

GANADI:

Uncovering C/C++ OSS Reuse Genealogies via Pivotal Function-Based Clustering to Enhance Supply Chain Security

Dongyeon Kim, Seunghoon Woo*, Heejo Lee*

Korea University

ASE 2026



**KOREA
UNIVERSITY**



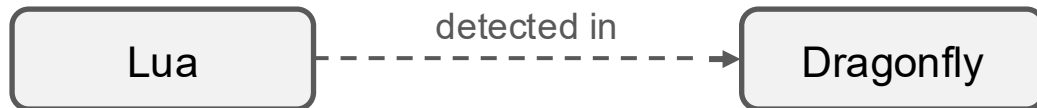
CCSLAB
Computer & Communication
Security Laboratory

Problem

Hidden reuse paths need to be uncovered

- C/C++ OSS is often copied, vendored, or forked
- Reused code can be modified and propagated downstream
- Existing SCA tells **what** is reused, but not **how** it propagated

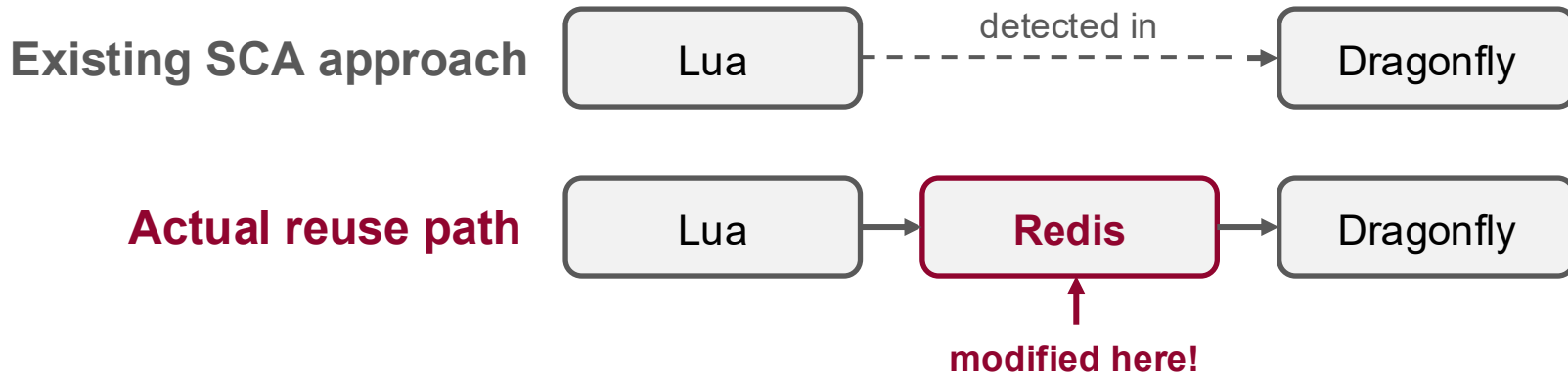
Existing SCA approach



Problem

Hidden reuse paths need to be uncovered

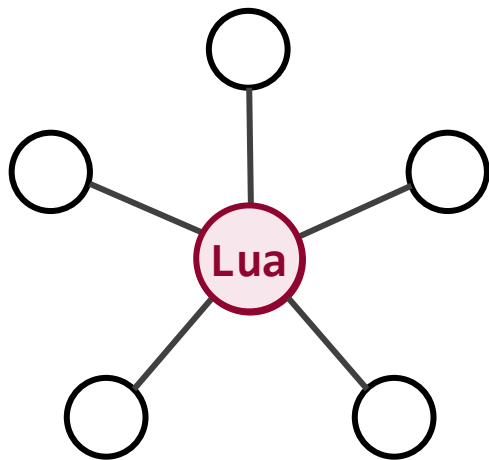
- C/C++ OSS is often copied, vendored, or forked
- Reused code can be modified and propagated downstream
- Existing SCA tells **what** is reused, but not **how** it propagated



Motivating Example

Lua reused by many paths – but through which path?

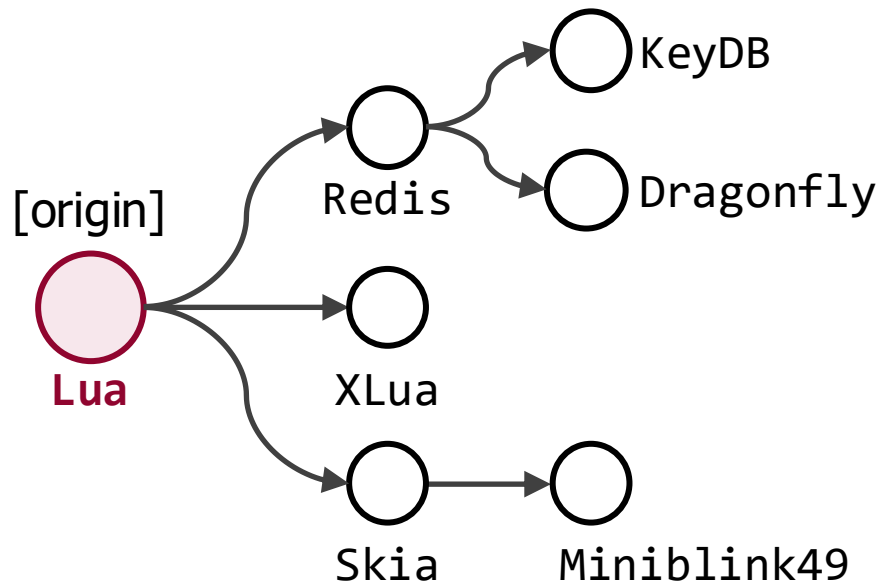
What SCA see



How?



What actually happend



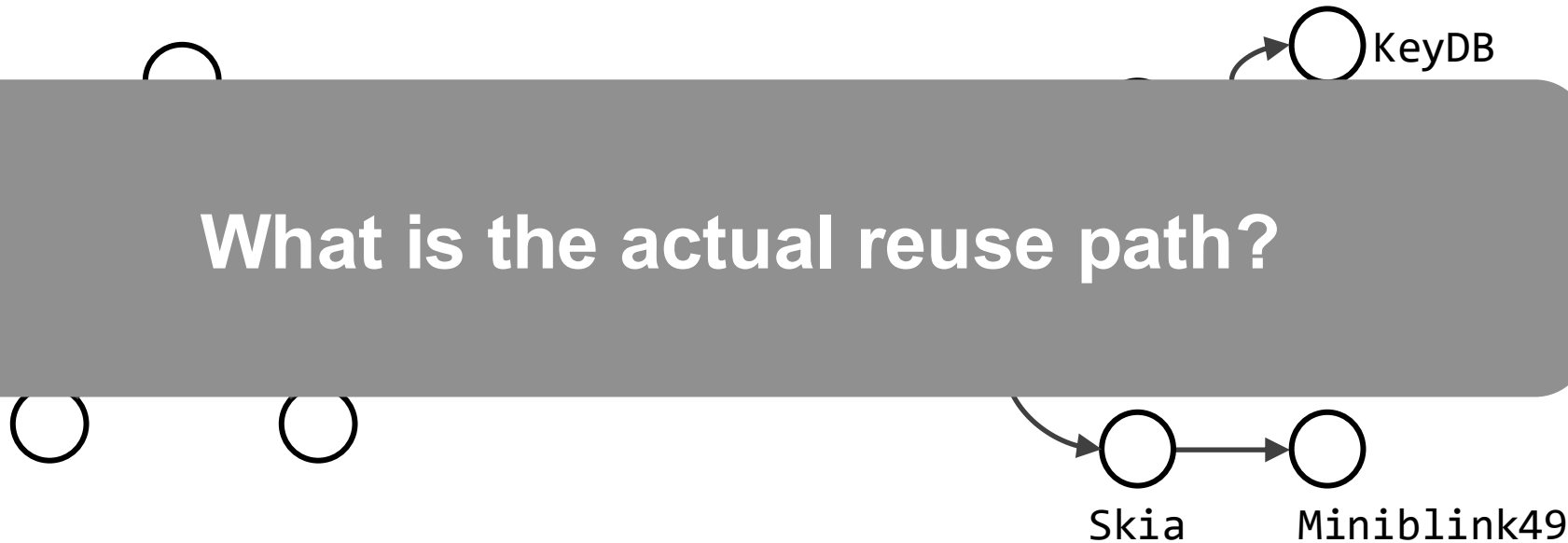
Motivating Example

Lua reused by many paths – but through which path?

What SCA see

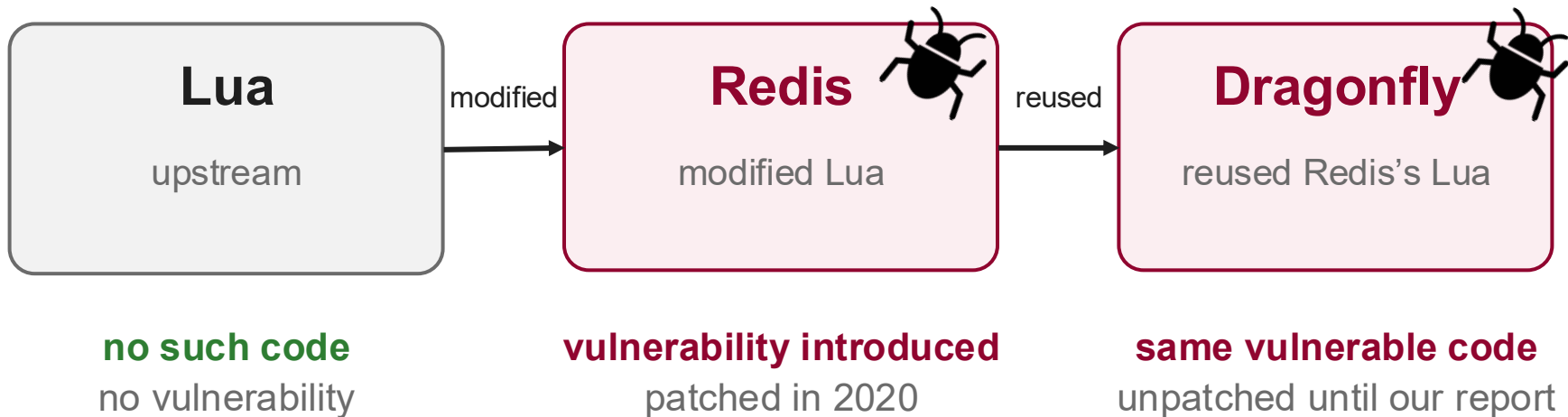
What actually happend

What is the actual reuse path?



Motivating Example

CVE-2020-14147: Lua → Redis → Dragonfly



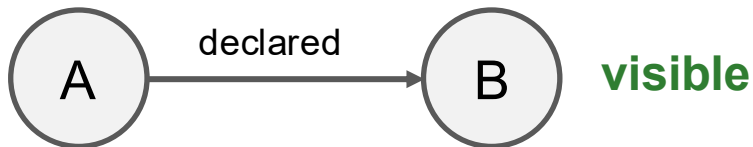
The bug lives on the path, not in the origin

Challenge

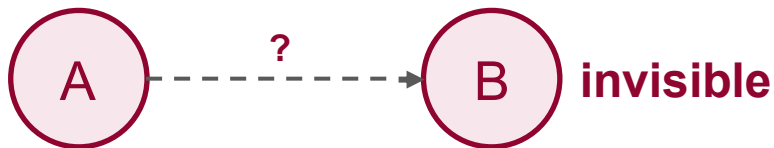
Why is the propagation path hard to reconstruct?

① Lack of explicit linkage

Package manager (e.g., Maven, pip)



C/C++ copy-paste



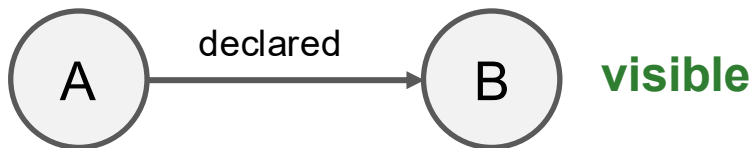
copied files leave no dependency trace

Challenge

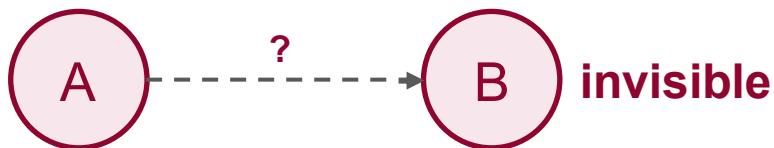
Why is the propagation path hard to reconstruct?

① Lack of explicit linkage

Package manager (e.g., Maven, pip)

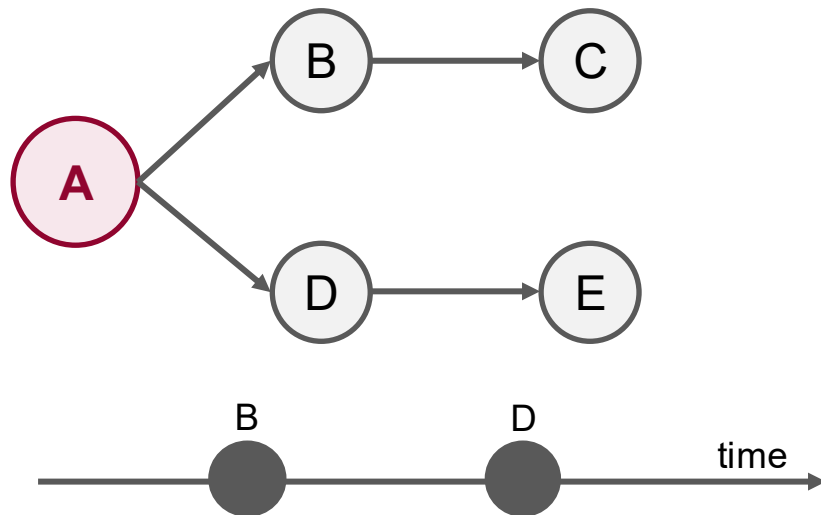


C/C++ copy-paste



copied files leave no dependency trace

② Timestamps $\neq$ reuse direction



“B before D” $\neq$ “B $\rightarrow$ D”

GANADI

Genealogy **AN**alysis for **D**ependency **I**nference

: first approach to C/C++ OSS reuse genealogy

Research Goal

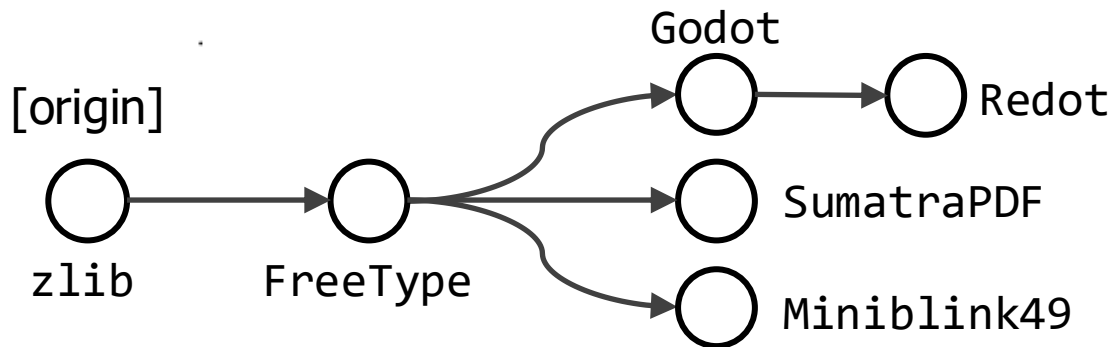
Construct the OSS reuse genealogy graph of an origin

- **Reuse genealogy:** how code flows from an origin to its descendants

Research Goal

Construct the OSS reuse genealogy graph of an origin

- **Reuse genealogy**: how code flows from an origin to its descendants
- **Output**: OSS reuse genealogy graph (node = project, edge = direct)
 - **zlib** → **FreeType** = “FreeType directly reuses zlib”



SBOM completeness · Vulnerability propagation · Prioritized remediation

Key Idea

Pivotal function-based tracing

Track pivotal functions to reconstruct reuse paths

Pivotal function: every function in a file that shares an identical function with the origin

= the code that came from the origin, even if it was modified later

Design of GANADI

Three phases, three questions

Input: one origin OSS (e.g., zlib)

P1

Software clustering

Which projects have
the origin's code?

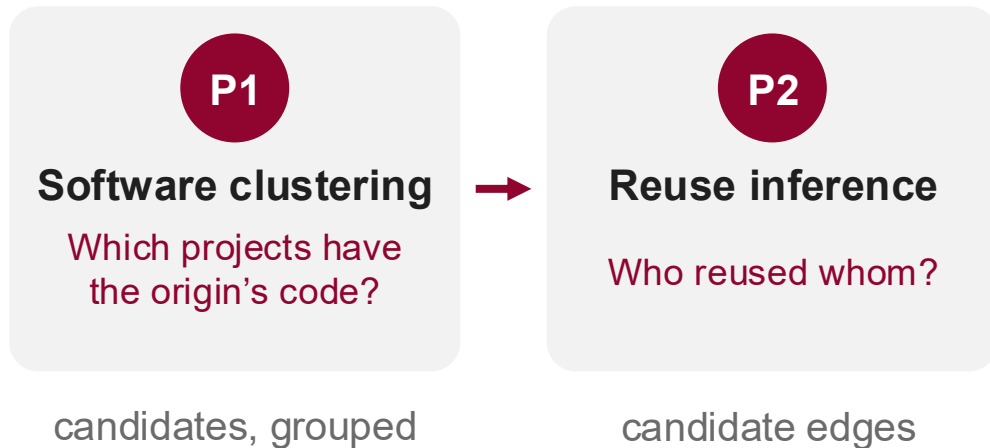
candidates, grouped

Output: the reuse genealogy graph of the origin

Design of GANADI

Three phases, three questions

Input: one origin OSS (e.g., zlib)

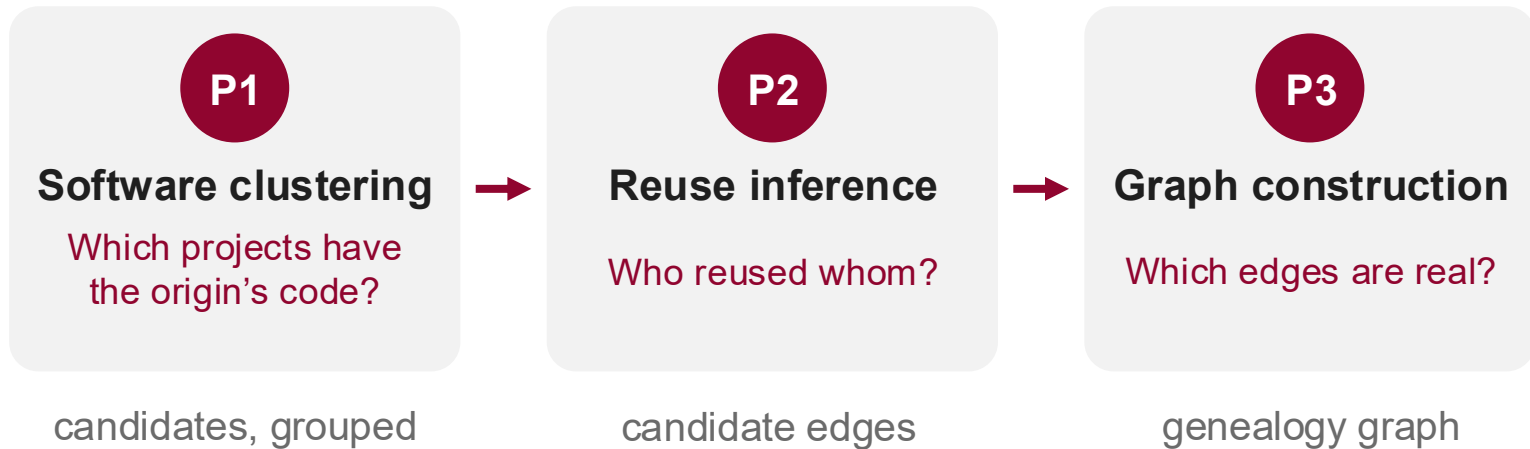


Output: the reuse genealogy graph of the origin

Design of GANADI

Three phases, three questions

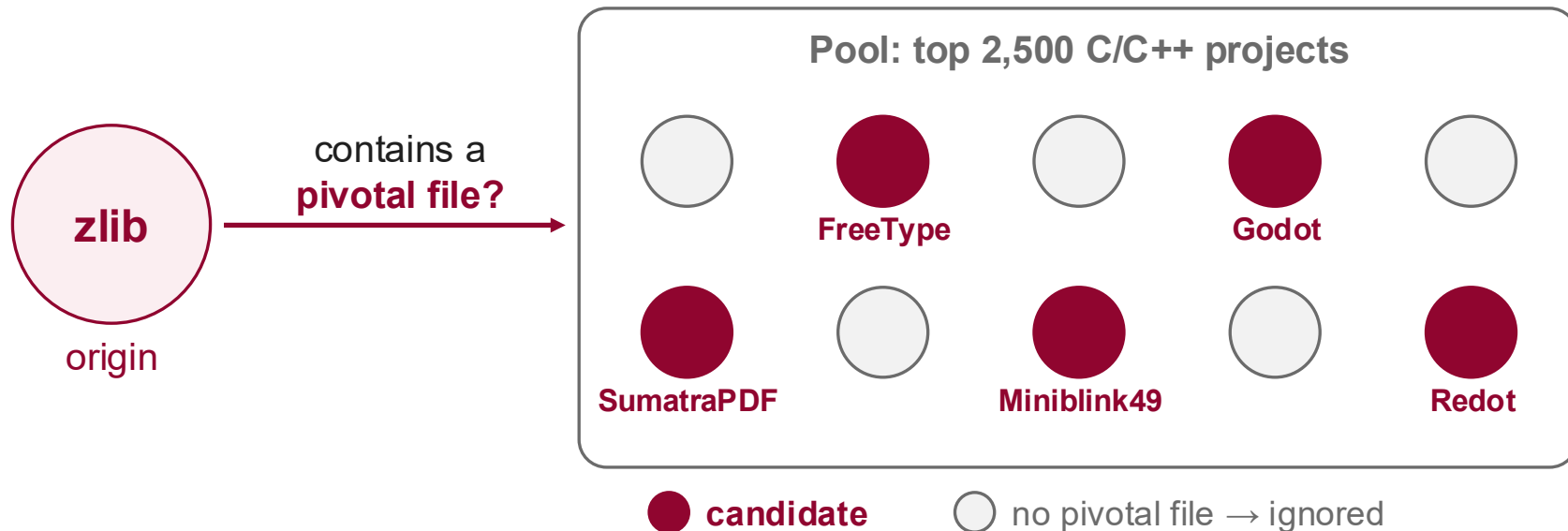
Input: one origin OSS (e.g., zlib)



Output: the reuse genealogy graph of the origin

P1. Software Clustering

Which projects have zlib code?



Candidates: projects that contain a pivotal file

P1. Software Clustering

Multi-criteria clustering — ① Fork relationship

① **Fork** ② File path ③ File names ④ Unique functions

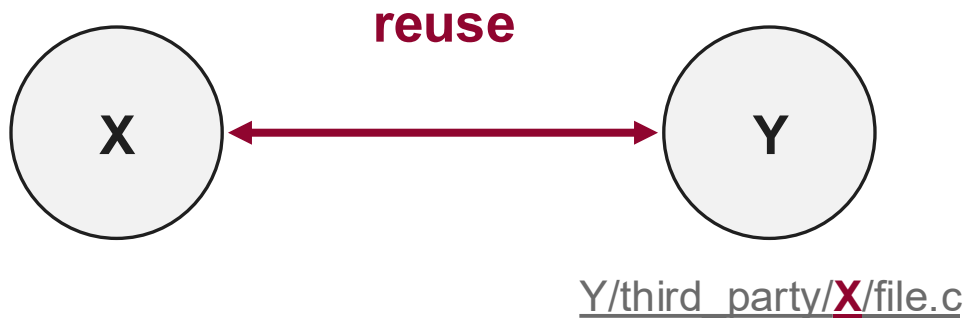


Y is a fork of X (GitHub fork metadata)

P1. Software Clustering

Multi-criteria clustering — ② File path

- ① Fork ② **File path** ③ File names ④ Unique functions

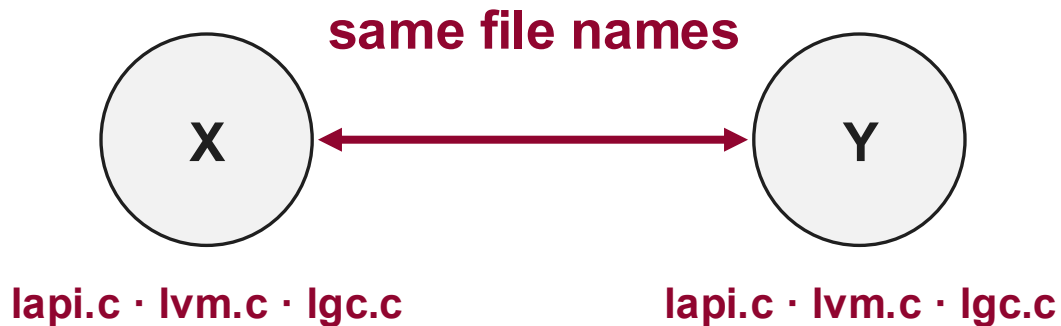


X's name appears in Y's file path

P1. Software Clustering

Multi-criteria clustering — ③ File names

① Fork ② File path ③ **File names** ④ Unique functions

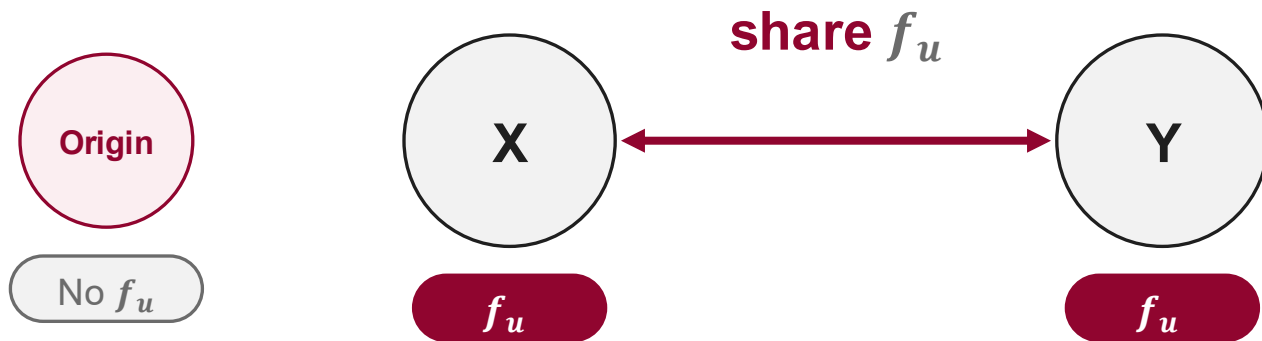


X and Y have the same pivotal file names

P1. Software Clustering

Multi-criteria clustering — ④ Unique functions

- ① Fork ② File path ③ File names ④ **Unique functions**



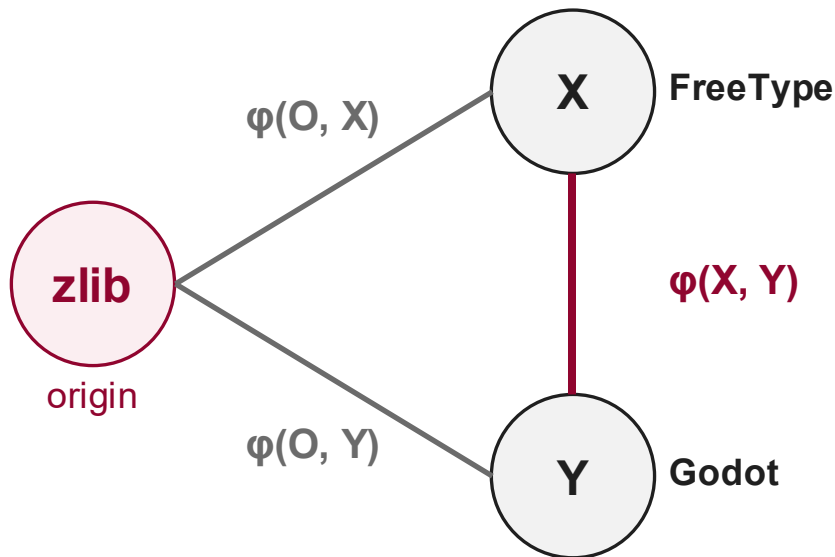
X and Y share a function not in the origin
→ the same modification was passed on

Any one criterion → same cluster

P2. Reuse Inference

Step 1. Code similarity — how much do FreeType and Godot share?

φ = **cosine similarity** of two projects' pivotal-function vectors (0 – 1)



$\varphi(X, Y)$

- Do X and Y share code?

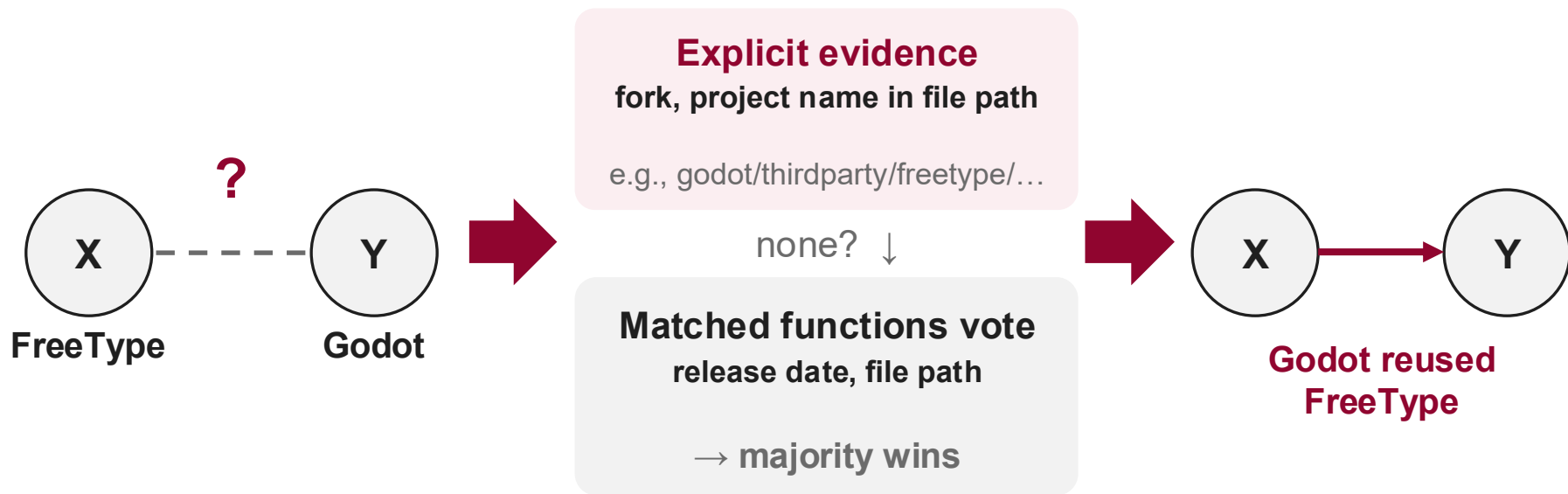
$\varphi(O, X), \varphi(O, Y)$

- Does it come from zlib?

Related pair → **candidate edge** (direction still unknown)

P2. Reuse Inference

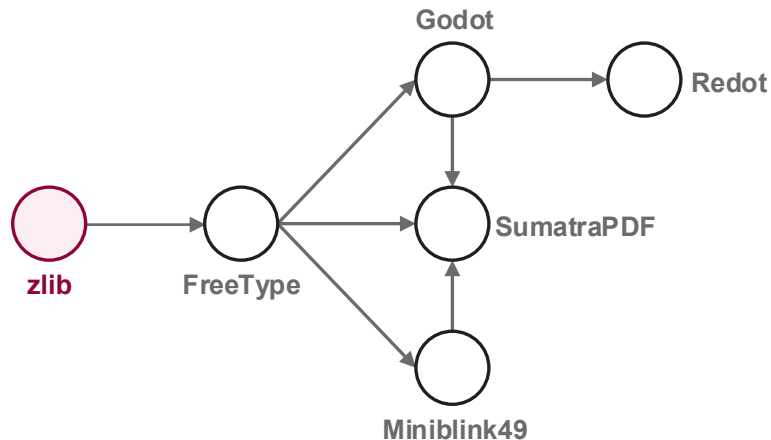
Step 2. Who reused whom?



Explicit evidence first, then majority vote

P3. Graph Construction

Which edges are real?



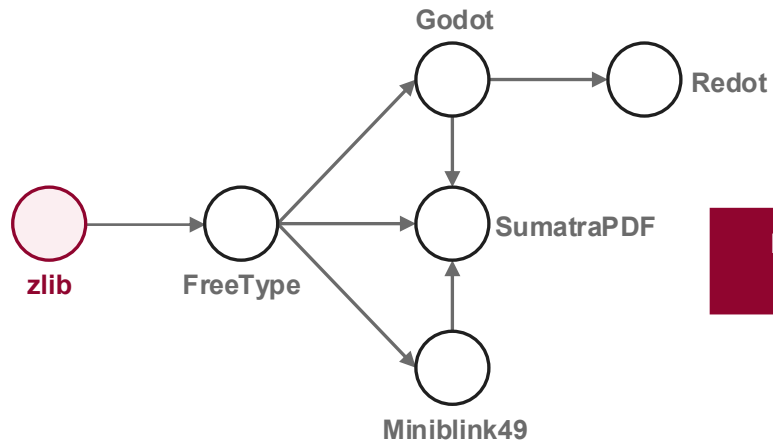
All candidate edges

noisy, contradictory

Explicit edges are always kept; implicit edges must pass the rules

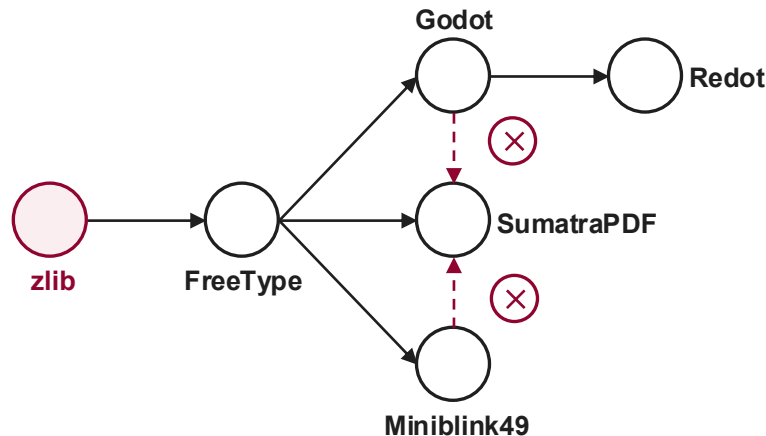
P3. Graph Construction

Which edges are real?



All candidate edges

noisy, contradictory



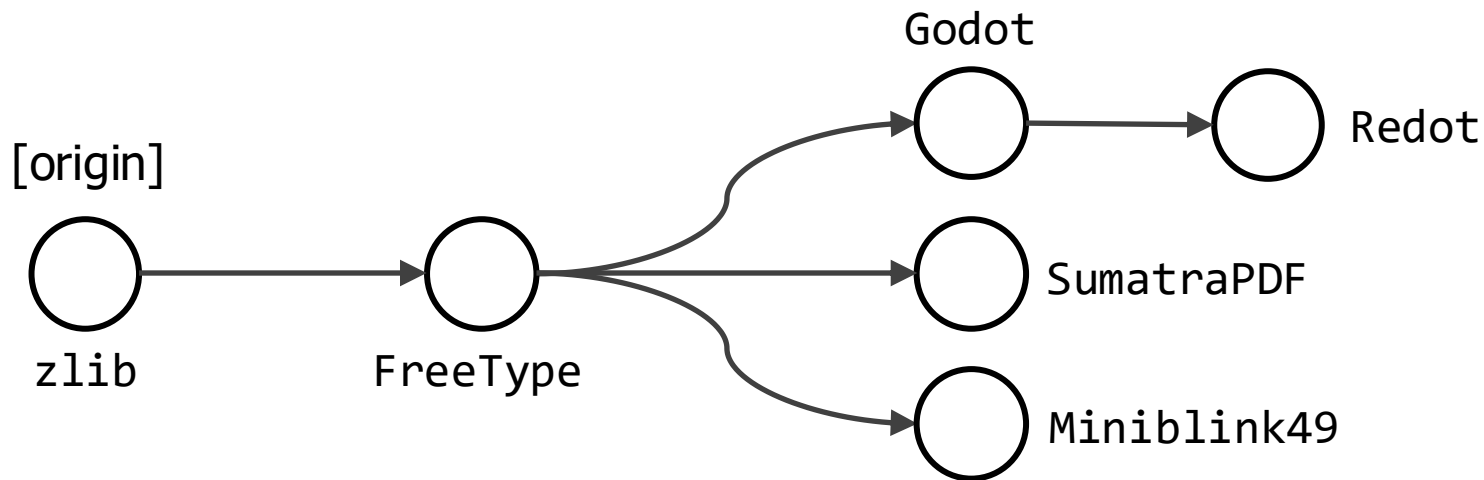
Reuse genealogy graph

one origin, many descendants

Explicit edges are always kept; implicit edges must pass the rules

Reuse Genealogy Graph

Example: reuse genealogy of zlib



Result: a DAG of how code actually propagates from one origin

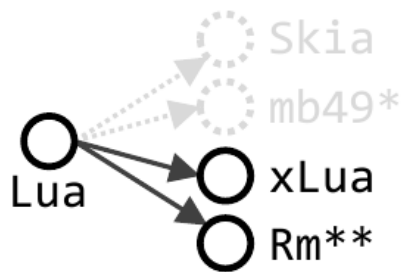
Evaluation

Accuracy – comparison with CNEPS & V0Finder

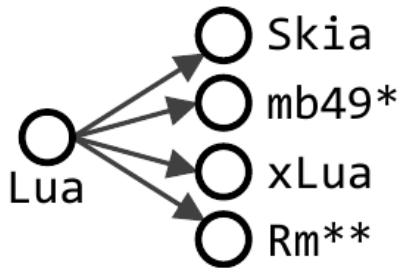
- **Pool:** top 2,500 C/C++ GitHub projects (2,006 with semantic versioning)
- **Targets:** 20 most-reused OSS from CENTRIS (ICSE '21)
- **Ground truth:** 1,600+ reuse relationships manually verified
- **Baselines:** CNEPS (dependency), V0Finder (vulnerability propagation)

Evaluation

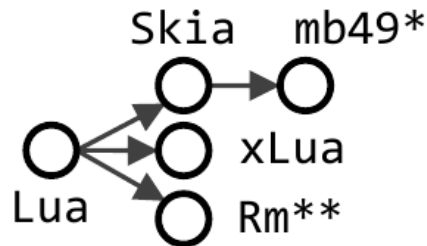
Key differences among V0Finder, CNEPS, and GANADI



(a) V0FINDER



(b) CNEPS



(c) GANADI

vs. V0Finder (USENIX SECURITY '21)

sees reuse only where vulnerable code moved

**misses reuse that carries
no known vulnerability**

vs. CNEPS (ICSE'24)

sees only one-step, explicit reuse

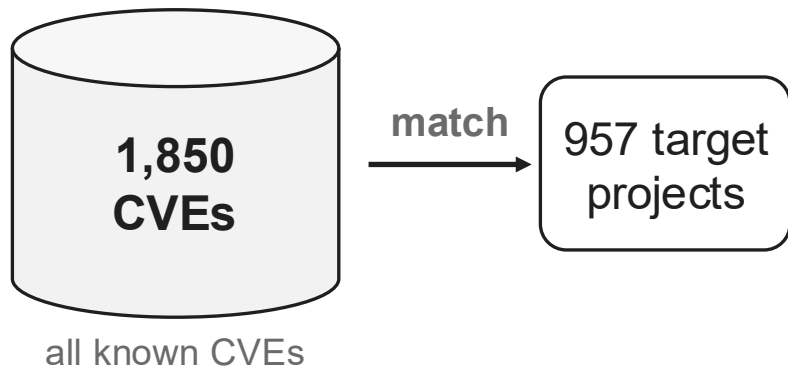
**misses reuse through
intermediate projects**

Each tool evaluated in its own setting - GANADI achieves far higher recall in

Application: Vulnerability Detection

Restrict the search space to CVEs within the same genealogy

Without genealogy

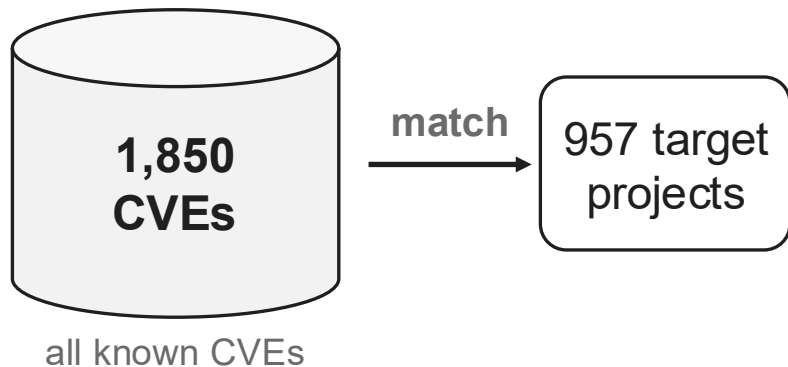


many false positives

Application: Vulnerability Detection

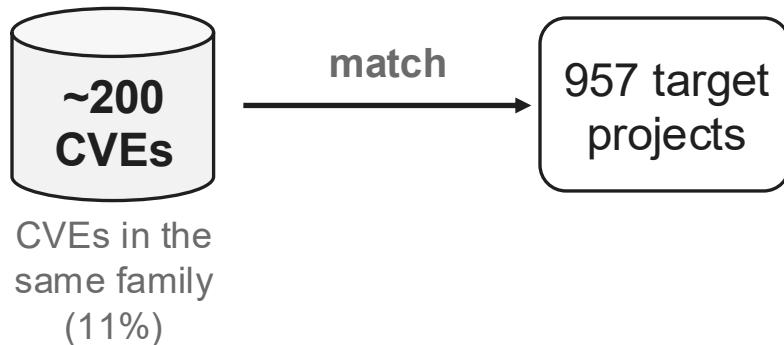
Restrict the search space to CVEs within the same genealogy

Without genealogy



many false positives

✨ **With genealogy** ✨



Precision ↑ 13–14 %
(Fewer unrelated CVEs → fewer false)

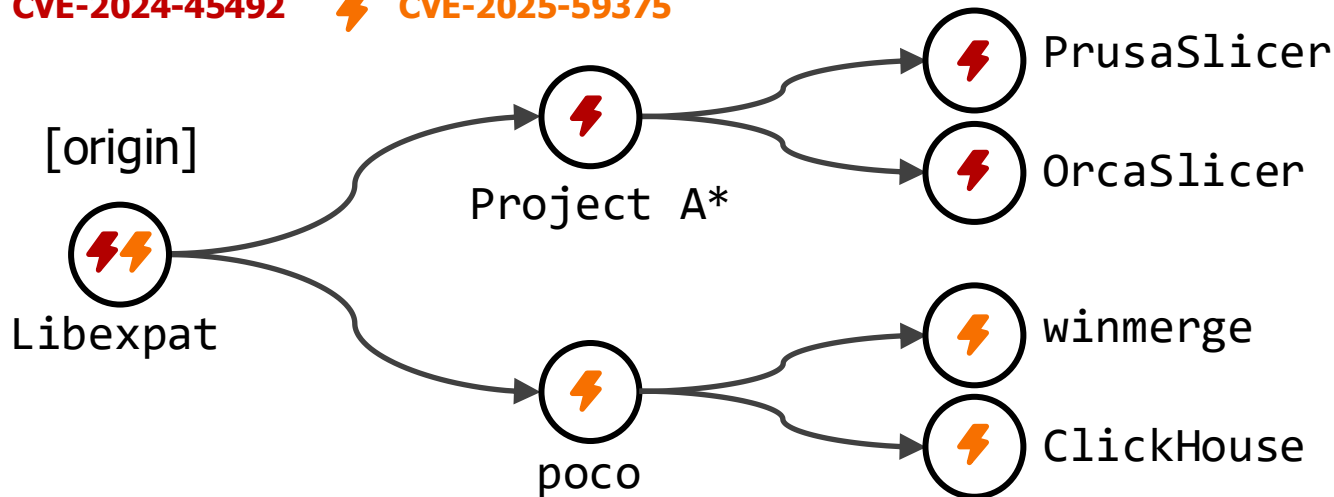
Higher precision with 1/9 of the CVEs, and clear propagation paths

Real-world Impact

Previously unknown vulnerabilities uncovered via genealogy

⚡ CVE-2024-45492

⚡ CVE-2025-59375



48 confirmed
Vulnerabilities Found

23 patched
Patched after our disclosure

1 CVE assigned
(CVE-2025-26269)

Real-world Usability

What a reuse genealogy enables in the supply chain

SBOM completeness

Transitive components and their reuse paths (pedigree)

Vulnerability propagation

Which downstream projects actually inherited a CVE

Vulnerability management

Clone detectors check only CVEs on the genealogy

License tracking

How licenses and code changes accumulate along the path

Conclusion

- **Genealogy:**
 - We reconstruct how OSS code actually propagates (the first for C/C++)
- **GANADI is accurate:**
 - It captures transitive reuse that prior tools miss
- **It works for supply chain security:**
 - 48 unpatched vulnerabilities found, 23 patched, 1 CVE



Thank you for your attention

Repository: <https://github.com/KIMDONGYEON00/GANADI>

Contact ʘ

- Dongyeon Kim (kimdongyeon@korea.ac.kr) 🐕
- Computer and Communication Security Lab (<https://ccs.korea.ac.kr>) ㅈ
- Software Security and Privacy Lab (<https://ssp.korea.ac.kr>) ㅈ