst, 0, req->cryptlen, 1);
                 scatterwalk_map_and_copy(dst, req->dst, 0, tempCipherLen, 1);
                 kfree(src);
             }
         return retval;
```

Diff of CVE-2015-3331
For Challenge 1: Vulnerability Feature

- **Vulnerability feature** = basic features + patch features
  - **Basic features**: Obtained from NVD
  - **Patch features**: Extracted from diff

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Non-substantive features</td>
</tr>
<tr>
<td>2-1</td>
<td>Changes in whitespace, format or comment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Component features</td>
</tr>
<tr>
<td>3-1</td>
<td>Variable name modification</td>
</tr>
<tr>
<td>3-2</td>
<td>Constant modification</td>
</tr>
<tr>
<td>3-3</td>
<td>Variable type modification</td>
</tr>
<tr>
<td>3-4</td>
<td>Function name modification</td>
</tr>
<tr>
<td>3-5</td>
<td>Function argument addition</td>
</tr>
<tr>
<td>3-6</td>
<td>Function argument deletion</td>
</tr>
<tr>
<td>3-7</td>
<td>Function argument modification</td>
</tr>
<tr>
<td>3-8</td>
<td>Variable declaration addition</td>
</tr>
<tr>
<td>3-9</td>
<td>Variable declaration deletion</td>
</tr>
<tr>
<td>3-10</td>
<td>Operator modification</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Expression features</td>
</tr>
<tr>
<td>4-1</td>
<td>Assignment expression modification</td>
</tr>
<tr>
<td>4-2</td>
<td>if condition modification</td>
</tr>
<tr>
<td>4-3</td>
<td>for condition modification</td>
</tr>
<tr>
<td>4-4</td>
<td>while condition modification</td>
</tr>
<tr>
<td>4-5</td>
<td>do while condition modification</td>
</tr>
<tr>
<td>4-6</td>
<td>switch condition modification</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Statement features</td>
</tr>
<tr>
<td>5-1</td>
<td>Line addition</td>
</tr>
<tr>
<td>5-2</td>
<td>Line deletion</td>
</tr>
<tr>
<td>5-3</td>
<td>Line movement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Function-level features</td>
</tr>
<tr>
<td>6-1</td>
<td>Entire function addition</td>
</tr>
<tr>
<td>6-2</td>
<td>Entire function deletion</td>
</tr>
<tr>
<td>6-3</td>
<td>Modification beyond the function</td>
</tr>
</tbody>
</table>
For Challenge 2: Creating Datasets

- For 19 C/C++ open source projects
  - Vulnerability Patch Database (VPD): 1,761 vulnerabilities with 3,454 diff hunks
  - Vulnerability Code Instance Database (VCID): 455 vulnerable code reuse instances
Overview of VulPecker

- **VulPecker**: A system for automatically detecting whether a program contains a vulnerability or not.
Overview of VulPecker

- The learning phase selects code-similarity algorithm(s) that is effective for a vulnerability.
- The select algorithms in turn guide the generation of vulnerability signatures and the detection of vulnerabilities.
Some Experimental Results

- Comparing existing code-similarity algorithms and their variants against VulPecker

**VulPecker has a higher F-measure --- Code similarity algorithm should be selected according to the vulnerability diff hunk features.**
Some Experimental Results

- Detecting vulnerabilities in products
  - We select 246 vulnerabilities (published between 2013 and 2015) for Firefox, Ffmpeg, and Qemu
  - 40 vulnerabilities are detected in target programs by VulPecker, but were not published in the NVD!
    - 22 vulnerabilities --- “silently” patched by product vendors (after 7.3 months on average since they were published)
2. Pattern-based Vulnerability Detection

VulDeePecker: A Deep Learning-based System for Automatic Vulnerability Detection, **NDSS 2018**
The number of serious vulnerabilities discovered in deployed software is on the rise.

In 2010, around 4,600 CVEs were registered. In 2016, there were about 6,500.

Many static vulnerability detection tools and studies

- Open source tools, commercial tools, academic research projects

Flawfinder  RATS  CHECKMARX

COVERITY  FORTIFY  QoBOT

Sourcebrella  ReDeBug  VUDDY
Problem Description

Existing tools
- Flawfinder
- CHECKMARX
- COVERITY
- VUDDY
- Chucky
- ...

Rely on human experts to generate patterns, or define features, or generate patterns automatically for a certain type of vulnerabilities

Subjective;
High false positives and false negatives;
Poor extensibility

How to automatically learn to generate vulnerability patterns to improve the effectiveness?

Given the source code of a target program, how can we determine whether or not the target program is vulnerable, and if so, where the vulnerabilities lie?
Using Deep Learning for Program Analysis

- No studies on vulnerability detection using deep learning

ICSE’16-Automatically learning semantic features for defect prediction
QRS’15- Deep learning for just-in-time defect prediction
Object Detection (1)

- **Inspired by the object detection in computer vision**
  - **Object detection**: Given an input image, the locations and categories of objects are detected from the complex background image.
  - **Object detection based on region proposal**: The object detection makes use of the texture, edge, color and other information in the image to extract the region proposals.
Object Detection (2)

- **A simple object detection process**
  - **Step 1:** Input image
  - **Step 2:** Extract region proposals (~2k)
  - **Step 3:** Compute CNN features
  - **Step 4:** Classify regions

**R-CNN: Regions with CNN features**
Three challenges when applying the object detection in computer vision to vulnerability detection

Challenge 1: Object detection makes use of the texture, edge, color and other information in the image to extract region proposals. But there is no similar information for program code to be used directly, thus we need to propose an efficient strategy to obtain fine-grained code fragment proposals.

Challenge 2: Object detection has large datasets. But there are not large-scale vulnerability datasets with labels which demonstrate vulnerability types or whether there are vulnerabilities or not.

Challenge 3: Object detection is based on the CNN model which is suitable for image processing. By contrast, vulnerability detection pays more attention on the context of statements. Therefore, other deep learning models need to be selected for vulnerability detection.
Library/API Function Calls Related Vulnerabilities

- Take library/API function calls related vulnerabilities for example
- Vulnerabilities caused by improper use of library/API function calls exist in multiple types of vulnerabilities

According to SARD in May 2017, library/API function calls appear in the vulnerable statements of 73.0% of the 44,855 vulnerable test cases of C/C++ programs corresponding to buffer error vulnerabilities, and 33.4% of the 13,601 vulnerable test cases for resource management error vulnerabilities.
A **code gadget** is composed of a number of program statements, which are semantically related to each other via (for example) data flows and control flows.

**Generating code gadget**

```c
void test(char *str)
{
    char buf[15];
    int i;
    /* ad-hoc strlen */
    for(i = 0; str[i]; i++)
        continue;
    if (i > 16)
        return;
    strcpy(buf, str); /*string copy*/
    printf("result: %s\n", buf);
}

int main(int argc, char **argv)
{
    char *userstr;
    if(argc > 1) {
        userstr = argv[1];
        test(userstr);
    }
    return 0;
}
```

**Program source code**

**Extracting library/API function calls**

**Generating slices of library/API function arguments**

**Assembling slices into code gadgets**
For Challenge 2: Creating Dataset

- Each code gadget is labeled as “1” (vulnerable) and “0” (not vulnerable) according to the vulnerability information from NVD and SARD database.

- 61,638 code gadgets, including 17,725 code gadgets and 43,913 code gadgets that are not vulnerable.

<table>
<thead>
<tr>
<th>Code gadget</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code gadget 1</td>
<td>1</td>
</tr>
<tr>
<td>Code gadget 2</td>
<td>0</td>
</tr>
<tr>
<td>Code gadget 3</td>
<td>1</td>
</tr>
<tr>
<td>Code gadget 4</td>
<td>0</td>
</tr>
<tr>
<td>Code gadget 5</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
For Challenge 3: Neural Network Selection

- Various neural networks
  - Image processing: CNN, DBN
  - Object detection: CNN, RNTN
  - Text processing: RNN
  - Speech recognition: RNN
  - ...

Neural network

- CNN
- DBN
- RNN
- RNTN
- ...

RNNS have been very successful in natural language processing. They are effective in coping with sequential data, and indeed have been used for program analysis.
Traditional RNN suffers from the Vanishing Gradient (VG) problem. Since the GRU does not outperform the LSTM on language modeling (ICML’15), we select LSTM.
The statement of a program may be affected by earlier statements in the program and may be also affected by the later statements. Therefore, we select Bidirectional LSTM (BLSTM).
Overview of VulDeePecker

- VulDeePecker, the first deep learning-based vulnerability detection system
- Code Gadget Database (61,638 code gadgets)
- BLSTM neural network
Some Experimental Results

RQ1: Can VulDeePecker deal with multiple types of vulnerabilities?
Insight: VulDeePecker can detect multiple types of vulnerabilities.

RQ2: Can human expertise (other than defining features) improve the effectiveness of VulDeePecker?
Insight: The effectiveness can be further improved by incorporating human expertise.

RQ3: How effective is VulDeePecker when compared with other approaches?
Insight: VulDeePecker is more effective than other pattern-based static analysis tools and code similarity-based tools.

<table>
<thead>
<tr>
<th>System</th>
<th>Dataset</th>
<th>FPR (%)</th>
<th>FNR (%)</th>
<th>TPR (%)</th>
<th>P (%)</th>
<th>F1 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VulDeePecker vs. Other pattern-based vulnerability detection systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flawfinder</td>
<td>BE-SEL</td>
<td>44.7</td>
<td>69.0</td>
<td>31.0</td>
<td>25.0</td>
<td>27.7</td>
</tr>
<tr>
<td>RATS</td>
<td>BE-SEL</td>
<td>42.2</td>
<td>78.9</td>
<td>21.1</td>
<td>19.4</td>
<td>20.2</td>
</tr>
<tr>
<td>Checkmarx</td>
<td>BE-SEL</td>
<td>43.1</td>
<td>41.1</td>
<td>58.9</td>
<td>39.6</td>
<td>47.3</td>
</tr>
<tr>
<td>VulDeePecker</td>
<td>BE-SEL</td>
<td>5.7</td>
<td>7.0</td>
<td>93.0</td>
<td>88.1</td>
<td>90.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System</th>
<th>Dataset</th>
<th>FPR (%)</th>
<th>FNR (%)</th>
<th>TPR (%)</th>
<th>P (%)</th>
<th>F1 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VulDeePecker vs. Code similarity-based vulnerability detection systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VUDDY</td>
<td>BE-SEL-NVD</td>
<td>0</td>
<td>95.1</td>
<td>4.9</td>
<td>100</td>
<td>9.3</td>
</tr>
<tr>
<td>VulPecker</td>
<td>BE-SEL-NVD</td>
<td>1.9</td>
<td>89.8</td>
<td>10.2</td>
<td>84.3</td>
<td>18.2</td>
</tr>
<tr>
<td>VulDeePecker</td>
<td>BE-SEL-NVD</td>
<td>22.9</td>
<td>16.9</td>
<td>83.1</td>
<td>78.6</td>
<td>80.8</td>
</tr>
</tbody>
</table>

<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>Using all function calls</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>

<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>Using some manually-selected function calls</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

<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>Using all function calls</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

Table III. RESULTS FOR ANSWERING RQ1.

Table IV. RESULTS FOR ANSWERING RQ2.
Q & A