MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components

Seunghoon Woo, Hyunji Hong, Eunjin Choi, Heejo Lee

Korea University

USENIX security 2022
Motivation

Modified open-source software reuse is prevalent

- Reuse of open-source software (OSS) becomes a trend in software development
- Unmanaged OSS reuse can pose security threats (e.g., vulnerability propagation)
- Most developers reuse OSS projects with code/structural modifications

ارية Hard to discover propagated vulnerable codes with code changes

How can we precisely discover propagated vulnerable codes with various syntaxes?

* [CCS 2017] “Identifying Open-Source License Violation and 1-day Security Risk at Large Scale”, Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee
* [ICSE 2021] “CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse”, Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh
Motivation

Addressing syntax diversity of vulnerable code

• Syntax diversity of vulnerable code
  ▪ **Internal** modification of OSS
    - OSS source code frequently changes during **OSS updates**
    - Vulnerable code may exist in various syntax depending on the reused OSS version
  ▪ **External** modification of OSS
    - Vulnerable code can be modified during the **OSS reuse process**
Motivation

Example: Syntax diversity caused by the internal OSS modification

- CVE-2014-5461 vulnerability in Lua (DoS vulnerability)
  - This vulnerable code existed in Redis (using Lua)
  - The syntax of the two vulnerable functions is quite different

```c
int luaD_preactcall (lua_State *L, StkId func, int nresults) {
lua_CFunction f;
CallInfo *ci;
int n; /* number of arguments (Lua) or returns (C) */
ptrdiff_t funcr = savestack(L, func);
switch (ttype(func)) {
  ...
  case LUA_TLCL: /* Lua function: prepare its call */
    StkId base;
    Proto *p = clvalue(func)->p;
    luaD_checkstack(L, p->maxstacksize);
    func = restack(L, funcr);
    n = cast_int(L->top - func) - 1;
    luaD_checkstack(L, p->maxstacksize);
  }
}
```

A patch snippet for CVE-2014-5461 in Lua 5.2.3

```c
int luaD_preactcall (lua_State *L, StkId func, int nresults) {
LClosure *cl;
ptdiff_t funcr;
if (!ttisfunction(func)) /* `func` is not a function? */
  func = tryfuncTM(L, func); /* check the `function' tag method */
funcr = savestack(L, func);
cl = &clvalue(func)->l;
L->ci->savedpc = L->savedpc;
if (!cl->isC) /* Lua function? prepare its call */

  CallInfo *ci;
  StkId st, base;
  Proto *p = cl->p;
  luaD_checkstack(L, p->maxstacksize);
  luaD_checkstack(L, p->maxstacksize + p->nuparams);
}
```

A patch snippet for CVE-2014-5461 in Redis (using Lua 5.1)
Motivation

Example: Syntax diversity caused by the internal OSS modification

- CVE-2014-5461 vulnerability in Lua (DoS vulnerability)
Motivation

Example: Syntax diversity caused by the internal OSS modification

- Existing approaches fail to detect this propagated vulnerable code
  - ReDeBug [S&P 2012] ❌

  Considering nearby three (by default) lines of deleted and added code lines from the patch

```c
int luaD_precall (lua_State *L, StkId func, int nresults) {
  lua_CFunction f;
  CallInfo *ci;
  int n; /* number of arguments (Lua) or returns (C) */
  ptrdiff_t funcr = savestack(L, func);
  switch (ttype(func)) {
    ...
    case LUA_TLCL: { /* Lua function: prepare its call */
      StkId base;
      Proto *p = clvalue(func)->p;
      - luaD_checkstack(L, p->maxstacksize);
      - func = restorestack(L, func);
      n = cast_int(L->top - func) - 1;
      + luaD_checkstack(L, p->maxstacksize);
  }
}
```
Motivation

Example: Syntax diversity caused by the internal OSS modification

- Existing approaches fail to detect this propagated vulnerable code
  - VUDDY [S&P 2017] ❌
    - Considering a whole vulnerable function
MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components
Design of MOVERY

MOdified Vulnerable code clone discovERY

• A novel approach to precisely detect modified vulnerable code clones

• Key techniques

  (1) Function collation

  (2) Core line extraction

    ❖ For addressing internal/external modifications of OSS

• Notations
Phase (1) Signature generation

Working example: Heap-buffer overflow vulnerability (CVE-2016-8654) in Jasper

```c
void jpc_qmfb_split_col (...) {
  ...
  if (bufsize > QMFB_SPLITBUFSIZE) {
    if (!(buf = jas_alloc2(bufsize, sizeof(jpc_fix_t)))) {
      abort();
    }
  }
  if (numrows >= 2) {
    // ORIGINAL (WRONG): m = (parity) ? hstartcol : (numrows - hstartcol);
    // ORIGINAL (WRONG): m = (parity) ? hstartrow : (numrows - hstartrow);
    + hstartrow = (numrows + 1 - parity) > 1;
    + // ORIGINAL (WRONG): m = (parity) ? hstartrow : (numrows - hstartrow);
    + m = numrows - hstartrow;
    n = m;
    dstptr = buf;
    srcptr = &a[(1 - parity) * stride]
  }
...```

A patch snippet for CVE-2016-8654 in Jasper

`hstartcol` -> `hstartrow`
Phase (1) Signature generation

Working example

• Function collation

![Oldest vulnerable function](image)

```c
1 void jpc_qmfb_split_col (...) {
2 ...  
3 if (bufsize > QMFB_SPLITBUFSIZE) {
4   if (!(buf = jas_alloc(bufsize * sizeof(jpc_fix_t)))) {
5     abort();
6   }
7  }
8 if (numrows >= 2) {
9   hstartcol = (numrows + 1 - parity) > 1;
10  m = (parity) ? hstartcol : (numrows - hstartcol);
11  
12  n = m;
13  dstptr = buf;
14  srcptr = &a[(1 - parity) * stride]
15  ...
```

![Disclosed vulnerable function](image)

```c
1 void jpc_qmfb_split_col (...) {
2 ...  
3 if (bufsize > QMFB_SPLITBUFSIZE) {
4   if (!(buf = jas_alloc2(bufsize, sizeof(jpc_fix_t)))) {
5     abort();
6   }
7  }
8 if (numrows >= 2) {
9   hstartcol = (numrows + 1 - parity) > 1;
10  m = (parity) ? hstartcol : (numrows - hstartcol);
11  
12  n = m;
13  dstptr = buf;
14  srcptr = &a[(1 - parity) * stride]
15  ...
```

• Highlighted areas indicate the code parts that differ from the disclosed vulnerable function
Phase (1) Signature generation

Definition of core lines

- Core lines in vulnerability signature generation

  1. **Essential code lines**
     - Code lines that were deleted from the patch and exist in both fo and fd

  2. **Dependent code lines**
     - Code lines that have control/data dependencies with the essential code lines

  3. **Control flow code lines**
     - Control statements that exist in both fo and fd
Phase (1) Signature generation

Working example

1) Extracting essential code lines (Ev)

- Code lines that were deleted from the patch (existing in both fo and fd)
- Essential code lines are closely related to the vulnerability manifestation

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!((buf = jas_alloc(bufsize * sizeof(jpc_fix_t))))){
5         abort();
6     }
7  }
8  if (numrows >= 2) {
9      hstartcol = (numrows + 1 - parity) > 1;
10     m = (parity) ? hstartcol : (numrows - hstartcol);
11 ...
```

```
1 void jpc_qmfb_split_col (...) {
2 ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!((buf = jas_alloc2(bufsize, sizeof(jpc_fix_t))))){
5         abort();
6     }
7  }
8  if (numrows >= 2) {
9      // ORIGINAL (WRONG): m = (parity) ? hstartcol : (numrows - hstartcol);
10     m = numrows - hstartcol;
11     n = m;
12     dstptr = buf;
13     srcptr = &a[(1 - parity) * stride]
14 ...
```

Oldest vulnerable function

Disclosed vulnerable function
Phase (1) Signature generation

Working example

2) Extracting dependent code lines (Dv):

- Code lines that have control/data dependency with the essential code lines
- To determine whether the vulnerability trigger environment has propagated

```c
1 void jpc_qmfb_split_col(...) {
2 ...
3 if (bufsize > QMFB_SPLITBUFSIZE) {
4   if (!(buf = jas_alloc(bufsize * sizeof(jpc_fix_t)))) {
5     abort();
6   }
7 }
8  if (numrows >= 2) {
9    hstartcol = (numrows + 1 - parity) > 1;
10   m = (parity) ? hstartcol : (numrows - hstartcol);
11   n = m;
12   dstptr = buf;
13   srcptr = &a[(1 - parity) * stride]
14 ...
```

```
1 void jpc_qmfb_split_col(...) {
2 ...
3 if (bufsize > QMFB_SPLITBUFSIZE) {
4   if (!(buf = jas_alloc2(bufsize, sizeof(jpc_fix_t)))) {
5     abort();
6 }
7 }
8  if (numrows >= 2) {
9    hstartcol = (numrows + 1 - parity) > 1;
10   m = numrows - hstartcol;
11   n = m;
12   dstptr = buf;
13   srcptr = &a[(1 - parity) * stride]
14 ...
```
Phase (1) Signature generation

Working example

3) Extracting control flow code lines (Fv)
   - To determine whether the essential code line has still reachable with the same conditions

```
1 void jpc_qmfb_split_col (...) {
2  ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!((buf = jas_alloc(bufsize * sizeof(jpc_fix_t))))) {  
5         abort();  
6     }
7  }
8  if (numrows >= 2) {
9     hstartcol = (numrows + 1 - parity) > 1;
10    m = (parity) ? hstartcol : (numrows - hstartcol);
11   
12    n = m;
13   dstptr = buf;
14   srcptr = &a[(1 - parity) * stride]
15  ...
```

Oldest vulnerable function

```
1 void jpc_qmfb_split_col (...) {
2  ...
3  if (bufsize > QMFB_SPLITBUFSIZE) {
4     if (!((buf = jas_alloc2(bufsize, sizeof(jpc_fix_t))))) {  
5         abort();  
6     }
7  }
8  if (numrows >= 2) {
9     hstartcol = (numrows + 1 - parity) > 1;
10    m = numrows - hstartcol;
11    n = m;
12    dstptr = buf;
13    srcptr = &a[(1 - parity) * stride]
14  ...
```

Disclosed vulnerable function
Phase (1) Signature generation

Gathering code lines and generating signatures

4) Generating signatures

- Vulnerability signature ($S_v$)
- Patch signature ($S_p$)
  - An approach similar to generating a vulnerability signature is performed (deleted -> added)
  - Control flow lines ($F_p$) that exist only in the patch function are already included in $E_p$

$$S_v = (E_v, D_v, F_v)$$

$$S_p = (E_p, D_p)$$
Phase (2) Vulnerable code clone discovery

Detecting vulnerable code clones in the target program (T)

• A function $f$ in $T$ is a vulnerable code clone if it satisfies:

- **Cond 1** $f$ should contain all code lines in $S_v$.
  \[
  \forall l \in S_v . (l \in f)
  \]

- **Cond 2** $f$ should not contain any code lines in $S_p$.
  \[
  \forall l \in S_p . (l \notin f)
  \]

- **Cond 3** The syntax of $f$ should be similar to $f_o$ or $f_d$.
  \[
  (\text{Sim}(f, f_o) \geq \theta) \lor (\text{Sim}(f, f_d) \geq \theta)
  \]

Phase (2) Vulnerable code clone discovery

Detecting vulnerable code clones in the target program (T)

- A function \( f \) in \( T \) is a vulnerable code clone if it satisfies:

  \begin{itemize}
  \item \textbf{Cond 1)} \( f \) should contain all code lines in \( S_v \).
    \[ \forall l \in S_v \cdot (l \in f) \]
  \item \textbf{Cond 2)} \( f \) should not contain any code lines in \( S_p \).
    \[ \forall l \in S_p \cdot (l \notin f) \]
  \item \textbf{Cond 3)} The syntax of \( f \) should be similar to \( f_o \) or \( f_d \).
    \[ (\text{Sim}(f, f_o) \geq \theta) \lor (\text{Sim}(f, f_d) \geq \theta) \]
  \end{itemize}

* Using the Jaccard index by considering the function as a set of code lines
EVALUATION

Dataset and parameter setting

- CVE dataset
  - 4,219 C/C++ CVE vulnerabilities (patches)
    - Collected from NVD
    - 7,762 vulnerable/patched function pairs
    - 5,936 oldest vulnerable functions
- Target programs
  - 10 software programs that are popular (based on GitHub stars) and contain a sufficient number of OSS components
- Parameter
  - $\theta = 0.5$

Target program overview

<table>
<thead>
<tr>
<th>IDX</th>
<th>Name</th>
<th>Version</th>
<th>#Line*</th>
<th>#Comp†</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>FreeBSD</td>
<td>v12.2.0</td>
<td>14,489,534</td>
<td>47</td>
<td>Operating system</td>
</tr>
<tr>
<td>T2</td>
<td>ReactOS</td>
<td>v0.4.13</td>
<td>6,419,855</td>
<td>23</td>
<td>Operating system</td>
</tr>
<tr>
<td>T3</td>
<td>ArangoDB</td>
<td>v3.7.9</td>
<td>3,064,973</td>
<td>22</td>
<td>Database</td>
</tr>
<tr>
<td>T4</td>
<td>FFmpeg</td>
<td>n4.3.2</td>
<td>1,230,520</td>
<td>4</td>
<td>Multimedia processing</td>
</tr>
<tr>
<td>T5</td>
<td>OpenCV</td>
<td>v4.5.1</td>
<td>1,092,317</td>
<td>15</td>
<td>Computer vision</td>
</tr>
<tr>
<td>T6</td>
<td>Emscripten</td>
<td>v2.0.15</td>
<td>759,020</td>
<td>11</td>
<td>Compiler</td>
</tr>
<tr>
<td>T7</td>
<td>Crown</td>
<td>v0.42.0</td>
<td>723,372</td>
<td>20</td>
<td>Game engine</td>
</tr>
<tr>
<td>T8</td>
<td>Git</td>
<td>v2.31.0</td>
<td>293,467</td>
<td>5</td>
<td>Version control system</td>
</tr>
<tr>
<td>T9</td>
<td>OpenMVG</td>
<td>v1.6</td>
<td>262,610</td>
<td>8</td>
<td>Image processing</td>
</tr>
<tr>
<td>T10</td>
<td>Redis</td>
<td>v5.0.12</td>
<td>212,672</td>
<td>8</td>
<td>Database</td>
</tr>
</tbody>
</table>

*: Counting only C/C++ code lines, †: The number of modified OSS components.
EVALUATION

Accuracy measurement

• Comparison targets
  - MOVERY significantly outperformed existing approaches

<table>
<thead>
<tr>
<th>Target software</th>
<th>#Discovered VCCs*</th>
<th>ReDeBug</th>
<th>VUDDY</th>
<th>MOVERY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#Tp</td>
<td>#FP</td>
<td>#FN</td>
</tr>
<tr>
<td>ReactOS</td>
<td>210</td>
<td>31</td>
<td>9</td>
<td>179</td>
</tr>
<tr>
<td>OpenCV</td>
<td>72</td>
<td>38</td>
<td>15</td>
<td>34</td>
</tr>
<tr>
<td>Emscripten</td>
<td>56</td>
<td>22</td>
<td>8</td>
<td>34</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>33</td>
<td>25</td>
<td>44</td>
<td>8</td>
</tr>
<tr>
<td>Crown</td>
<td>23</td>
<td>22</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>OpenMVG</td>
<td>23</td>
<td>15</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>ArangoDB</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>FFmpeg</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Redis</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Git</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>434</td>
<td>163</td>
<td>87</td>
<td>271</td>
</tr>
</tbody>
</table>

*VCCs: Vulnerable Code Clones
EVALUATION

Accuracy measurement

- Comparison targets
  - Two existing vulnerable code clone detection tools: **VUDDY [S&P 2017]** and **ReDeBug [S&P 2012]**
  - **MOVERY** significantly outperformed existing approaches

<table>
<thead>
<tr>
<th>Target software</th>
<th>#Discovered VCCs*</th>
<th>ReDeBug</th>
<th>VUDDY</th>
<th>MOVERY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Tp</td>
<td>#Fp</td>
<td>#Fn</td>
<td>Precision</td>
</tr>
<tr>
<td>ReactOS</td>
<td>210</td>
<td>31</td>
<td>9</td>
<td>179</td>
</tr>
<tr>
<td>OpenCV</td>
<td>72</td>
<td>38</td>
<td>15</td>
<td>34</td>
</tr>
<tr>
<td>Emscripten</td>
<td>56</td>
<td>22</td>
<td>8</td>
<td>34</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>33</td>
<td>25</td>
<td>44</td>
<td>8</td>
</tr>
<tr>
<td>Crown</td>
<td>23</td>
<td>22</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>OpenMVG</td>
<td>23</td>
<td>15</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>ArangoDB</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>FFmpeg</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Redis</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Git</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>434</strong></td>
<td><strong>163</strong></td>
<td><strong>87</strong></td>
<td><strong>271</strong></td>
</tr>
</tbody>
</table>

*VCCs: Vulnerable Code Clones

MOVERY could discover 2.5x and 5.8x more vulnerable codes than ReDeBug and VUDDY
EVALUATION

Accuracy measurement

- MOVERY could discover more VCCs than VUDDY and ReDeBug

VCCs that are hardly discovered by existing techniques

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
<th>#Discovered VCCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>VCCs without code lines deleted in security patches.</td>
<td>32</td>
</tr>
<tr>
<td>T2</td>
<td>VCCs with various syntaxes derived from the oldest vulnerable function (fo).</td>
<td>221 (221 VCCs closer to fo than fd)</td>
</tr>
<tr>
<td>T3</td>
<td>VCCs with heavy syntax change.</td>
<td>166 (166 VCCS: $\text{Sim}(f, f_d) \leq 0.5)$</td>
</tr>
</tbody>
</table>
Conclusion

- Many vulnerable codes are propagated with syntax modifications
  - 396 (91%) out of 434 VCCs existed in a different syntax to the disclosed vulnerable function
- MOVERY
  - A precise approach for discovering modified VCCs from modified components
  - MOVERY significantly outperformed existing approaches in vulnerable code clone discovery
    - High vulnerability discovery accuracy: 96% precision and 96% recall
- Equipped with VCC discovery results from MOVERY,
  - Developers can address threats caused by propagated vulnerabilities in modified components
Thank you for your attention!

• MOVERY repository (https://github.com/wooseunghoon/MOVERY-public)

CONTACT

• Seunghoon Woo (seunghoonwoo@korea.ac.kr, https://wooseunghoon.github.io)
• Computer & Communication Security Lab (https://ccs.korea.ac.kr)
• Center for Software Security and Assurance (https://cssa.korea.ac.kr)
APPENDIX I

Preprocessing

• Abstraction
  ▪ Replacing every occurrence of parameters, variable names, variable types, and callee function names in each function with symbols PARAM, DNAME, DTYPE, and FCALL

```c
3  if (bufsize > QMFB_SPLITBUFSIZE) {
8  if (numrows >= 2) {
9      hstartcol = (numrows + 1 - parity) > 1;
14     srcptr = &a[(1 - parity) * stride]
}
```

• Selective abstraction
  ▪ Abstraction is applied only when the abstraction code before and after the patch is different

```c
3  if (DVAL > QMFB_SPLITBUFSIZE) {
8  if (PARAM >= 2) {
9      DVAL = (PARAM + 1 - PARAM) > 1;
14     DVAL = &PARAM[(1 - PARAM) * PARAM]
```
MOVEREY requires the least amount of time in the vulnerability discovery.

MOVEREY discovers VCCs from the target programs varied from 213 K to 14.5 M LoC. The required time is not significantly increased.

Speed and scalability measurement

(a) Target preprocessing times.  (b) Matching times.  (c) Total times (preprocessing + matching).