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

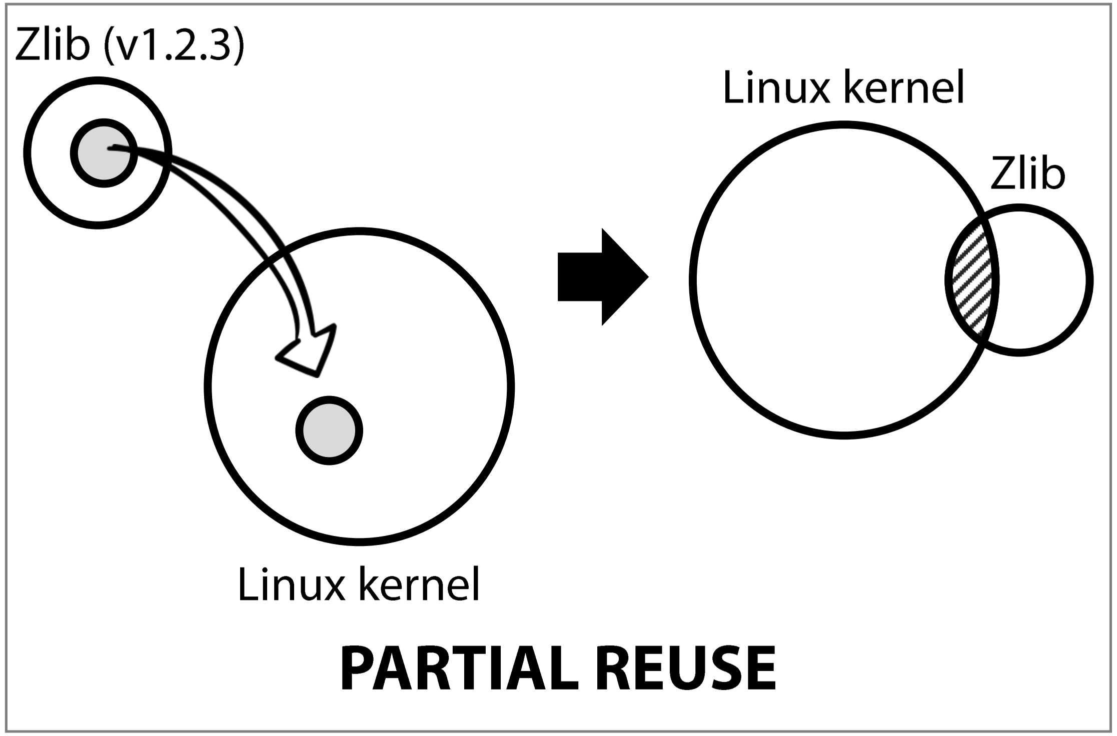
    🙁 Supply chain attack

CHALLENGES

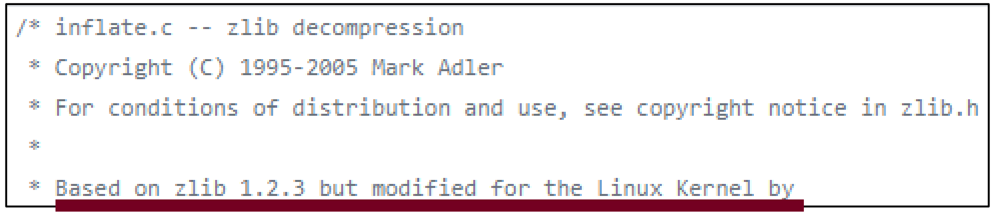
- **Previous approaches cannot precisely identify OSS components**
  - **Modified OSS reuse**
    - The cause of false negatives in component identification
  - **Nested OSS components**
    - The cause of false positives in component identification

CHALLENGES

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



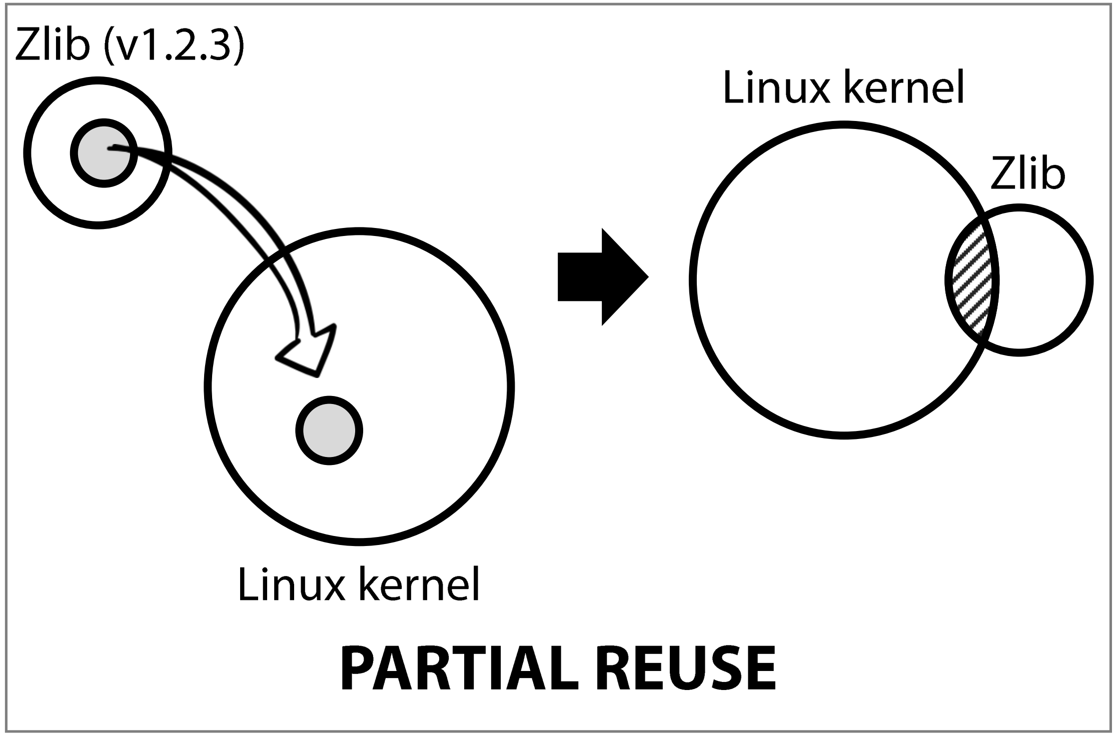
PARTIAL REUSE



```
/* 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



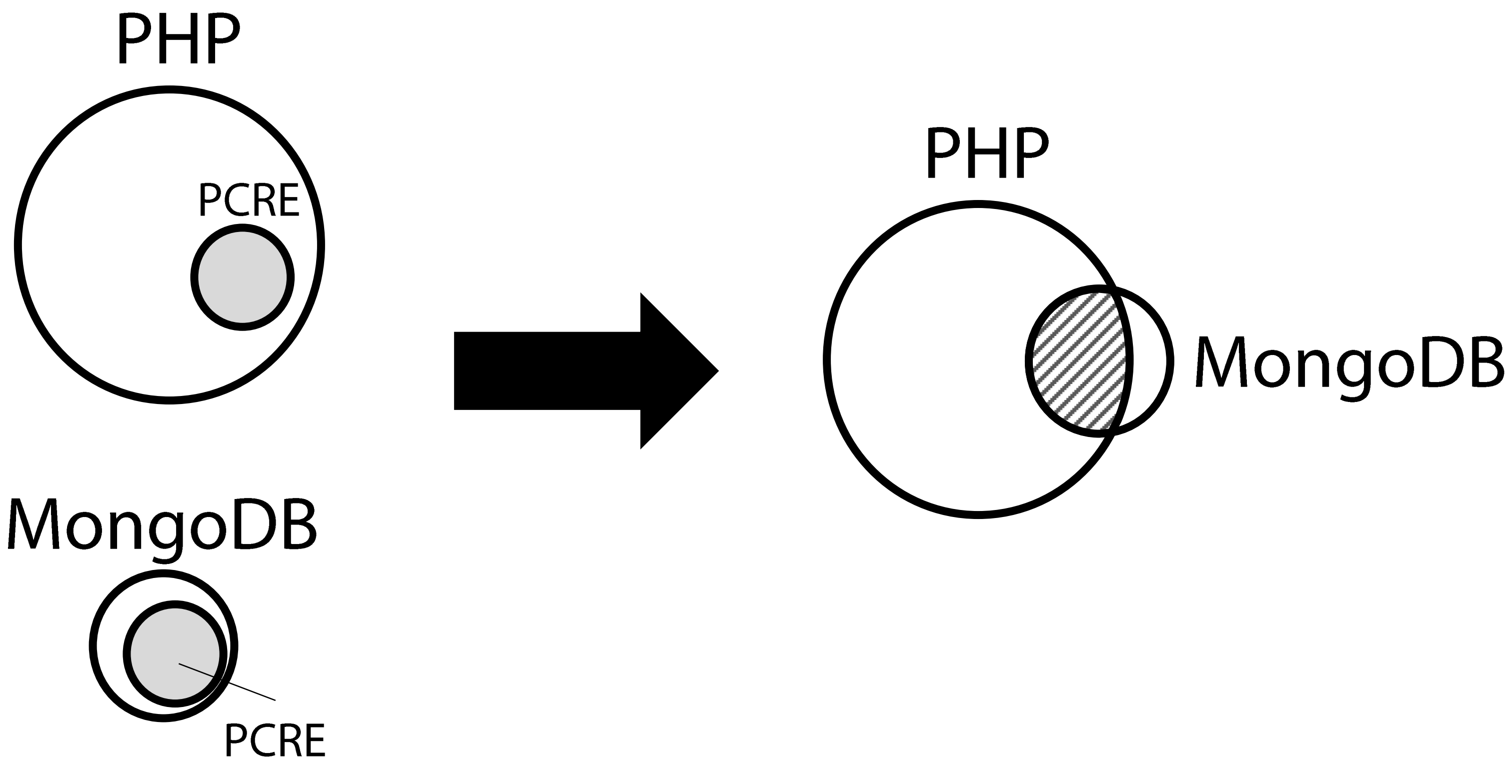
PARTIAL REUSE

Simple threshold-based approach

Many **false negatives**

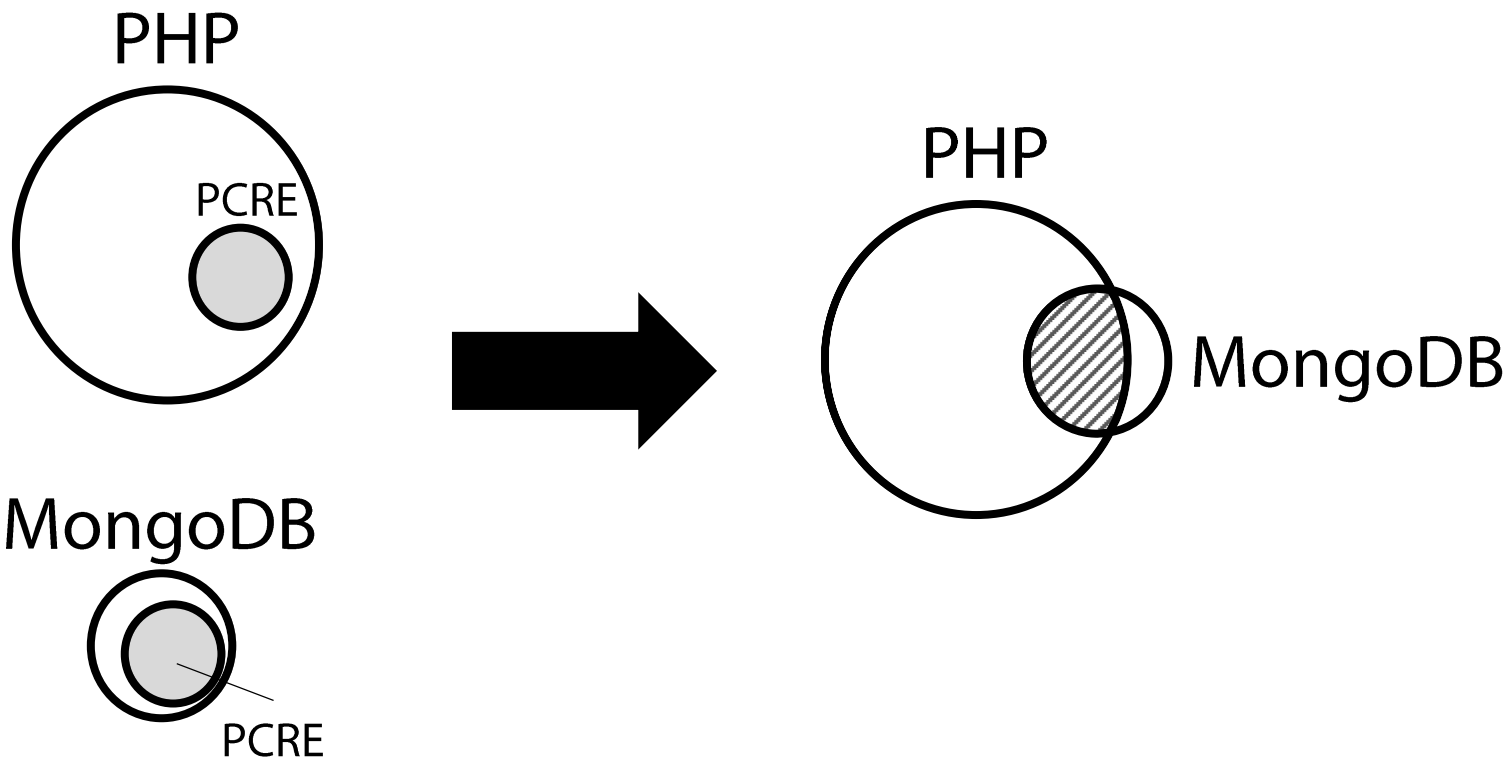
Challenges

- **Nested components**



CHALLENGES

- **Nested components**



Correct answers

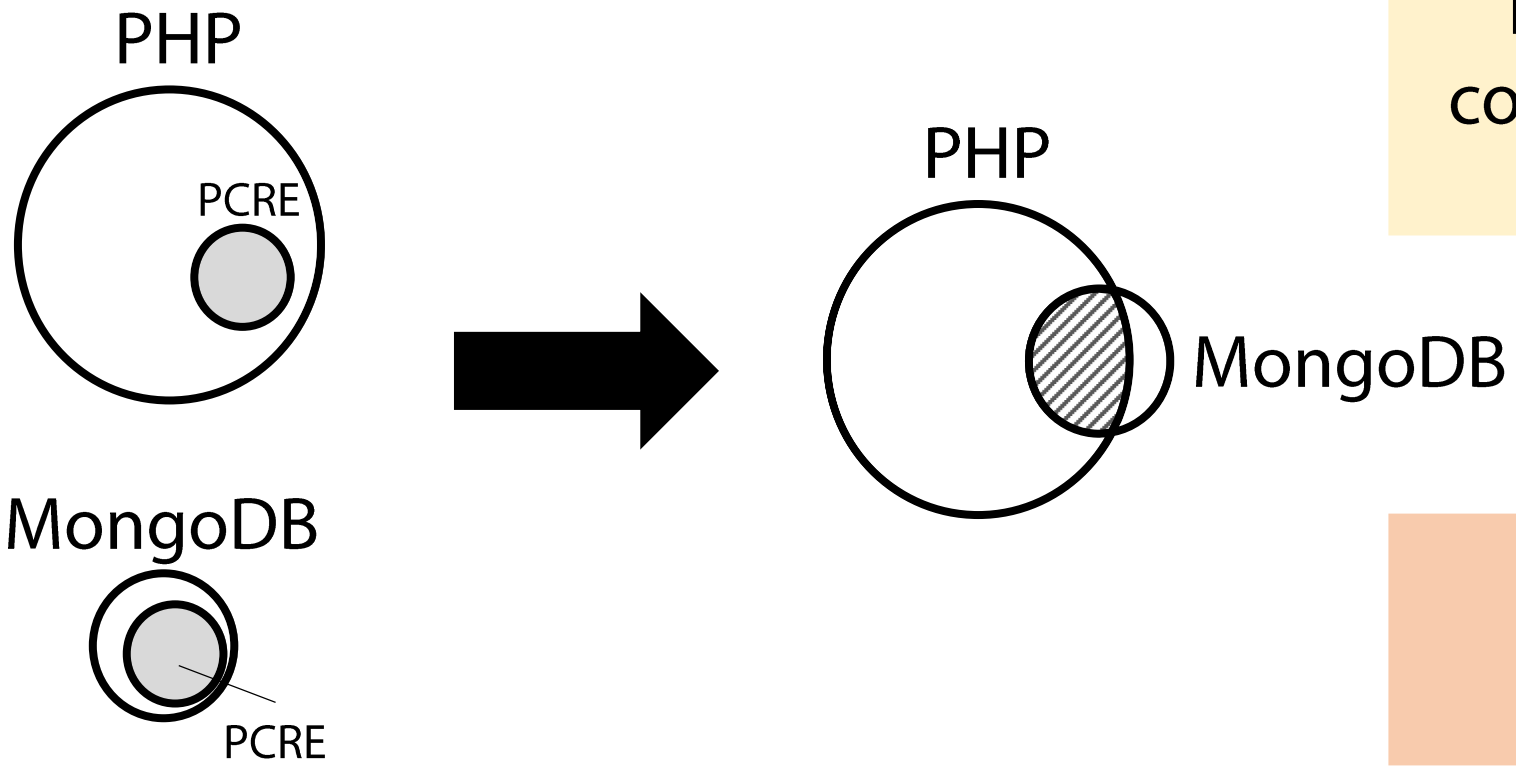
- PHP reuses PCRE
- MongoDB reuses PCRE

Wrong answers

- MongoDB reuses PHP
- PHP reuses MongoDB

CHALLENGES

- **Nested components**



Existing software composition analysis approaches

Many
false positives

CENTRIS

- **CENTRI**fuge for **S**oftware
  - The first approach to precisely and scalably identify **modified** OSS components
  - Key techniques

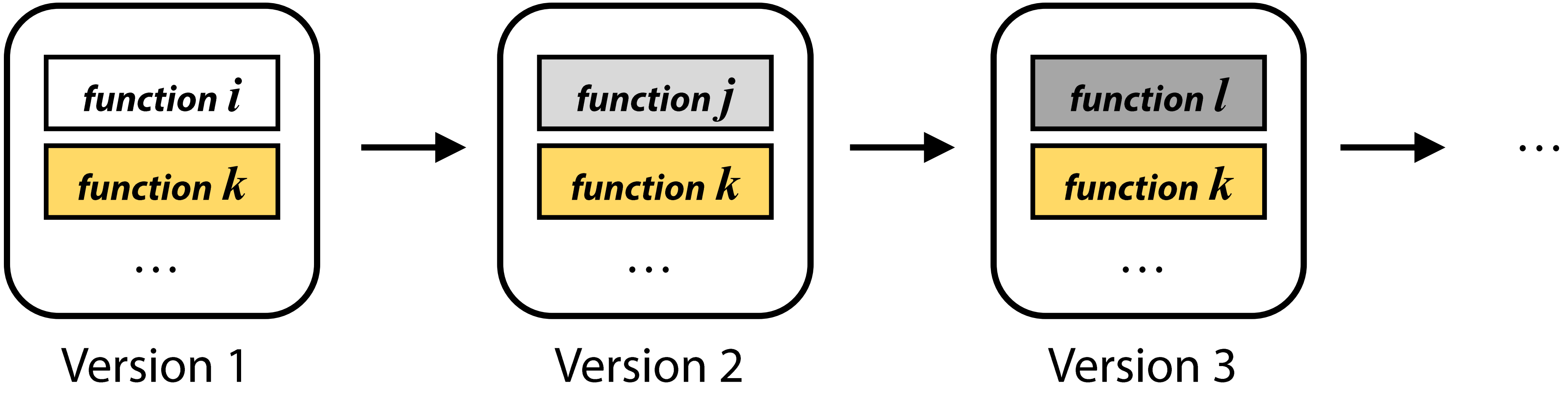
    S1. Redundancy elimination

    - For *high scalability*

    S2. Code segmentation

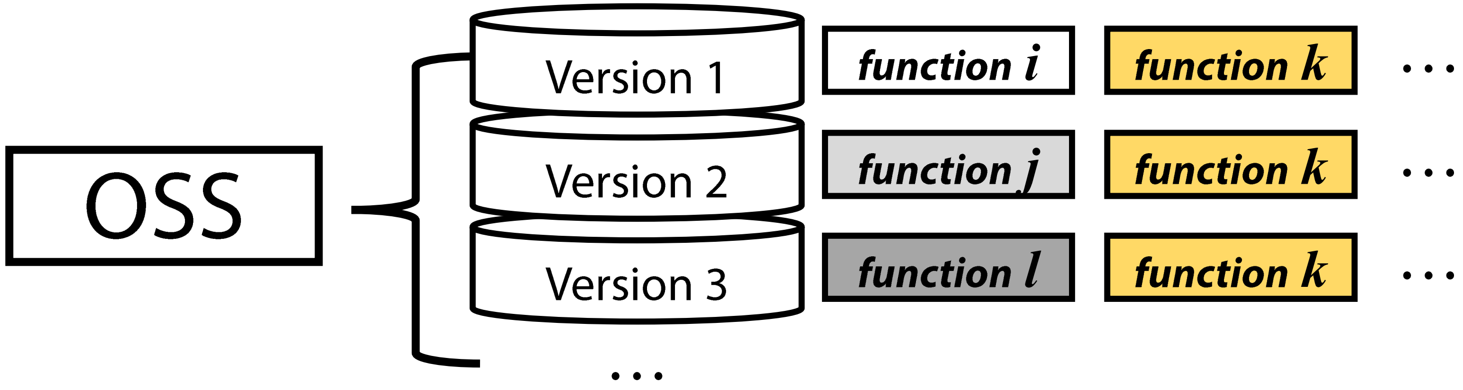
    - For *high accuracy*

S1. Redundancy elimination



Version update in an OSS

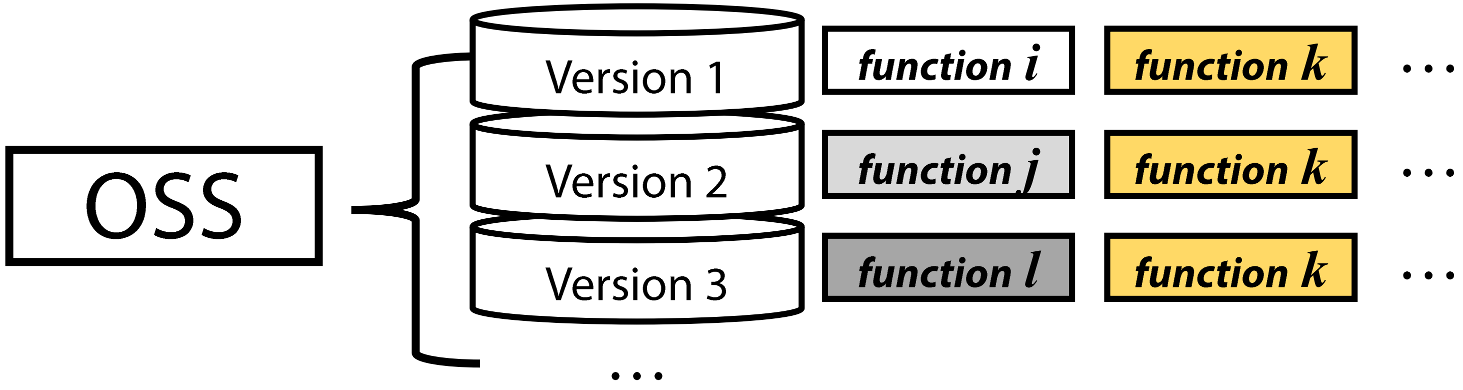
S1. Redundancy elimination



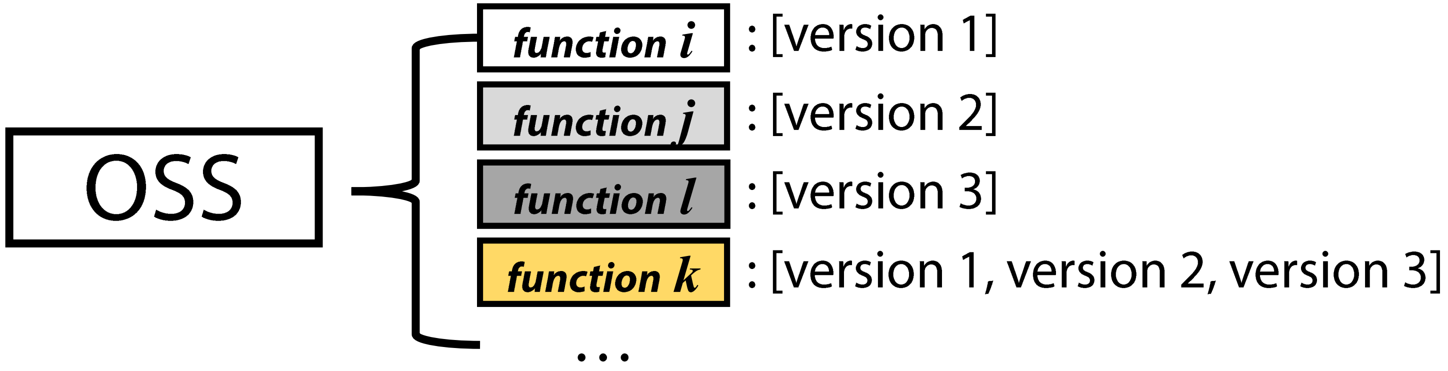
A naively generated OSS signature

function *k* : compared 3+ times

S1. Redundancy elimination

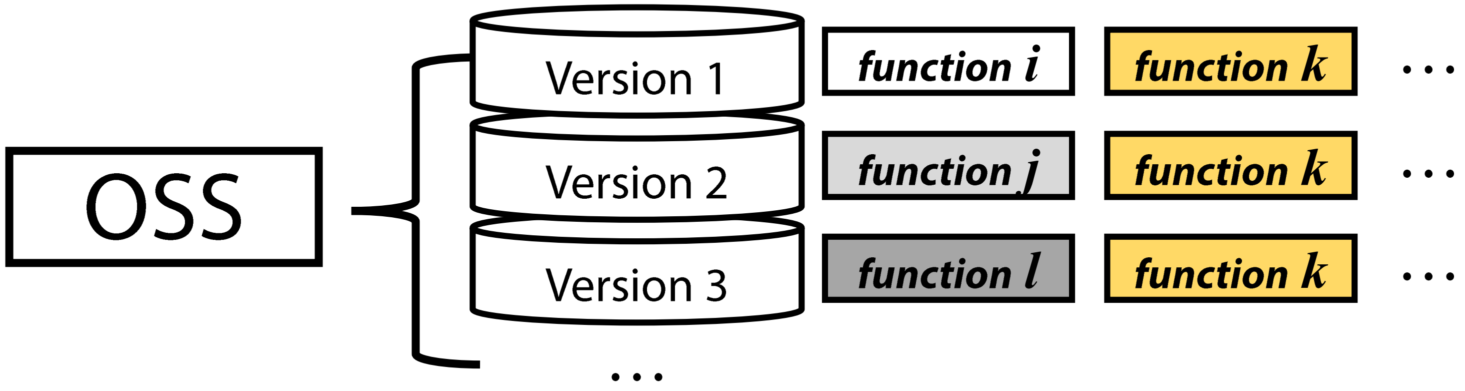


A naively generated OSS signature



A redundancy eliminated signature for an OSS

S1. Redundancy elimination



OSS

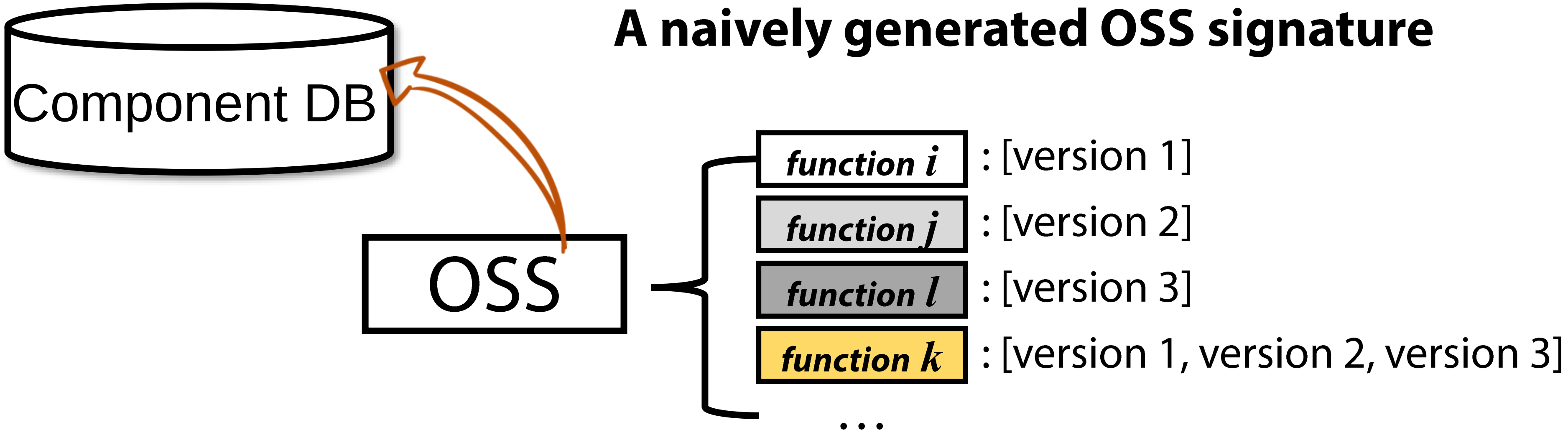
Version 1 — function *i* — function *k* …

Version 2 — function *j* — function *k* …

Version 3 — function *l* — function *k* …

…

A naively generated OSS signature



Component DB

OSS

function *i* : [version 1]

function *j* : [version 2]

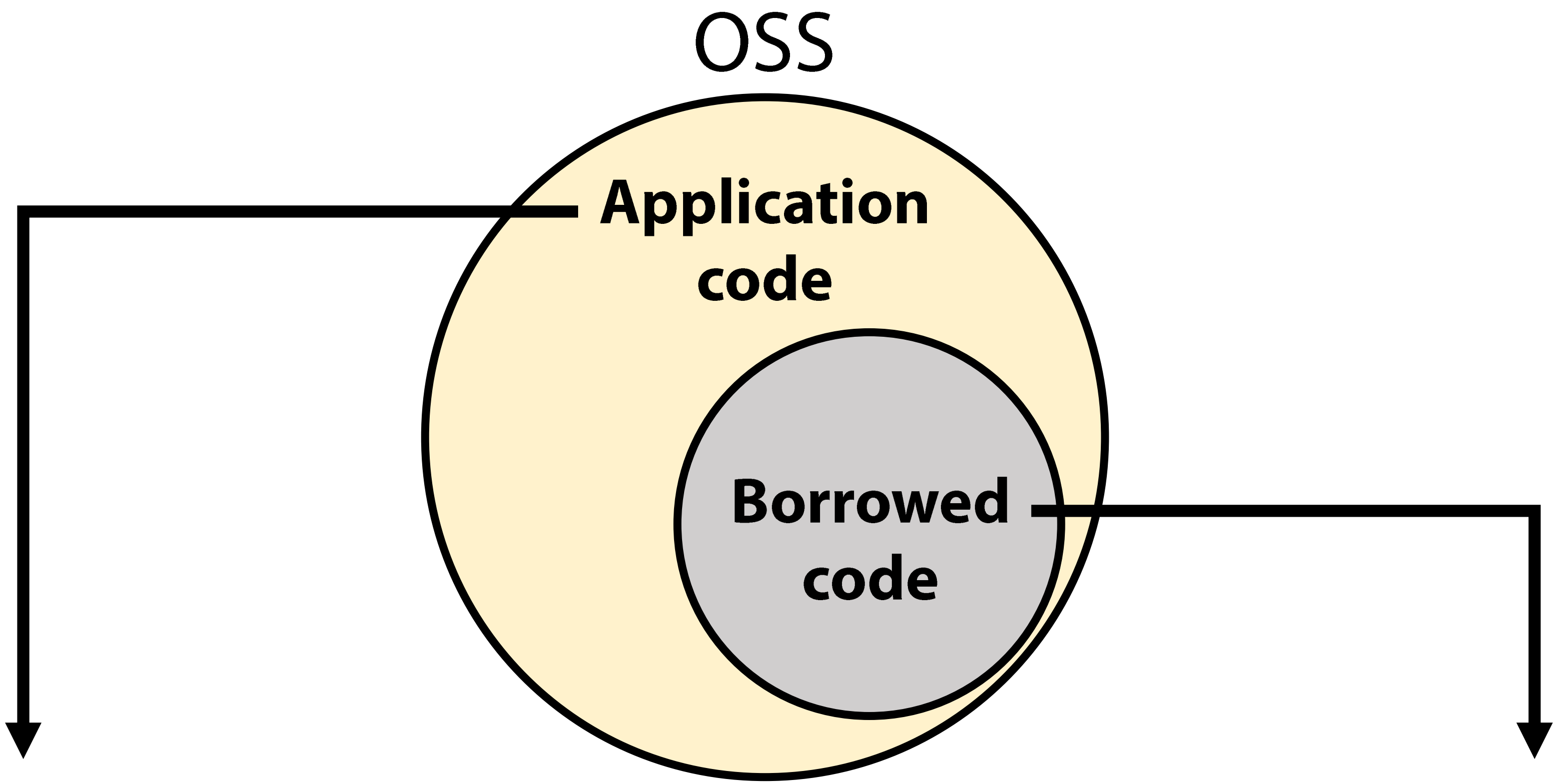
function *l* : [version 3]

function *k* : [version 1, version 2, version 3]

…

A redundancy eliminated signature for an OSS

S2. Code segmentation



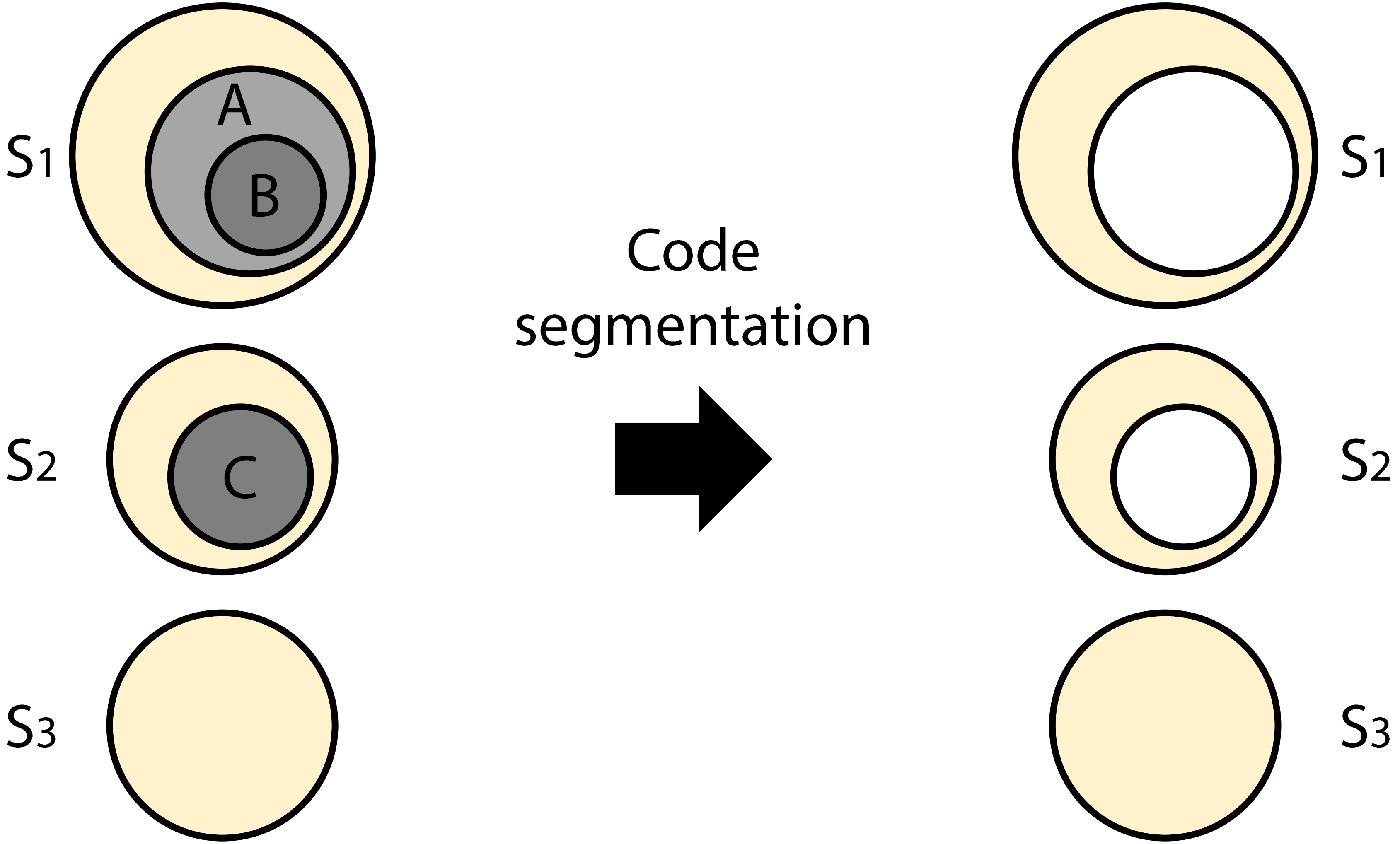
The unique part of the software

- Non-reused code parts
- Self-developed code

Non-unique part of the software

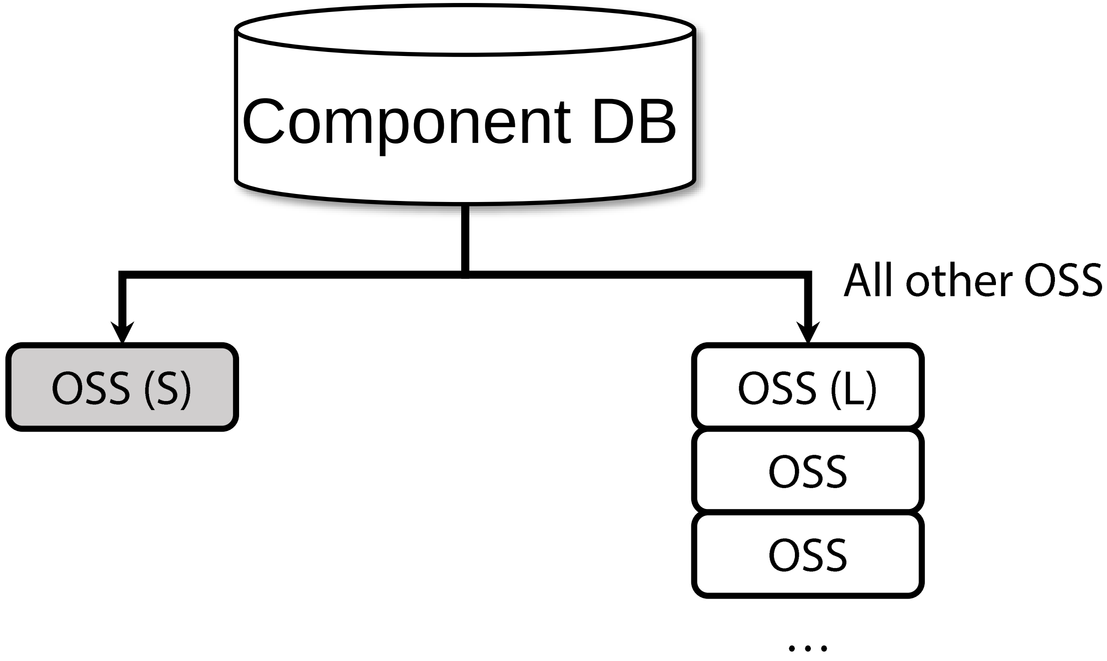
- Reused code parts
- **Cause of false alarms**

S2. Code segmentation



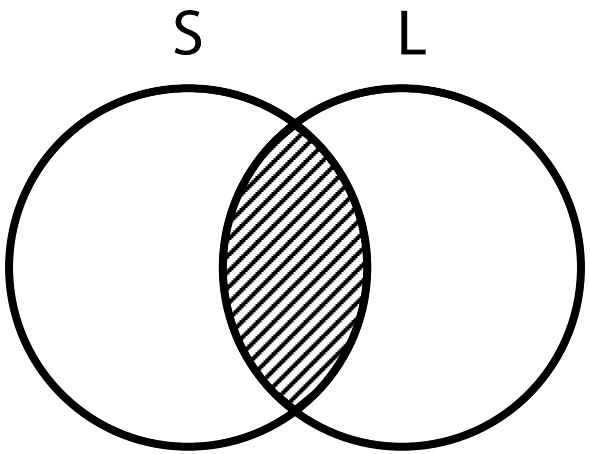
S2. Code segmentation

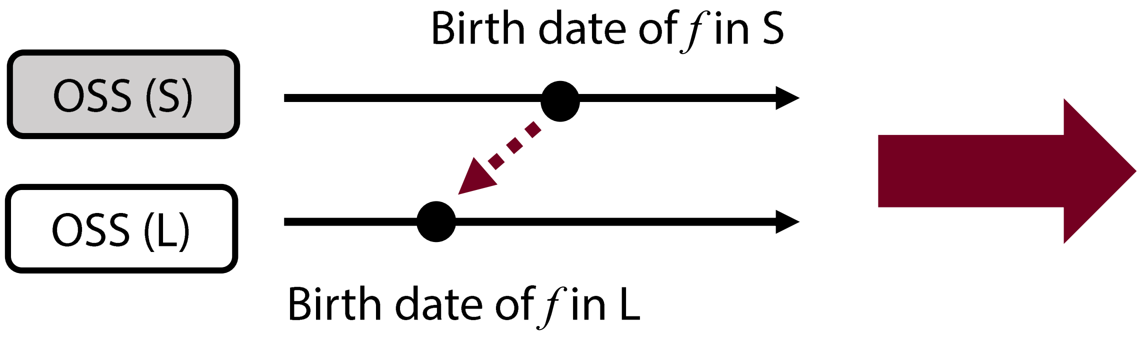
- **How to segment an OSS?**



S2. Code segmentation

- **Detecting functions belonging to the <u>borrowed code part</u> of S**

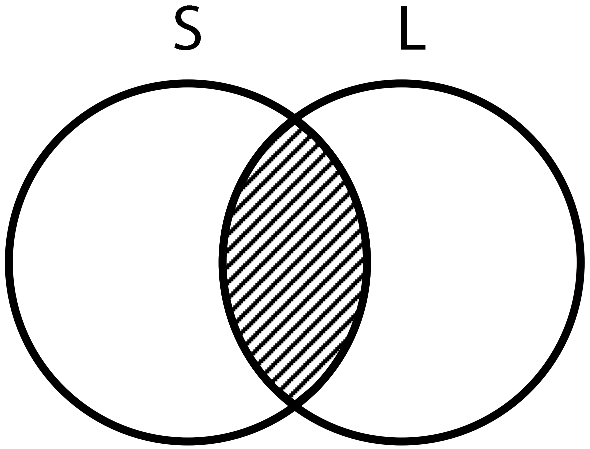




$f \subset$ Borrowed code part of S

S2. Code segmentation

- **Detecting functions belonging to the <u>borrowed code part</u> of S**



$$G = \{f \mid \big(f \in (S \cap L)\big) \wedge (birth(f, L) \leq birth\ (f, S))\}$$

S2. Code segmentation

1) **Measure similarity between S and L**

$$\phi(\mathrm{S}, \mathrm{L}) = \frac{|G|}{|L|}$$

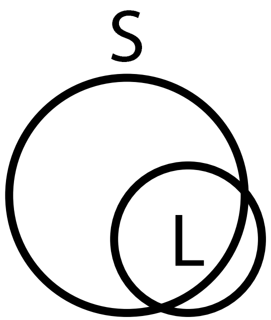
S2. Code segmentation

1) Measure similarity between S and L

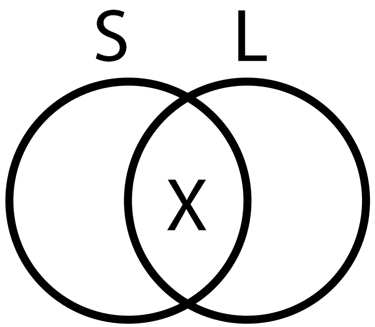
$$\phi(\mathrm{S}, \mathrm{L}) = \frac{|G|}{|L|}$$

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

If $\phi \geq \theta$ then :



or



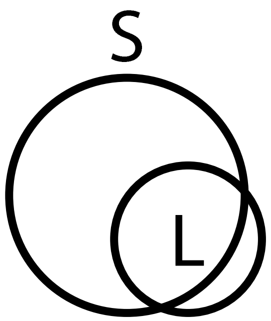
S2. Code segmentation

1) **Measure similarity between S and L**

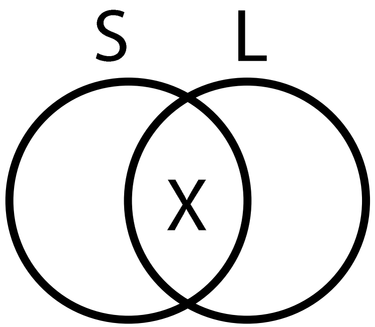
$$\phi(\mathrm{S},\mathrm{L}) = \frac{|G|}{|L|}$$

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

If $\phi \geq \theta$ then :



or



3) **Remove G from S**

S = (S \ G)

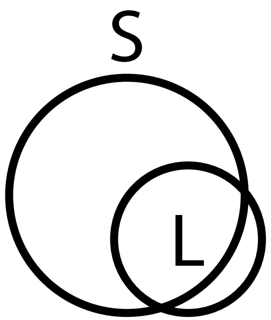
S2. Code segmentation

1) **Measure similarity between S and L**

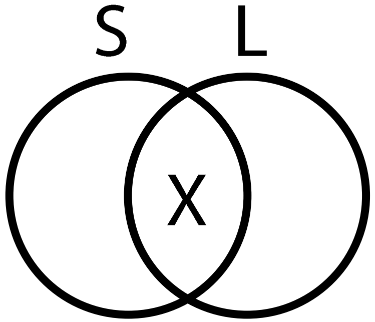
$$\phi(\mathrm{S},\mathrm{L}) = \frac{|G|}{|L|}$$

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

If $\phi \geq \theta$ then :



or



3) **Remove G from S**

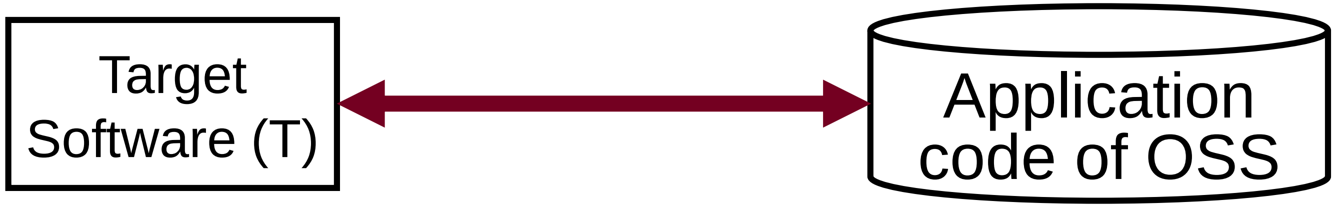
S = (S \ G)

Repeat this process for all OSS in the component DB

=> Only <u>the application code of S</u> remains

Component identification in the target software

- **Comparing T with the application code part of the collected OSS**



$$\Phi(\mathrm{T}, S) = \frac{|T \cap S_A|}{|S_A|}$$

=> if $\boldsymbol{\Phi}(\mathbf{T}, \mathbf{S}) \geq \boldsymbol{\theta}$, then S is the component of T

Evaluation

- **Dataset**
  - **Popular C/C++ OSS projects from GitHub (April, 2020)**
    - #Stars >= 100
    - A total of 10,241 projects, 229,326 versions, and 80 billion lines of code (LoC)

- **Parameter**
  - $\theta = 0.1$

EVALUATION

1) Accuracy

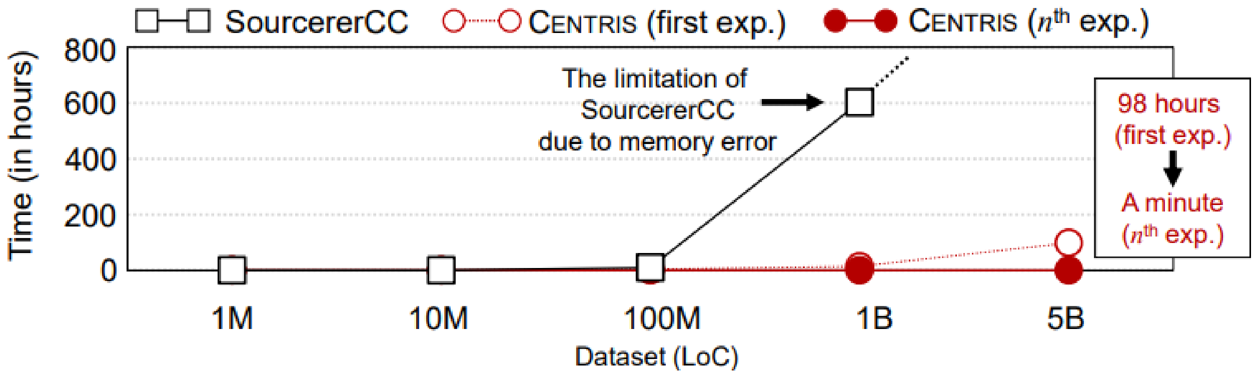
- Cross-comparison experiments (10,241 vs 10,241)
- **91% precision and 94% recall**
  - Modified components account for 95% of the detected components!

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

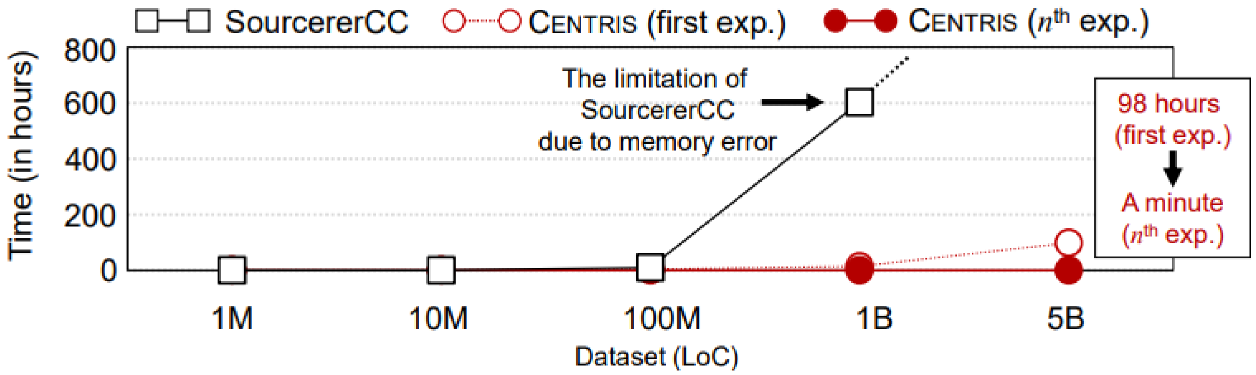


EVALUATION

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



The limitation of SourcererCC due to memory error

98 hours (first exp.)

A minute (n^{th} exp.)

3) Identification speed

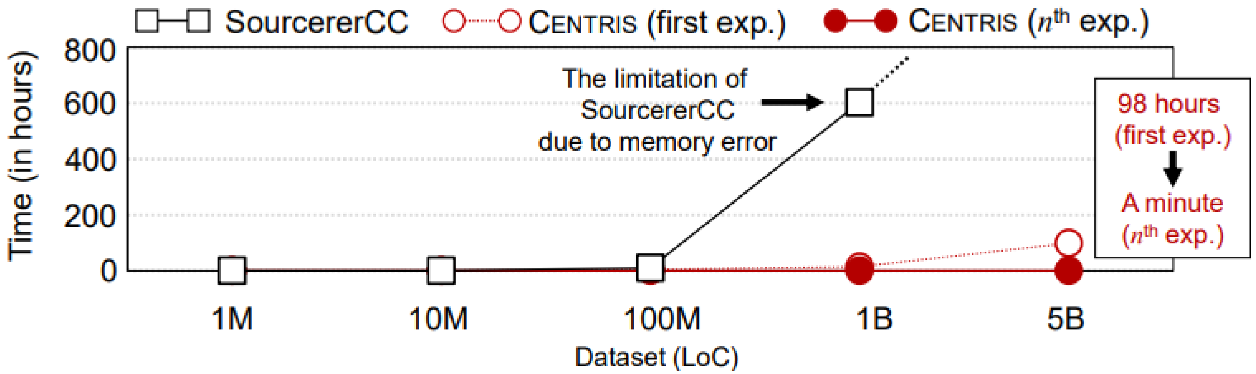
- Takes ≤ **1 min** to identify components in the 1 M LoC target software

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



The limitation of SourcererCC due to memory error

98 hours (first exp.) → A minute (nth exp.)

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

	DejaVu	CENTRIS
Precision	10%	95%
Recall	40%	100%

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)
- Computer & Communication Security Lab (https://ccs.korea.ac.kr)
- Center for Software Security and Assurance (https://cssa.korea.ac.kr)