CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse

43rd International Conference on Software Engineering

Seunghoon Woo*, Sunghan Park*, Seulbae Kim†, Heejo Lee*, Hakjoo Oh*

*Korea University, †Georgia Institute of Technology

ICSE 2021
GOAL

• Identifying Open-source software (OSS) components in the target software

• Motivation
  • Open-source software is reused extensively in software development
  • Reusing OSS without proper management
    😞 Vulnerability propagation
    😞 License violation
    😞 Supply chain attack
CHALLENGES

• Previous approaches cannot precisely identify OSS components
  • Modified OSS reuse
    • The cause of false negatives in component identification
  • Nested OSS components
    • The cause of false positives in component identification
**CHALLENGES**

- **Modified OSS reuse**
  - Modified reuse patterns
  - Partial reuse, *structure-changed* reuse, *code-changed* reuse

![Diagram showing partial reuse between Zlib and Linux kernel]

```c
/* inflate.c -- zlib decompression
 * Copyright (C) 1995-2005 Mark Adler
 * For conditions of distribution and use, see copyright notice in zlib.h
 * Based on zlib 1.2.3 but modified for the Linux Kernel by
```
CHALLENGES

• Modified OSS reuse
  • Modified reuse patterns
    • Partial reuse, structure-changed reuse, code-changed reuse

Simple threshold-based approach
Many false negatives
CHALLENGES

• Nested components
CHALLENGES

• Nested components

Correct answers
• PHP reuses PCRE
• MongoDB reuses PCRE

Wrong answers
• MongoDB reuses PHP
• PHP reuses MongoDB
CHALLENGES

- Nested components

Existing software composition analysis approaches

Many false positives
**Centris**

- **CENTRIfuge for Software**
  - The first approach to precisely and scalably identify *modified* OSS components
  - Key techniques

  **S1. Redundancy elimination**
  - For *high scalability*

  **S2. Code segmentation**
  - For *high accuracy*
S1. Redundancy elimination

Version 1

\[\text{function } i\]
\[\text{function } k\]
...

Version 2

\[\text{function } j\]
\[\text{function } k\]
...

Version 3

\[\text{function } l\]
\[\text{function } k\]
...

Version update in an OSS
S1. Redundancy elimination

![Diagram showing redundancy elimination in OSS]

A naively generated OSS signature

function k: compared 3+ times
S1. Redundancy elimination

A naively generated OSS signature

A redundancy eliminated signature for an OSS
S1. Redundancy elimination

A naively generated OSS signature

A redundancy eliminated signature for an OSS
S2. Code segmentation

The unique part of the software
- Non-reused code parts
- Self-developed code

Non-unique part of the software
- Reused code parts
  - Cause of false alarms
S2. Code segmentation

Code segmentation
S2. Code segmentation

- How to segment an OSS?
S2. Code segmentation

- Detecting functions belonging to the **borrowed code part of S**
S2. Code segmentation

• Detecting functions belonging to the **borrowed code part** of $S$

\[ G = \{ f \mid (f \in (S \cap L)) \land (birth(f, L) \leq birth(f, S)) \} \]
S2. Code segmentation

1) Measure similarity between S and L

\[ \phi(S, L) = \frac{|G|}{|L|} \]
S2. Code segmentation

1) Measure similarity between S and L
\[
\phi(S, L) = \frac{|G|}{|L|}
\]

2) Check whether G is included in the borrowed code part of S

If \( \phi \geq \theta \) then:

\[ S \cap L \]
\[ S \cap X \]
S2. Code segmentation

1) Measure similarity between S and L

\[ \phi(S, L) = \frac{|G|}{|L|} \]

2) Check whether G is included in the borrowed code part of S

If \( \phi \geq \theta \) then:

3) Remove G from S

\[ S = (S \setminus G) \]
S2. Code segmentation

1) Measure similarity between S and L
\[
\phi(S, L) = \frac{|G|}{|L|}
\]

2) Check whether G is included in the borrowed code part of S

If \( \phi \geq \theta \) then:

3) Remove G from S
\[
S = (S \setminus G)
\]

Repeat this process for all OSS in the component DB

=> Only the application code of S remains
Component identification in the target software

• Comparing T with the application code part of the collected OSS

\[
\Phi(T, S) = \frac{|T \cap S_A|}{|S_A|}
\]

=> if \( \Phi(T, S) \geq \theta \), then S is the component of T
EVALUATION

• Dataset
  • Popular C/C++ OSS projects from GitHub (April, 2020)
    • #Stars $\geq$ 100
    • A total of 10,241 projects, 229,326 versions, and 80 billion lines of code (LoC)

• Parameter
  • $\theta = 0.1$
**EVALUATION**

1) **Accuracy**
- Cross-comparison experiments (10,241 vs 10,241)
- **91% precision and 94% recall**
  - Modified components account for 95% of the detected components!
EVALUATION

1) Accuracy
- Cross-comparison experiments (10,241 vs 10,241)
- **91% precision and 94% recall**
  - Modified components account for 95% of the detected components!

2) Scalability

![Graph showing time in hours for different dataset sizes (1M, 10M, 100M, 1B, 5B) with different methods: SourcererCC (square), CENTRIS (first exp., triangular), CENTRIS (nth exp., circular). The x-axis represents the dataset size in lines of code (LoC), and the y-axis represents time in hours. The graph highlights the limitation of SourcererCC due to memory error and shows significant time savings in the later experiments.]
**EVALUATION**

1) **Accuracy**
   - Cross-comparison experiments (10,241 vs 10,241)
   - 91% precision and 94% recall
     - Modified components account for 95% of the detected components!

2) **Scalability**

3) **Identification speed**
   - Takes ≤ 1 min to identify components in the 1 M LoC target software
1) Accuracy
- Cross-comparison experiments (10,241 vs 10,241)
- 91% precision and 94% recall
  • Modified components account for 95% of the detected components!

2) Scalability

3) Identification speed
- Takes ≤ 1 min to identify components in the 1 M LoC target software

4) vs. DejaVu (OOPSLA 2017)
- Code-duplication detection tool
- Using four target software programs
- DejaVu showed only 10% precision

<table>
<thead>
<tr>
<th>DejaVu</th>
<th>CENTRIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision 10%</td>
<td>95%</td>
</tr>
<tr>
<td>Recall 40%</td>
<td>100%</td>
</tr>
</tbody>
</table>
CONCLUSION

• 95% of detected components were reused with modification
  • Modified components, not likely to be identified, have more chances to pose security threats
  • Management for supply chains considering modified components is required

• CENTRIS can be the first step towards addressing problems arising from unmanaged OSS components in practice
  • With the information provided by CENTRIS, developers can mitigate security threats
    • e.g., they can update old-and-vulnerable components
Q&A

Thank you for your attention!

- CENTRIS repository (https://github.com/wooseunghoon/Centris-public)
- CENTRIS at IoTcube (https://iotcube.net/Centris)

CONTACT

- Seunghoon Woo (seunghoonwoo@korea.ac.kr, https://wooseunghoon.github.io)
- Center for Software Security and Assurance (https://cssa.korea.ac.kr)