About

- 8 minutes per presentation

CHAINS PHD FESTIVAL

Research. Connect. Celebrate.

RESEARCH.
CONNECT.
CELEBRATE.

CHAINS PHD FESTIVAL

IDEAS.
IMPACT.
INSPIRE.

2025

WHERE BRILLIANT MINDS MAKE NOISE

RESEARCH | CONNECT | CELEBRATE

Polyseed: Reducing trust assumptions with diverse binary seeds

Language-based Supply Chain Security

Past

- GHunter: Server side prototype pollution
- NodeShield: Runtime Enforcement of Security-Enhanced SBOMs

Current

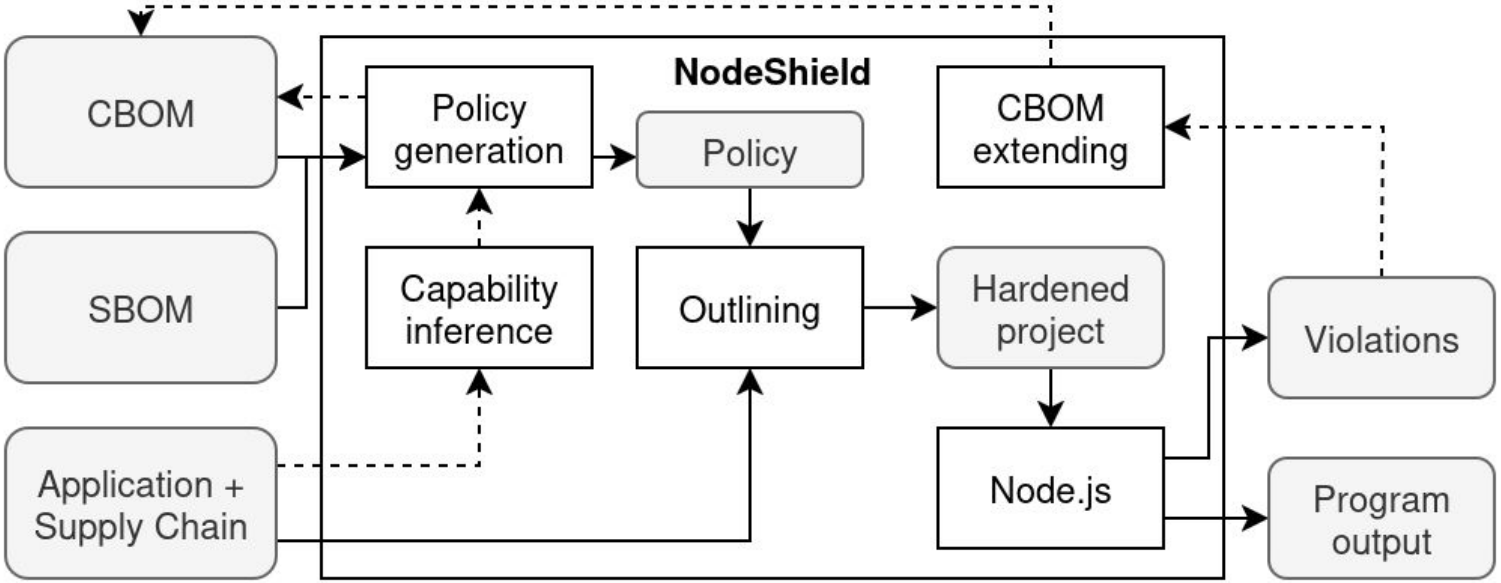
- Client side prototype pollution (websites, Browser extensions)
- Formalizing code reuse attacks

Exploring

- GitHub Actions Security
  - Checksums, Reproducible builds, Immutable releases, Vulnerability detection, edge cases
- Build time supply chain attacks
  - Transpilers introducing vulnerabilities, Compilers stealing secrets

NodeShield

- **Start**: Application+SBOM
- **Infer**: Dependency capabilities
- **Run**: Hardened execution



Build time supply chain attacks

Transpilers introducing vulnerabilities

- Minify (save 5 chars): `let x = undefined` → `let x = 0[0]`
- What if: `Object.prototype[0] = "evil"`
- Well: `x == "evil"`

Compilers stealing secrets

- Embedding external files as strings/bytes during compilation
  - Go (//go:embed hello.txt): only reads local files ✅
  - Rust (include_str!("hello.txt")): reads files anywhere ❌

My work

Past

- SBOM.exe: Protecting against dynamically loaded class in Java
- Causes and Canonicalization of Unreproducible Builds in Java
- google/oss-rebuild
- Maven Class Hijack

Current

- chains-project/maven-lockfile
- Exploring ahead of time aspects of JVM

JVM Warm-up Problem

Parses JVM bytecode → Create memory representations → Link classes to each other → Verifies the bytecode → Compile to native code at runtime (Just in time compilation)

How is Java solving this problem?

Create a cache of classes beforehand

Class Data Sharing (2004) - list of all common classes package with JDK

```
java -Xshare:off -cp HelloWorld
```

Application CDS (2017) - allows you to cache classes related to application

Ahead of Time Compilation (AOT) Cache (2023) - same as application CDS but:

a. Also verifies the classes beforehand (JDK 24)
b. Also links them (JDK 24)
c. Does JIT profiling (JDK 25)
d. Does JIT compilation (~JDK 27)

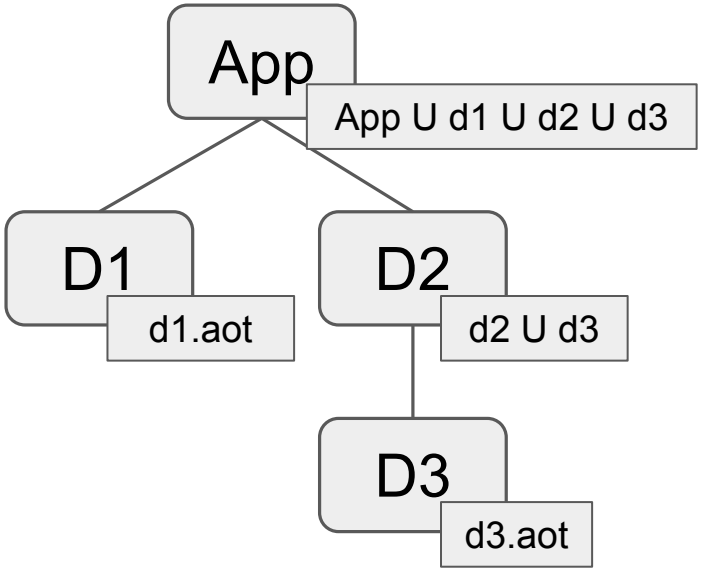
Using AOTCache (JDK >= 25)

1. You need to record them
   ```
   java -XX:AOTCacheOutput=cache.aot -jar pdfbox.jar
   export:text -i input.pdf
   ```
2. Now use them
   ```
   java -XX:AOTCache=cache.aot -jar pdfbox.jar export:text
   -i input.pdf
   ```

This gives you a 42% performance benefit.

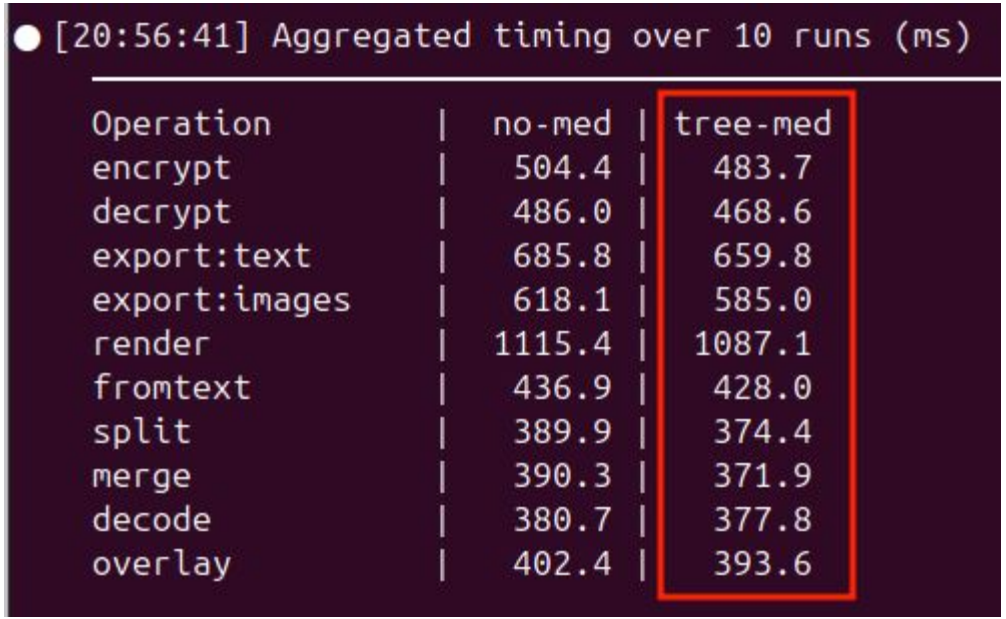
However, the workload should not change much and classpath should also be exactly the same.

AOTCache as a software supply chain artifact



- More general cache for all kinds of workload
- The app developer follows a distributed training model
- A library update only requires update of the library AOTCache

Preliminary Results (comparing with production JDK 25)



[20:56:41] Aggregated timing over 10 runs (ms)

Operation	no-med	tree-med
encrypt	504.4	483.7
decrypt	486.0	468.6
export:text	685.8	659.8
export:images	618.1	585.0
render	1115.4	1087.1
fromtext	436.9	428.0
split	389.9	374.4
merge	390.3	371.9
decode	380.7	377.8
overlay	402.4	393.6

- io
- fontbox
- pdfbox
- tools
- pdfbox-jbig2
- apache-commons-io
- commons-logging-workload
- bc-java-prov-workload
- bc-java-util-workload
- bc-java-pkix-workload

Security Concern

```
java -XX:AOTCache=merged.aot -jar pdfbox.jar export:text
```

Where does class come from? AOTCache or jar?

It is possible to hide a malicious class in aot and distribute it.

Breaking Dependency Updates

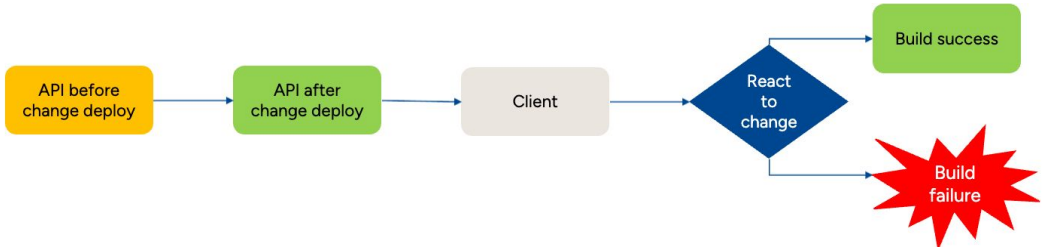
Breaking dependency updates occur when library APIs evolve:

Before update

```
public void process(){
    TTransport = factory.create();
    topen();
}
```

After update

```
public void process(){
    TTransport = factory.create();
    #Method signature change
}
```



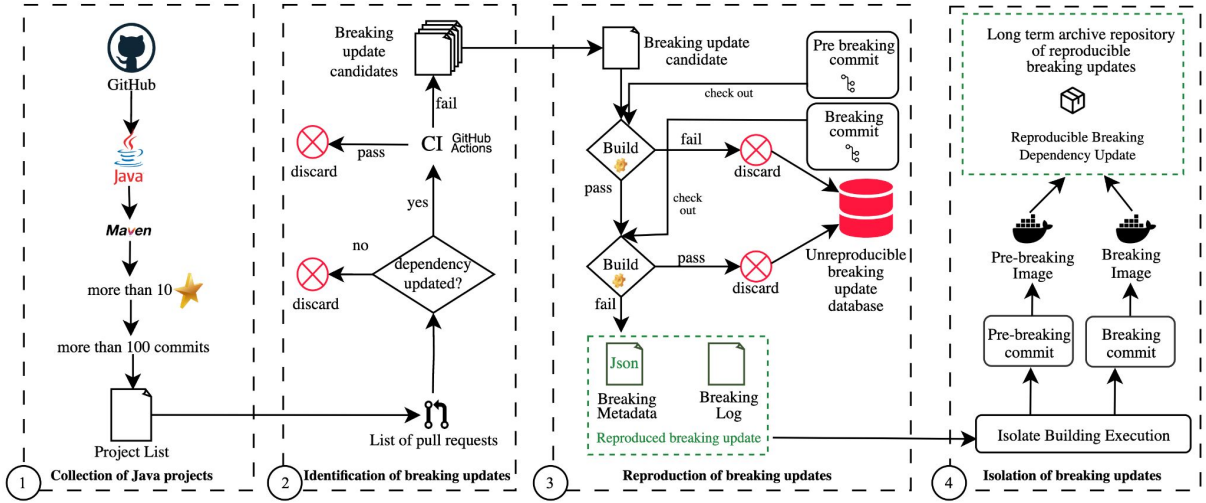
Client code breaks at compilation. Manual repair is:

- Time-consuming (hours per project)
- Error-prone (complex scope analysis required)
- Repeated across many independent projects

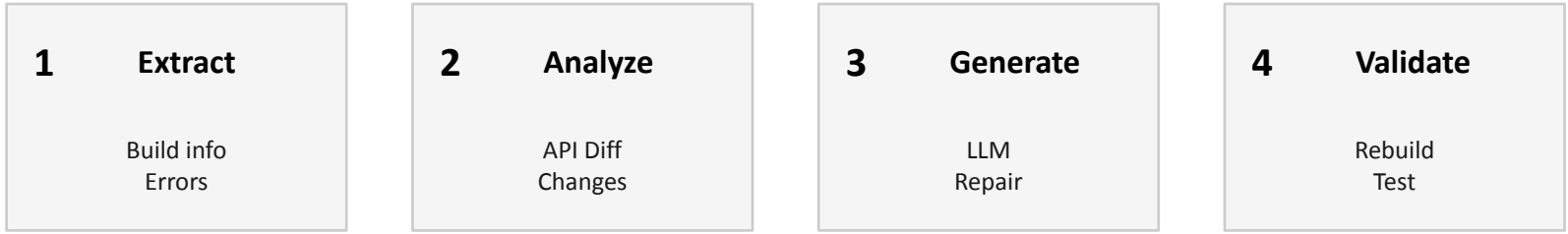
Real Problem

Bump Dataset:

- 571 breaking dependency updates
- Real projects from GitHub
- Includes compilation, test, and dependency resolution failures



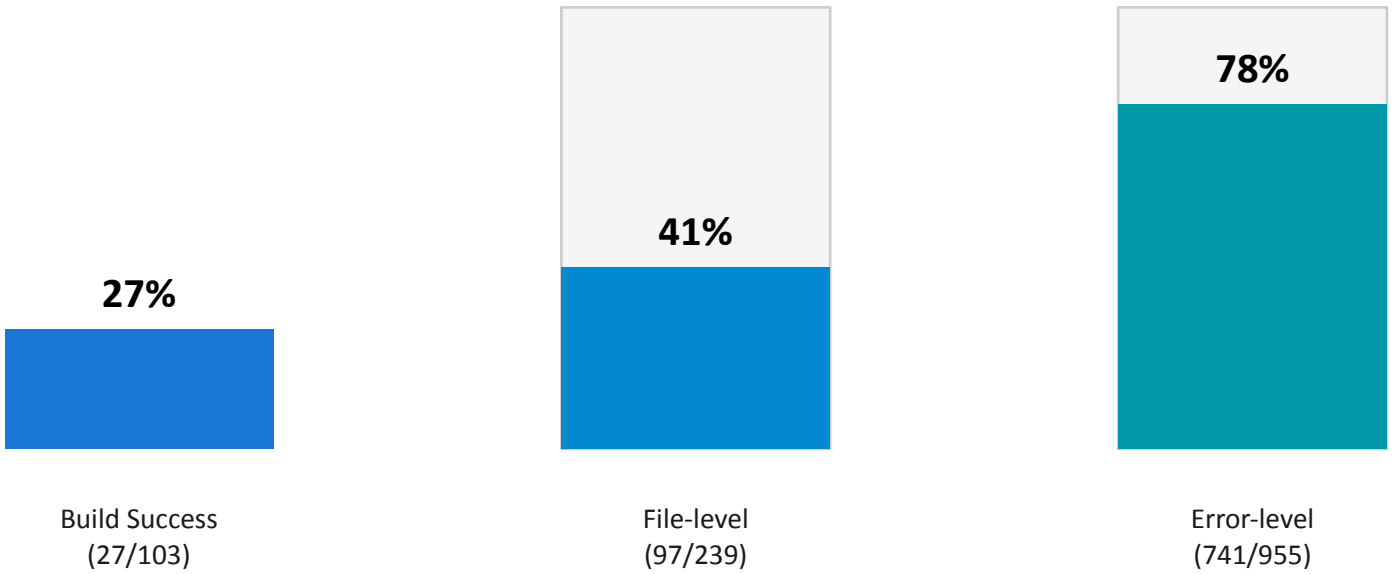
Byam: Fixing Breaking dependency updates with LLM



- Extract: Identify files with compilation errors, extract error messages and line locations
- Analyze: Compare dependency versions with japicmp, generate machine-readable API diff
- Generate: Context-rich prompts (client code, errors, APIDiff), Chain-of-Thought reasoning
- Validate: Replace files, rebuild project, verify success

Results: Byam on BUMP

o3-mini with surgical prompts



Partial repairs valuable even when full build fails. Significant reduction in manual debugging effort.

BigBag: From Repairs to Rules

Byam

Fix each project individually
High-quality but not reusable
Limited to single context

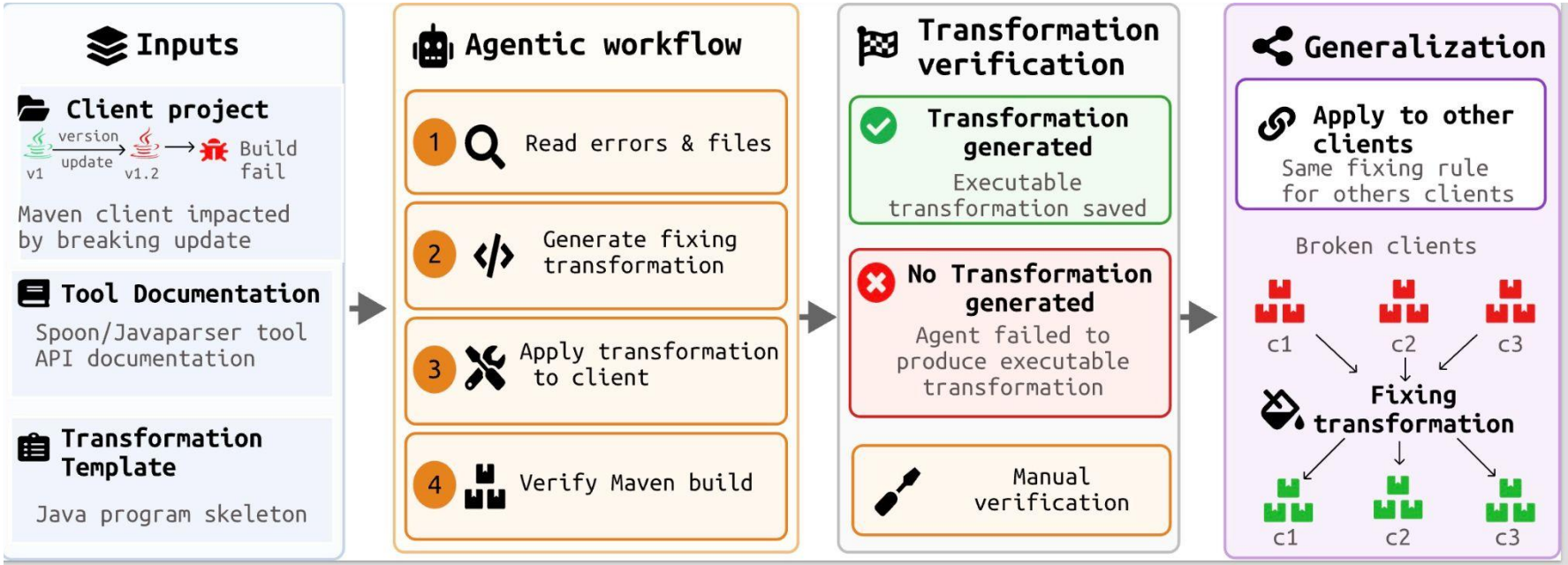
BIGBAG

Extract reusable AST rules
Apply across thousands
Production-grade automation

Architecture: Agentic pipeline (Gemini CLI, OpenCode, Claude Code) → **Spoon/JavaParser** rule generation → Cross-project testing

Example: FopFactory.newInstance() → FopFactory.newInstance(File)
Applies to all clients automatically

BigBag: Overview



Summary

1 **Byam**

27% full repair, 78% error-level
Prompt engineering outperforms generic
o3-mini best performer

2 **BIGBAG**

Agentic AST rule generation
Extract reusable rules
Apply across thousands

Code-based vulnerability propagation in software supply chains

Presented by: Yekatierina Churakova

Supervisor: Mathias Ekstedt, Larissa Schmid

Background: Why dependency graphs are not enough

1 Correctness of dependency graph problem

2 Static vulnerability scanners' precision

3 Higher-resolution approach

Related works

Threat-propagation modelling in software supply chains (Soeiro et al., 2023) uses a log-based formalism to assess how risks propagate across ecosystem elements.

Logical attack-graph analysis for software supply chains (Soeiro et al., 2025) identify end-to-end attack paths across dependencies and compromises.

Provenance-graph learning for software installations (Han et al., 2021) uses graph learning over execution provenance to detect malicious installer behavior.

Build Structural Context (Dependency + Call Graphs)

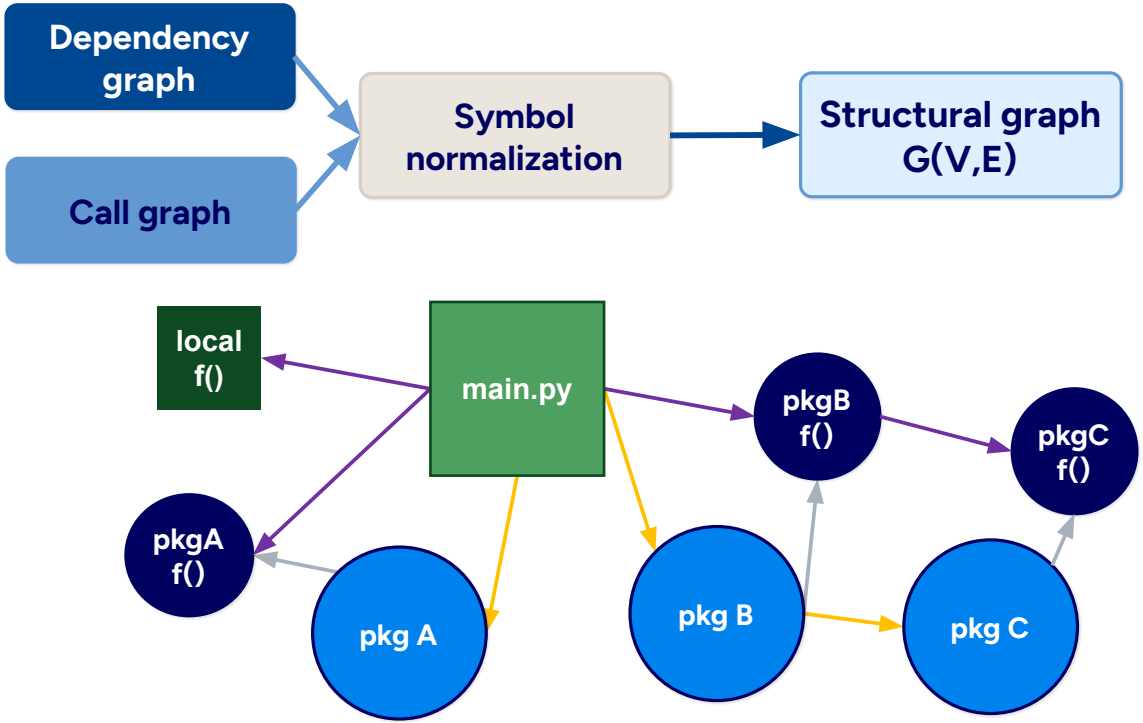
Extract dependencies from project manifests and lockfiles.

Build a dependency graph for module and package relationships.

Build a call graph with PyCG for function-to-function calls.

Normalize symbols and modules across project code and dependencies.

Result: directed structural graph G(V,E) of code entities and relations.



big green square – local module
small dark-green square – local function
big blue bubbles – imported package
small dark-blue bubbles – imported function

yellow arrows – imports
purple arrows – calls
grey arrows – package content

Extract and Map Security Findings

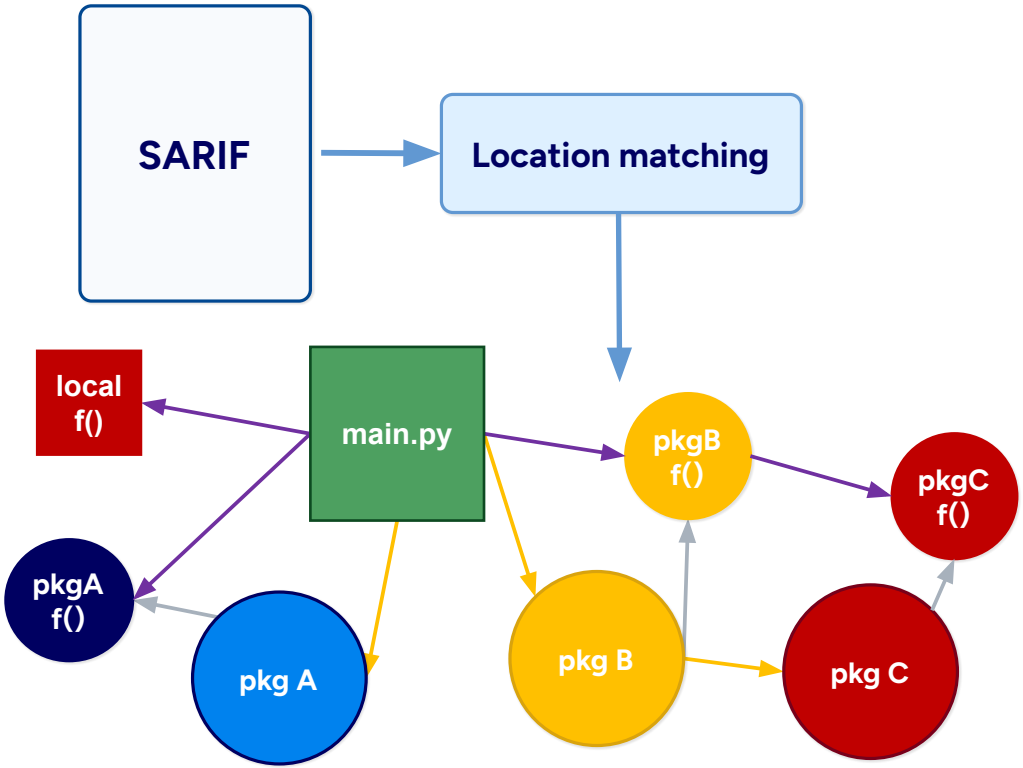
Run static code analysis query suites on the same code snapshot.

Export findings in SARIF (Static Analysis Results Interchange Format): rule, file, line, and message.

Map SARIF locations to module/function candidates in the graph.

Mark vulnerable or misconfigured nodes as risky (red nodes).

Keep provenance: SARIF file, URI, line, and code snippet.

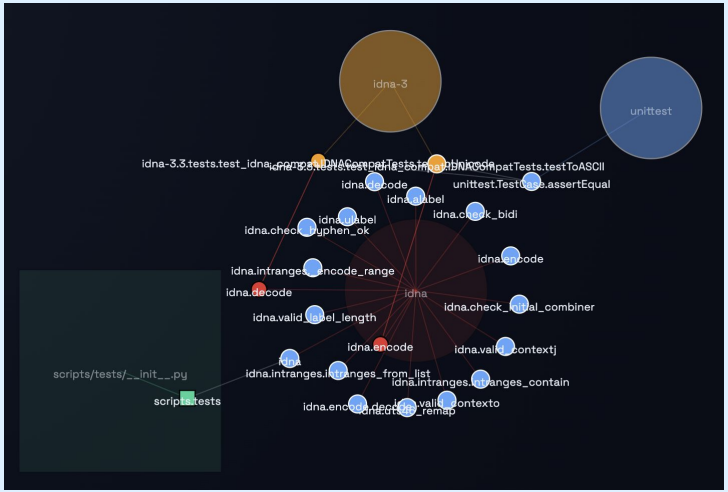


red nodes become risk seeds
yellow nodes pinpoint propagation

Repository: **public-apis**
Commit: **932e6f25cc47** (February 2026)
Dependency: **idna-3.3**
Vulnerability: **CVE-2024-3651**
Vulnerable function: **idna.encode()**

big green square – local module
small bright-green square – local function
big blue bubbles – imported package
small bright-blue bubbles – imported function
big yellow bubble – vulnerable dependency internal module
small bright-yellow bubbles – callers of the vulnerable functions
red bubbles – alerted nodes

Graph analysis: the vulnerable function are not reachable from project's entry point



idna **is pesent** in requirements.txt

```
certifi==2021.10.8
charset-normalizer==2.0.10
idna==3.3
requests==2.27.1
urllib3==1.26.8
```

But **is not** the list of function:

- error_message
- get_categories_content
- check_alphabetical_order
- check_title
- check_description
- check_auth
- check_https
- check_cors
- check_entry
- check_file_format

Future work

- Vulnerability prioritization and exploitation prediction
- Threat-model syntax development
- AI-assisted graph-analysis tasks
- LLM-ready security context