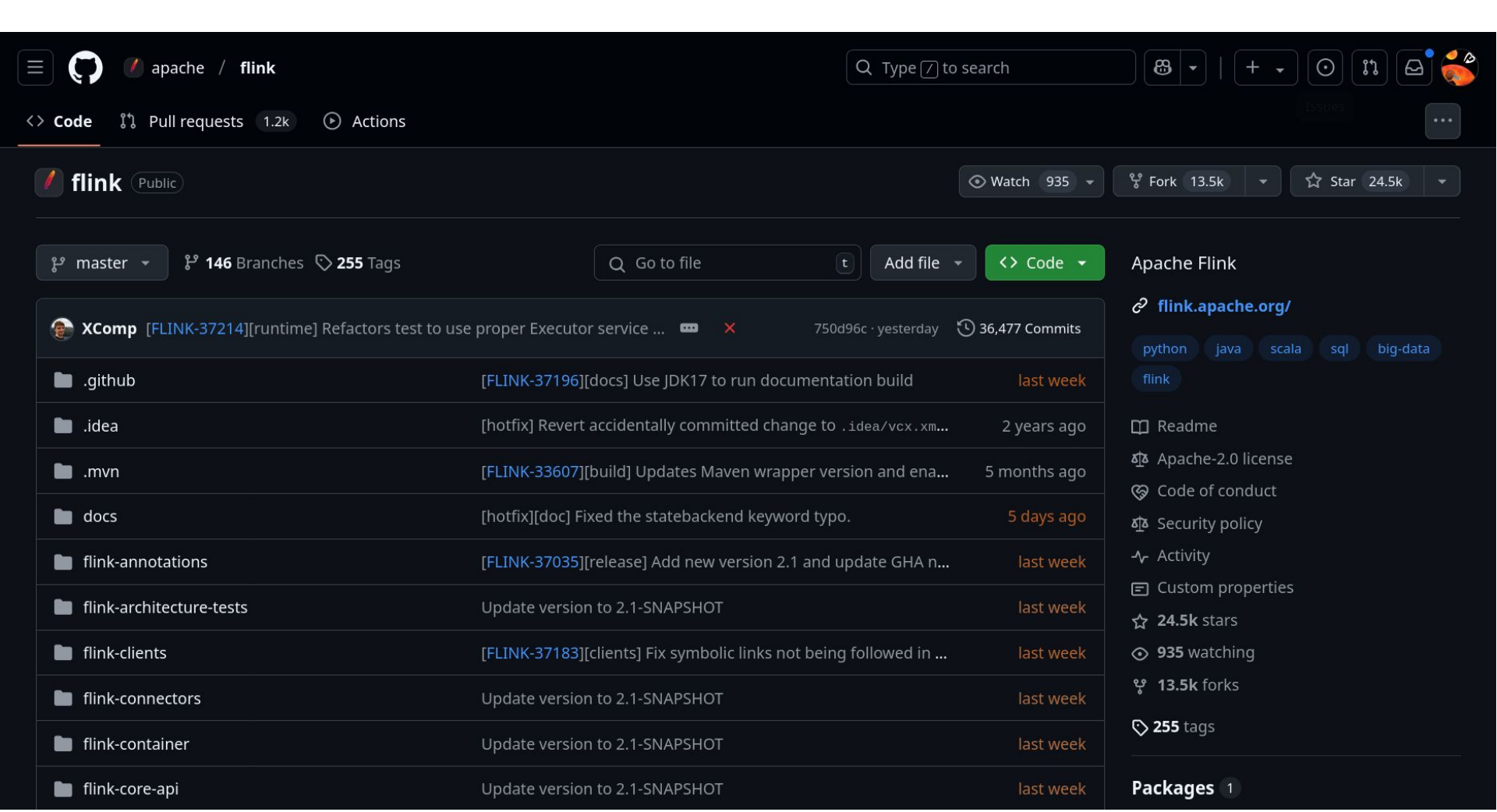
The CHAINS Project
Consistent Hardening and Analysis of Software Supply Chains

Prof. Martin Monperrus, KTH Royal Institute of Technology

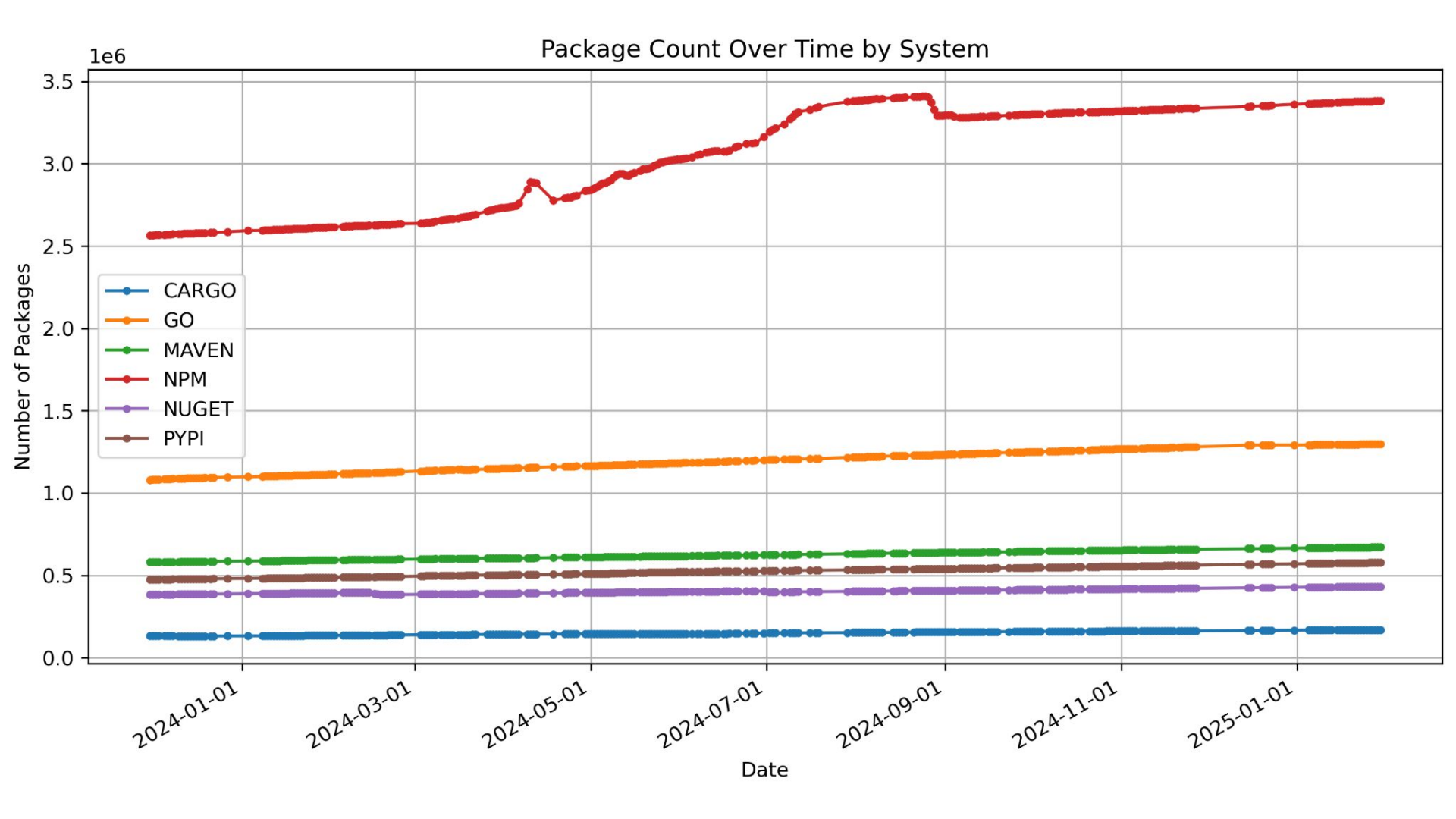
Sep 18 2026

Overture


apache / flink
Type / to search
Code
Pull requests 1.2k
Actions
flink Public
Watch 935
Fork 13.5k
Star 24.5k
master
146 Branches
255 Tags
Go to file t
Add file
Code
XComp [FLINK-37214][runtime] Refactors test to use proper Executor service ...
750d96c · yesterday
36,477 Commits
.github [FLINK-37196][docs] Use JDK17 to run documentation build last week
.idea [hotfix] Revert accidentally committed change to .idea/vcx.xm... 2 years ago
.mvn [FLINK-33607][build] Updates Maven wrapper version and ena... 5 months ago
docs [hotfix][doc] Fixed the statebackend keyword typo. 5 days ago
flink-annotations [FLINK-37035][release] Add new version 2.1 and update GHA n... last week
flink-architecture-tests Update version to 2.1-SNAPSHOT last week
flink-clients [FLINK-37183][clients] Fix symbolic links not being followed in ... last week
flink-connectors Update version to 2.1-SNAPSHOT last week
flink-container Update version to 2.1-SNAPSHOT last week
flink-core-api Update version to 2.1-SNAPSHOT last week
Apache Flink
flink.apache.org/
python java scala sql big-data flink
Readme
Apache-2.0 license
Code of conduct
Security policy
Activity
Custom properties
24.5k stars
935 watching
13.5k forks
255 tags
Packages 1

Demo

- cd ~/martin-no-backup/
- git clone https://github.com/apache/flink
- cd ~/martin-no-backup/flink/
- mvn dependency:tree
- mvn dependency:tree -DoutputType=dot -DoutputFile=deps.dot
- mvn dependency:tree -DoutputType=json -DoutputFile=deps.json
- cp ~/deps-d3.html .
- php --server localhost:8080
- Take away: lots of dependencies

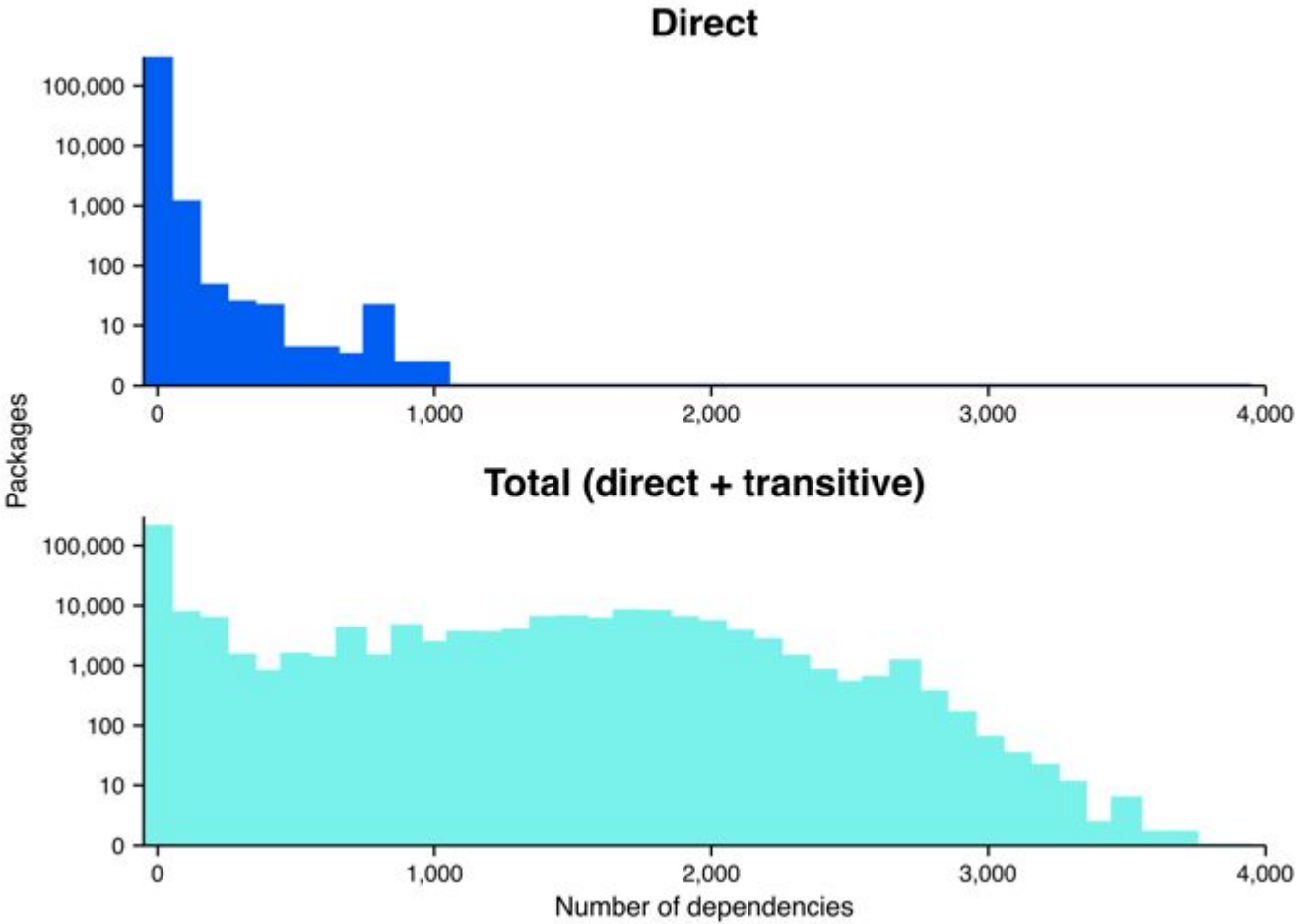

Package Count Over Time by System
1e6
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Number of Packages
CARGO
GO
MAVEN
NPM
NUGET
PYPI
2024-01-01
2024-03-01
2024-05-01
2024-07-01
2024-09-01
2024-11-01
2025-01-01
Date

Definition of Software Supply Chain

“The set of all dependencies and tools used to build, deploy and run a software system”

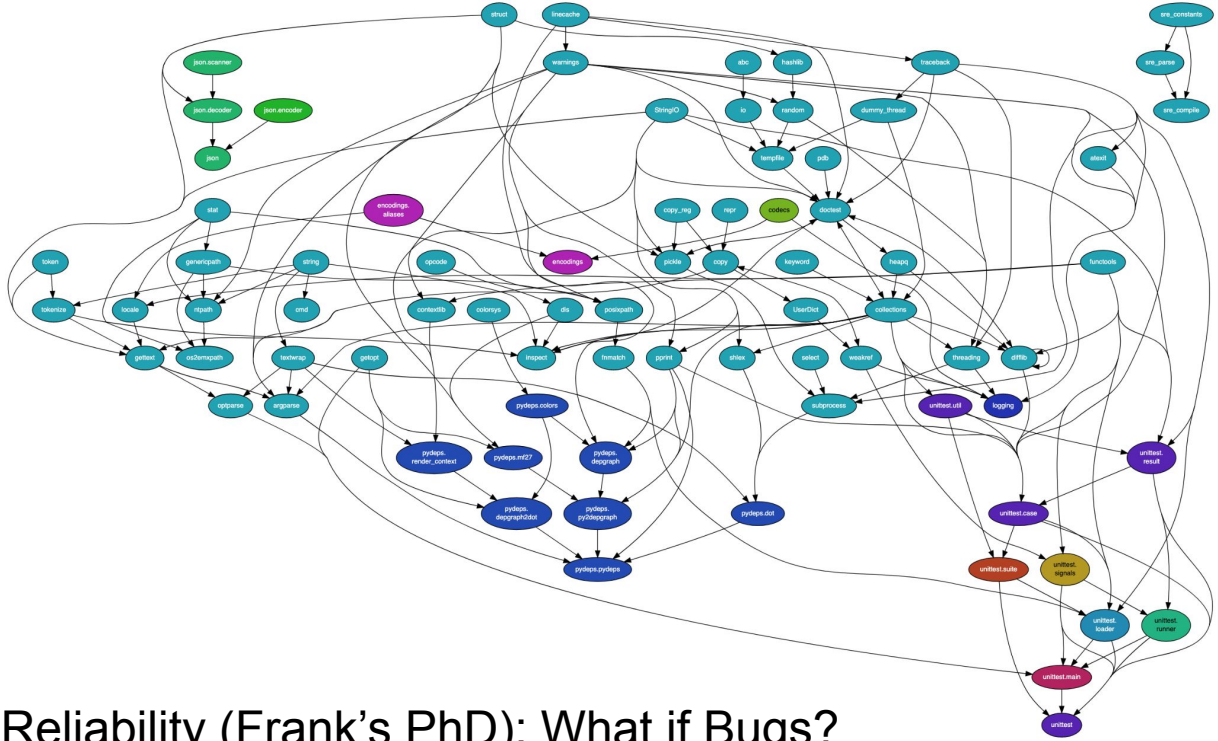
Starting with

- The dependencies (Maven, NPM, etc)
- The compiler
- The pipelines (CI/CD)



https://blog.phylum.io/hidden-dependencies-lurking-in-the-software-dependency-network/

Problem statement

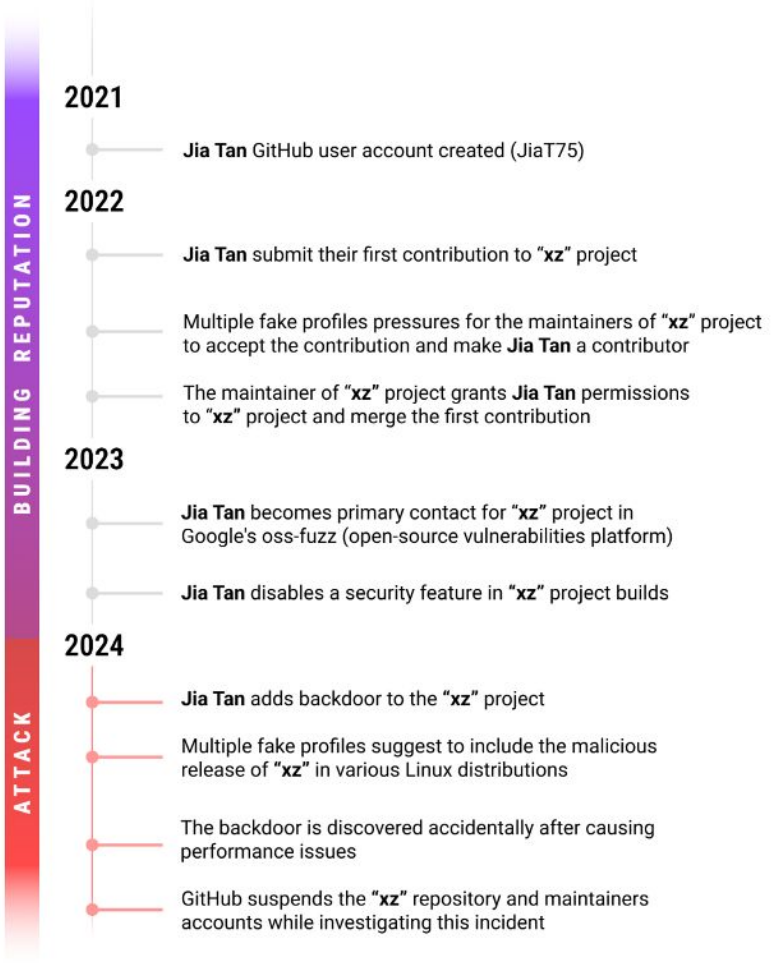


Reliability (Frank's PhD): What if Bugs?

Security (Today): Vulnerability? Malicious code injection?

Example attack: the XZ attack

or “How the world avoided full SSH catastrophe”



https://www.sonatype.com/resources/vulnerability-timeline

A History of Software Supply Chain Attacks

July 2017–Present

778,529 PIECES OF OPEN SOURCE MALWARE DETECTED... AND COUNTING

NOVEMBER 2024

Fake IP checker utilities on npm are crypto stealers

Recently identified npm packages called "node-request-ip", "request-ip-check" and "request-ip-validator" impersonate handy open source utilities relied upon by developers to retrieve an external IP address but instead target Windows

Act I - Supply Chain Attack

Trusting-Trust Attack against an Entire Linux Distribution through Binary Manipulation

Julien Malka
julien.malka@telecom-paris.fr
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Palaiseau, France

Aman Sharma
amansha@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Martin Monperrus
monperrus@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Stefano Zacchiroli
stefano.zacchiroli@telecom-paris.fr
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Palaiseau, France

Théo Zimmermann
theo.zimmermann@telecom-paris.fr
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Palaiseau, France

Abstract

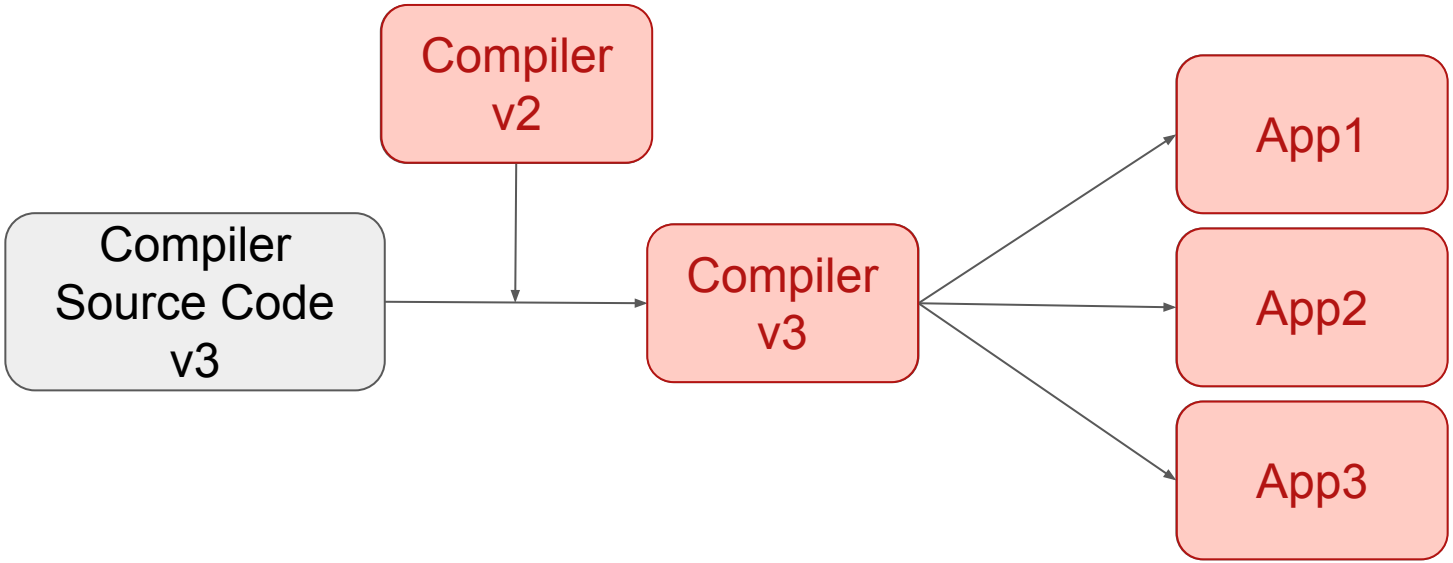
Ken Thompson's trusting-trust attack, in which a compromised compiler backdoors the programs it builds and reproduces the backdoor in subsequent rebuilds of itself, is often regarded as a threat specific to compilers. We show that it is not. We construct a complete trusting-trust attack around GNU `strip`, an ordinary build utility that neither inspects nor generates source code, using only manipulations of finished ELF files. In one bootstrap execution of the NixOS Linux distribution, a single tampered `strip` in the binary seed implants its payload into each later `strip` rebuilt from unmodified source. The infected `strip` in the final standard environment can then implant the payload into binaries of downstream packages, even though that environment has no runtime reference to the bootstrap seed. On a real nixpkgs revision, the attack builds a complete graphical installer without failures and backdoors almost every one of its binaries, enabling arbitrary malicious behavior of the subverted packages.

1 Introduction

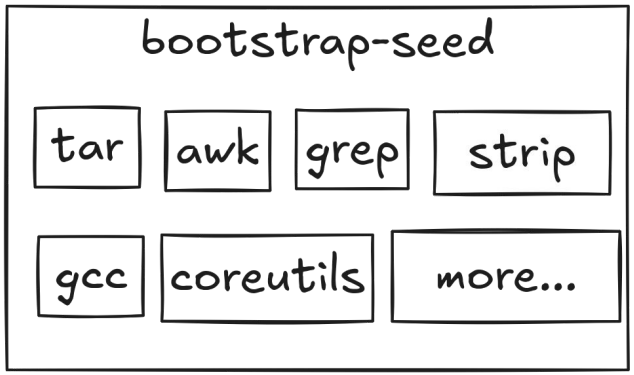
"Given enough eyeballs, all bugs are shallow." Eric S. Raymond [20] popularizes this statement as Linus's law in his essay on open source software development. It means that making source code publicly available for review, including by security experts, makes free and open source software less likely to contain bugs, security vulnerabilities, or backdoors. Yet software must first be compiled before it can run, which limits the trust that source availability alone provides. As Ken Thompson showed in his Turing Award lecture [23], some compromises remain invisible to source code review. In Thompson's construction, a malicious compiler recognizes both a security-sensitive program and the source of the compiler itself. It inserts a backdoor into the former and reproduces this logic in the latter. The malicious source can then be removed: recompiling unmodified compiler source with the compromised binary propagates the compromise.

Thompson's attack, dubbed "trusting trust", was more than a thought experiment because he implemented a prototype. However,

Trusting Trust Attack (Thomson 1984)



Nix Bootstrapping



2) binaries are built from source and strip from seed runs on them

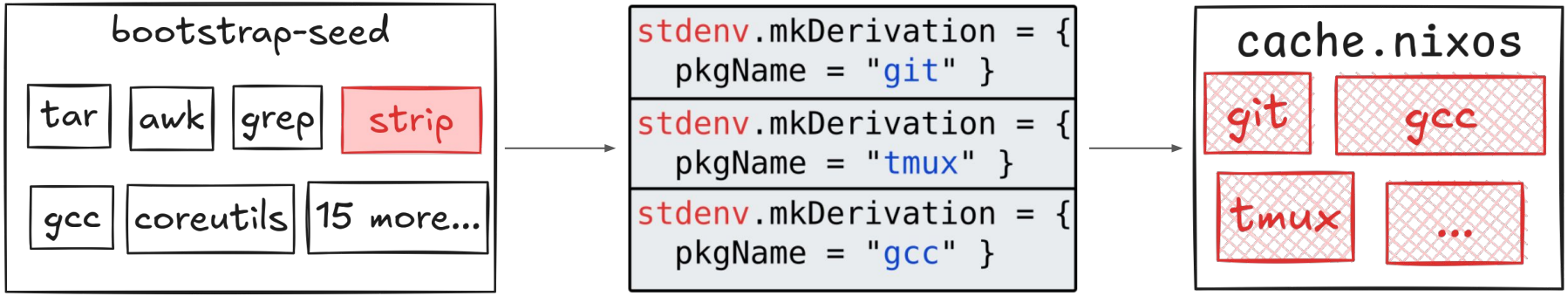
3) gcc is build from source using the rebuilt utilities

4) final packages and stdenv is built; no requisite on seed

```
stdenv.mkDerivation {
  pname = "git";
  version = "2.45.2";
  src = fetchurl {
    url ="git-scm.com";
    sha256 ="abcdef";
  };
};
```

https://github.com/NixOS/nixpkgs/blob/fef9403a3e4d31b0a23f0bacebbec52c248fbb51/pkgs/stdenv/linux/bootstrap-files/x86_64-unknown-linux-gnu.nix

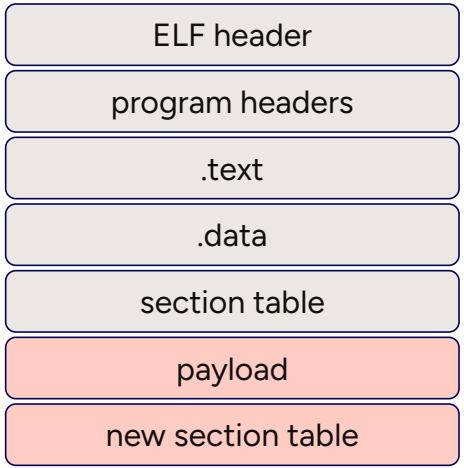
Attack



The attacker replaces one executable in the distribution's binary bootstrap seed.

Three ELF transformations

APPEND



REPURPOSE

```
NOTE offset virtaddr physaddr
filesz memsiz flags align

LOAD offset virtaddr physaddr
filesz memsiz flags align
```

REDIRECT

e_entry
0x401050 → payload

e_shoff
0x3400 → section table

The injector never rewrites existing code or data. Execution now begins in the payload, which returns to the host's original entry point.

Methodology: Why strip?

C1 — Must be part of bootstrap seed

strip, patchelf, tar, xz, make, awk, cp, install

C2 — Writes a finished binary

strip, patchelf, cp, install

C3 — Writes its own successor

strip, patchelf

Evaluation: core NixOS

3,790

of 3,799 user-invokable binaries in the trojaned graphical installer print the infection marker. The one exception is firefox 147.0.3, whose recipe passes --disable-strip and opts out of the phase entirely.

Buildability and runtime behavior

Nothing fails to build

The complete graphical ISO builds under the trojaned seed. Its runtime closure holds 1,199 package outputs, 3,799 executables and 6.16 GB on disk.

Nothing fails its tests

The NixOS functional tests drive a full GNOME session and a from-scratch install. Every binary in that VM, gnome-shell and nautilus included, came from the trojaned strip. No test fails.

nixpkgs at revision fef9403a3e4d, GNU binutils 2.44, x86_64-linux, about 2,000 packages.

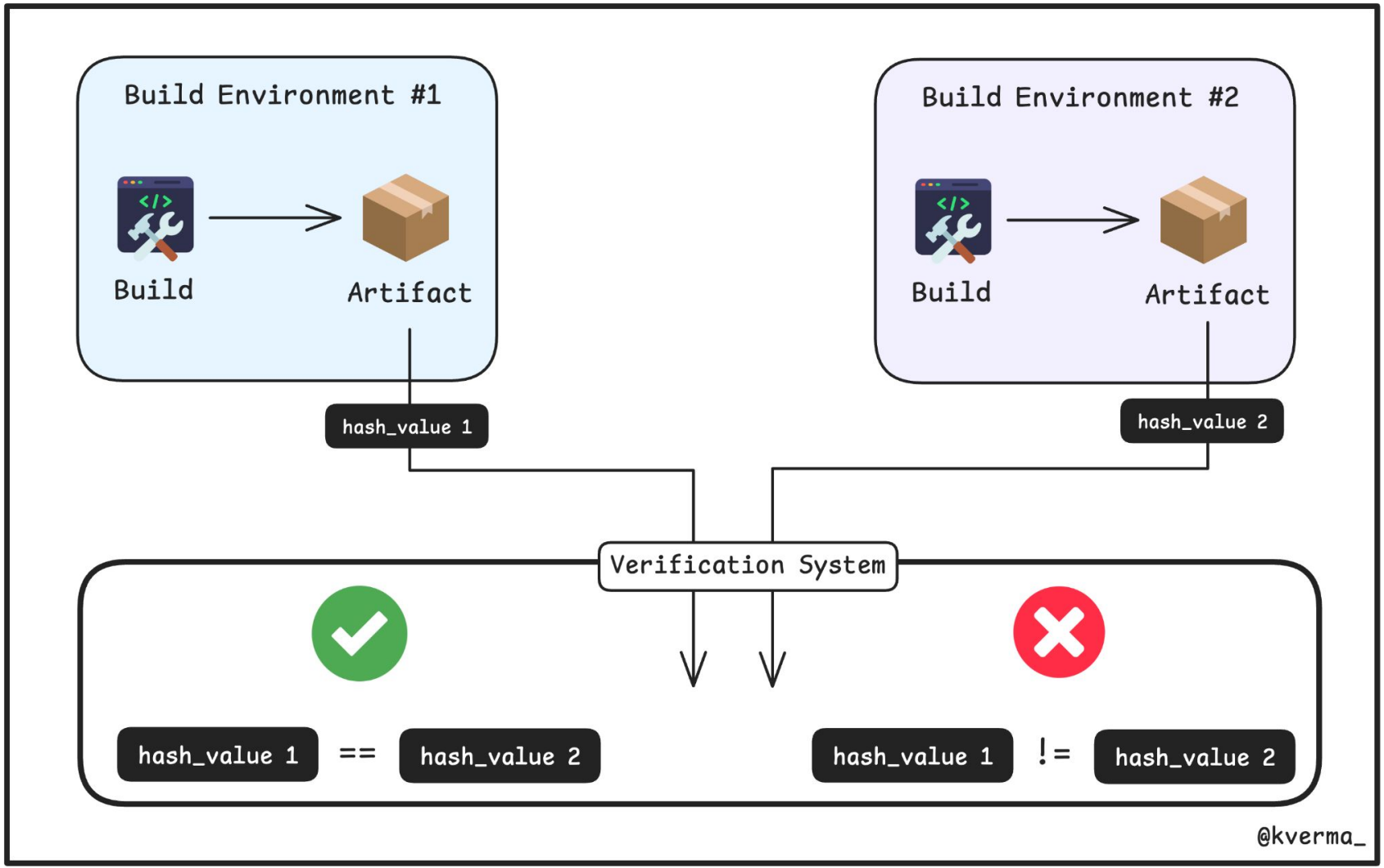
Language independence

ECOSYSTEM	EXAMPLE BINARY	SIZE	INFECTED
C / C++	git	56,313 KB	yes
Python	pydoc3.13	110 KB	yes
Rust	rsvg-convert	8,065 KB	yes
Go	captree	1,964 KB	yes
Lua	lua	328 KB	yes

We’re the first to demonstrate a trusting trust attack at the scale of an entire distribution / ecosystem

Trusting-Trust Attack against an Entire Linux Distribution through Binary Manipulation,
Proceedings of ACM SCORED 2026

Act II - Reproducible Builds


Build Environment #1
Build
Artifact
Build Environment #2
Build
Artifact
hash_value 1
hash_value 2
Verification System
hash_value 1
==
hash_value 2
hash_value 1
!=
hash_value 2
@kverma_

Unreproducible Builds in Java

Causes and Canonicalization of Unreproducible Builds in Java

In IEEE Transactions on Software Engineering, 2025.

Causes and Canonicalization of Unreproducible Builds in Java

Aman Sharma
KTH Royal Institute of Technology
Stockholm, Sweden
amansha@kth.se

Benoit Baudry
Université de Montréal
Montréal, Canada
benoit.baudry@umontreal.ca

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
monperrus@kth.se

Abstract

The increasing complexity of software supply chains and the rise of supply chain attacks have elevated concerns around software integrity. Users and stakeholders face significant challenges in validating that a given software artifact corresponds to its declared source. Reproducible Builds address this challenge by ensuring that independently performed builds from identical source code produce identical binaries. However, achieving reproducibility at scale remains difficult, especially in Java, due to a range of non-deterministic factors and caveats in the build process. In this work, we focus on reproducibility in Java-based software, archetypal of enterprise applications. We introduce a conceptual framework for reproducible builds, we analyze a large dataset from Reproducible Central, and we develop a novel taxonomy of six root causes of unreproducibility. We study actionable mitigations: artifact and bytecode canonicalization using OSS-Rebuild and jNorm respectively. Finally, we present Chains-Rebuild (improvements to OSS-Rebuild), a tool that raises reproducibility success from 9.48% to 26.60% on 12,803 unreproducible artifacts. To sum up, our contributions are the first large-scale taxonomy of build unreproducibility causes in Java, a publicly available dataset of unreproducible builds, and Chains-Rebuild, a canonicalization tool for mitigating unreproducible builds in Java.

Keywords

Reproducible Builds, Software Supply Chain, Canonicalization, Java

1 Introduction

The growing complexity of software supply chains [9, 56], coupled with the increasing frequency of supply chain attacks [1], raises concerns about software integrity. In such a fragmented ecosystem, relying solely on assumptions about the identity of the distributor [58] is no longer sufficient — what is needed is verifiable evidence that the software one installs corresponds exactly to its declared source. This challenge is especially acute in open source environments, where binaries are often distributed separately from their source code [11], making it difficult for users to independently validate what they are running. As a result, the focus is shifting toward techniques that can guarantee that what is built matches what was intended, regardless of who performs the build.

This technique is formally called "Reproducible Build" [10, 61]: builds are considered "reproducible" if and only if the build process is deterministic, where the same binary can be computed from the same source code by independent parties [25]. It helps prevent attacks on the build process, where an attacker modifies it to insert backdoors or malicious code into the software application to be built [41]. Any backdoor or malicious code would be detected by the verifier as the built artifact would not match the original artifact. This approach has already seen adoption in security and privacy critical environments, such as Bitcoin and Tor Browser [28]. Ensuring that clients for both of them are free of any backdoors injected by the build system.

Despite the promises of Reproducible Builds, ensuring and verifying reproducibility at scale remains technically challenging due to the multitude of spurious differences in build outputs (e.g., timestamps, file ordering). Those differences make binaries appear different even when built from the same source, with an untampered build pipeline [17]. Spurious differences hinder reproducibility and create a new attack surface. Several high-profile supply chain attacks have exploited unreproducible builds. For instance, in 2024 it was discovered that the official tarballs of XZ Utils, compression library used in almost all Linux distributions, contained a backdoor that enabled remote code execution, even though the backdoor was absent in the project's Git repository [44]. Similarly, the Ultralytics Python package, a widely used library for AI models, was compromised when attackers injected malicious code into the build pipeline, resulting in a PyPI release with a backdoor [26]. A related incident targeted the Solana ecosystem as well [7]. These incidents share a common vector: the artifacts distributed through registries or release tarballs differed from what could be obtained by building directly from the publicly available source code. If the build processes for these projects had been reproducible, users or third party rebuilders could have verified that the distributed binaries and packages matched the source. Any discrepancies would have immediately signaled tampering, thereby preventing such attacks [32].

In this work, we contribute to solving the problem of achieving build reproducibility in the context of enterprise software in Java. Recall that Java has consistently been one of the top 5 programming languages in the world for the past 5 years [2], underscoring its widespread use and relevance. Its importance in the context of finance [36, 56], government services [51], and military applications [3,4] highlights the need for reproducible builds in Java. We aim at building the most comprehensive taxonomy of unreproducible builds in Java, and the corresponding mitigation strategies. Furthermore, we perform a deep study how canonicalization — removal of non-deterministic differences — is a good solution for mitigating unreproducibility.

Our approach for analyzing unreproducible builds in Java is shown in Figure 1. We first propose an original framework for build reproducibility where we clearly define the roles of the builder and rebuilder and how both of them contribute to reproducible build

[1] https://www.sonatype.com/hubfs/SSCR-2024/SSCR_2024-FINAL-10-10-24.pdf

[2] https://innovationgraph.github.com/global-metrics/programming-languages
[3] http://pdf.cloud.opensystemsmedia.com/vita-technologies.com/Aonix.Jun07.pdf
[4] https://tsri.com/news-blog/press/u-s-department-of-defense-mainframe-cobol-to-java-on-aws

Verifiable Provenance of Software Artifacts with ZK Compilation

- Cryptographic proof binding of source code, compiler execution, and compiler output.

- Establish provenance by verifying a succinct proof. No need for re-compilation.

https://arxiv.org/abs/2602.11887

Verifiable Provenance of Software Artifacts with Zero-Knowledge Compilation

Javier Ron, Martin Monperrus
javierro@kth.se, monperrus@kth.se
KTH Royal Institute of Technology

***Abstract*—Verifying that a compiled binary originates from its claimed source code is a fundamental security requirement, called source code provenance. Achieving verifiable source code provenance in practice remains challenging. The most popular technique, called reproducible builds, requires difficult matching and reexecution of build toolchains and environments. We propose a novel approach to verifiable provenance based on compiling software with zero-knowledge virtual machines (zkVMs). By executing a compiler within a zkVM, our system produces both the compiled output and a cryptographic proof attesting that the compilation was performed on the claimed source code with the claimed compiler. We implement a proof-of-concept implementation using the RISC Zero zkVM and the ChibiCC C compiler, and evaluate it on 200 synthetic programs as well as 31 OpenSSL and 21 libsodium source files. Our results show that zk-compilation is applicable to real-world software and provides strong security guarantees: all adversarial tests targeting compiler substitution, source tampering, output manipulation, and replay attacks are successfully blocked.**

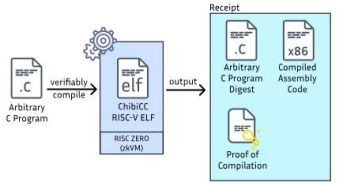


Fig. 1: Provenance guarantees in CosmicTurtle. The compiler is executed inside RISC Zero's zkVM. It records the computation trace while compiling the source program. It then generates a succinct cryptographic proof attesting that the compilation occurred on the claimed source code with the claimed compiler, i.e. cryptographically verifiable provenance.

I. Introduction

Establishing source code provenance, i.e., the ability to trace a binary back to the source files that were used to compile it and to demonstrate that the two correspond, is a fundamental security requirement for trustworthy software distribution [1]. Ensuring provenance of software artifacts is essential to mitigating supply chain attacks [2], [3] and maintaining public trust in open-source ecosystems.

Traditional compilation processes do not ensure provenance: when a user receives a compiled binary, there is no guarantee that they are also given the source code and that it was actually produced from this claimed source code. It is even uncommon to know the exact compiler that was used to compile the binary [4]. This creates significant security risks, as malicious actors could distribute binaries that differ from their purported source, potentially containing backdoors or other malicious code.

More recently, the reproducible build concept [5] has gained popularity, defined as independent parties being able to regenerate identical binaries from the same source, with byte-for-byte equality as strong evidence of provenance. Yet, reproducibility is brittle and expensive to maintain, especially for complex toolchains or non-deterministic build environments [6], [7]. Complementary efforts explore trusted execution environments (TEEs) to attest compilation and build processes [8], [9], but these approaches require specialized hardware and shift trust to hardware vendors. In this paper, the problem we are addressing is to provide strong guarantees of software provenance without the hurdle of full rebuilding and without relying on trusted proprietary hardware.

We propose a novel approach to provenance based on verifiable compilation through zero-knowledge virtual machines (zkVMs). By executing a compiler within a zkVM, our system produces both the compiled output –the binary– and a cryptographic proof attesting that the compilation ran on the claimed source code with the claimed compiler. Our approach provides cryptographic guarantees of provenance that can be independently verified by anyone without reproducing the build or trusting specialized hardware. Because zkVMs execute standard binaries, the approach is generally applicable to any compiler without requiring compiler modifications.

We evaluate our design on a mix of real-world and synthetic programs: 31 libsodium source files (crypto library), 21 OpenSSL source files (crypto library), as well as 200 randomly generated C programs produced by Csmith [10]. All 252 programs are successfully zk-compiled and verified with cryptographic proofs of provenance. Our systematic adversarial tests confirm that the core security properties of provenance hold: compilers cannot be substituted, source code cannot be tampered with, binary code also, and replay attacks are all detected and rejected by the verification process. In other words, our experiments fully validate our concept of zero-knowledge compilation.

The main contributions of this work are:

Verifiable Provenance of Software Artifacts with ZK compilation

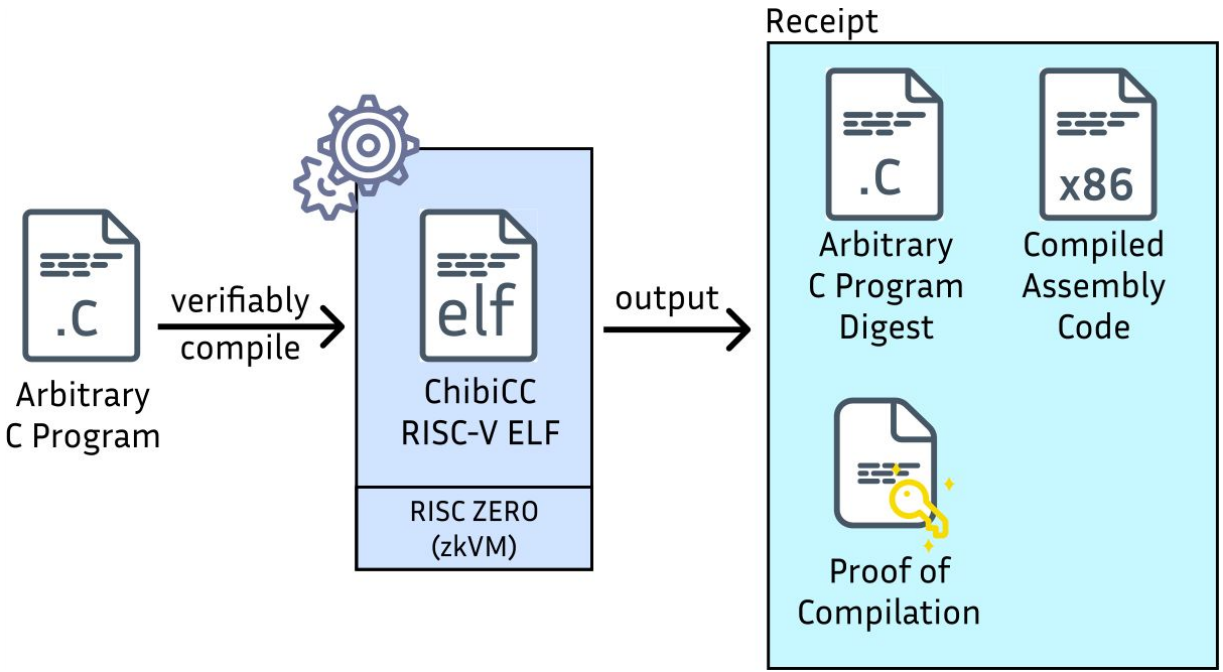
Enter **zero-knowledge Virtual Machines (zkVM)**

- zkVM executes a program and produces a cryptographic proof of execution [4].

- The proof is a witness of the program's execution trace and corresponding state *instruction by instruction*.

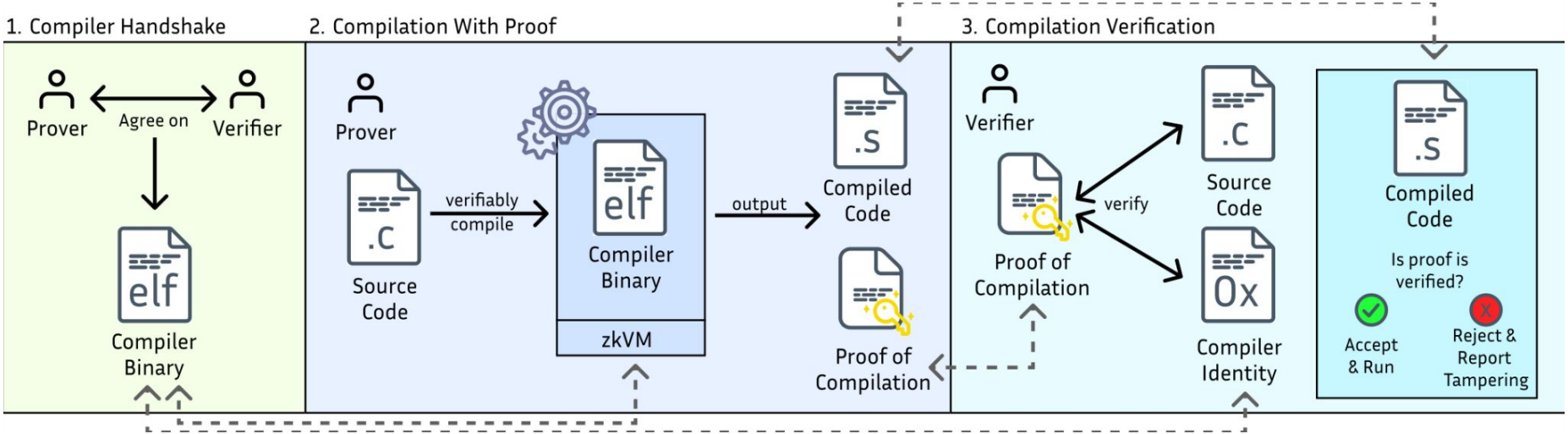
[4] Tim Dokchitser and Alexandr Bulkin. Zero knowledge virtual machine step by step. Cryptology ePrint Archive, Paper 2023/1032, 2023.

CosmicTurtle: Verifiable Provenance of Software Artifacts with ZK compilation



Verifiable Provenance of Software Artifacts with ZK compilation

Prototype: CosmicTurtle



Verifiable Provenance of Software Artifacts with ZK compilation

Evaluation

Applicability:

- 200 CSmith synthetic programs.
- 52 real-world source files from OpenSSL and libsodium.

Security:

- Source code tampering
- Compiler tampering
- Output tampering
- Replay attacks

Verifiable Provenance of Software Artifacts with ZK compilation

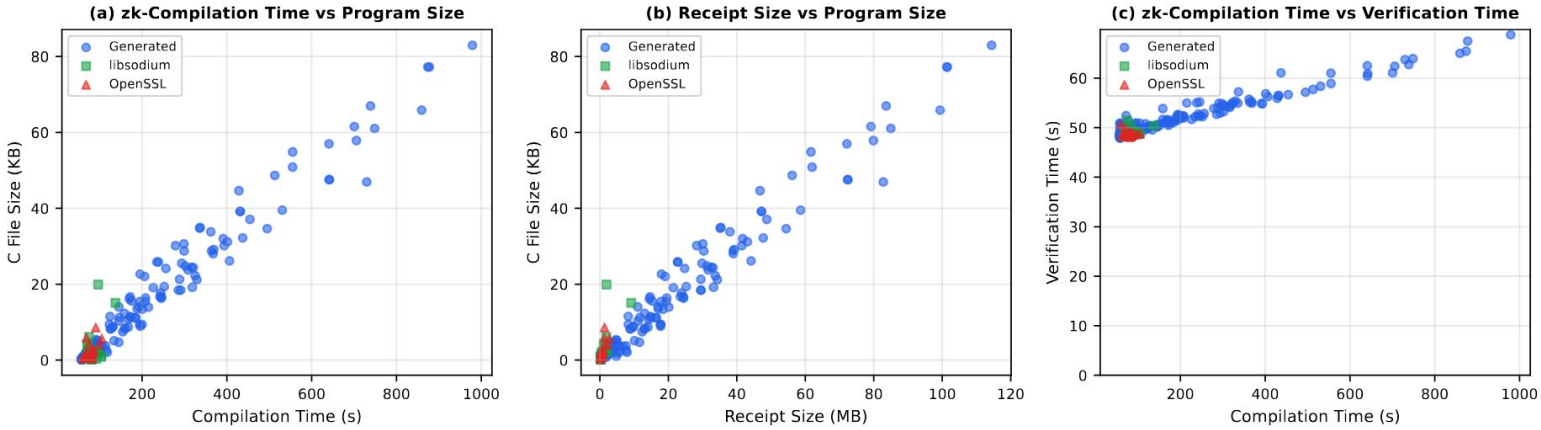
Results

Metric	Random Programs (200)			OpenSSL (31)			libsodium (21)		
	Min	Median	Max	Min	Median	Max	Min	Median	Max
C File Size (KB)	0,14	2,37	82,94	0,42	1,85	8,57	0,19	1,21	19,91
Receipt Size (MB)	0,26	2,79	114,41	0,21	0,52	2,74	0,25	0,52	9,15
Standard Compile Time (s)	0,0006	0,0092	0,0665	0,0012	0,0021	0,0046	0,0007	0,0023	0,0124
zk-Compile Time (s)	57,01	78,91	979,12	59,71	80,55	104,83	68,25	82,28	137,21
Verify Time (s)	47,93	49,88	68,75	48,12	48,64	50,64	48,17	48,81	51,34

TABLE I: Summary statistics across 200 randomly generated test programs, 31 OpenSSL files, and 21 libsodium files.

Verifiable Provenance of Software Artifacts with ZK compilation

Results



Verifiable Provenance of Software Artifacts with ZK compilation

Takeaway:

- zk compilation gives unprecedented provenance and security guarantees.

- Tradeoff is compilation times, but:
  - It only needs to happen once
  - zk tech is improving quickly

- See also Proving and Rewarding Client Diversity to Strengthen Resilience of Blockchain Networks, in ACM Distributed Ledger Technologies

Act III - SBOM

What is an SBOM?

“An SBOM is a formal, machine-readable inventory of software components and dependencies, information about those components, and their hierarchical relationships.”

- NTIA

- Machine-readable
- List of components and dependencies
- Information about components
- Hierarchical relationships
- Standardized

Content of an SBOM


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

```
"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"
   }
```

Challenges of Producing Software Bill of Materials for Java

IEEE Security & Privacy, 2023

https://arxiv.org/abs/2303.11102

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, and Martin Wittlinger | KTH Royal Institute of Technology

Software bills of materials (SBOMs) promise to become the backbone of software supply chain hardening. We deep-dive into six tools and the SBOMs they produce for complex open source Java projects, revealing challenges regarding the accurate production and usage of SBOMs.

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.

Introduction

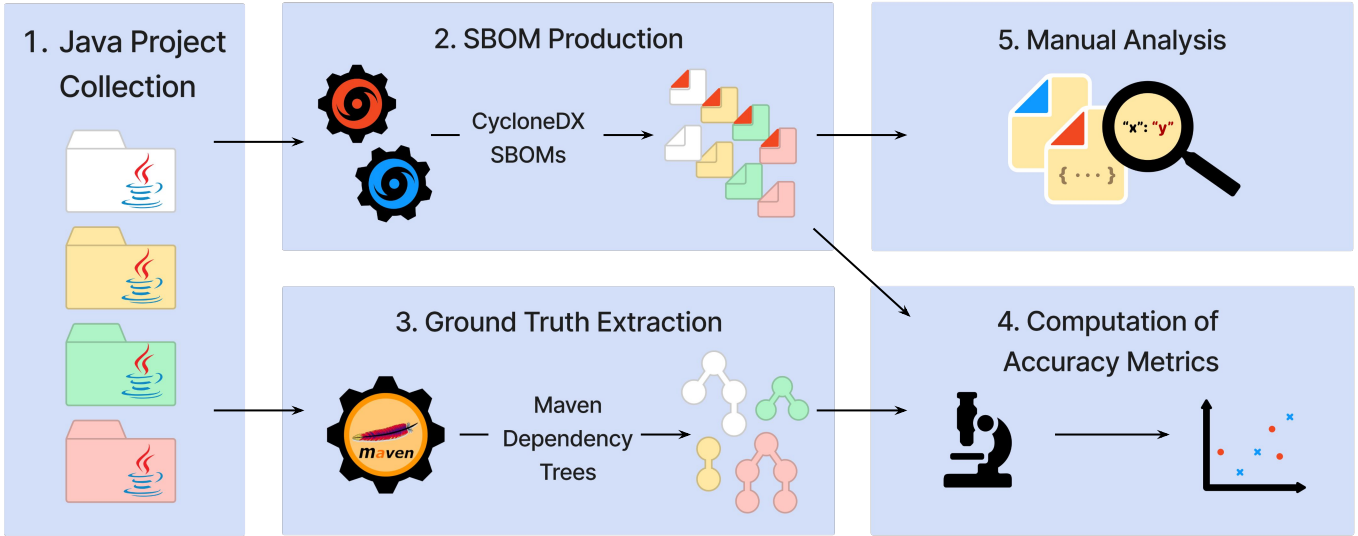
The software bill of materials (SBOM) has recently emerged as a key concept to enable principled engineering of software supply chains. This takes the well-known concept of "bill of materials" for manufacturing physical goods into the world of software development. The purpose of an SBOM is to capture relevant information about the internals of a software artifact. First and foremost, an SBOM is expected to include a complete inventory of all of 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 software component versions against vulnerability databases and reporting a warning whenever a vulnerable component is part of an application. For example, in 2021, a serious vulnerability present 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 endeavor to identify all impacted projects.[7] Had all of these Java projects published an SBOM, it would have facilitated the precise identification and remediation of vulnerable applications.

Digital Object Identifier 10.1109/MSEC.2023.3302956
Date of current version: 29 August 2023

Copublished by the IEEE Computer and Reliability Societies 1540-7993/23©2023IEEE

Protocol for studying SBOM generation



Qualitative Analysis

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

Qualitative Analysis

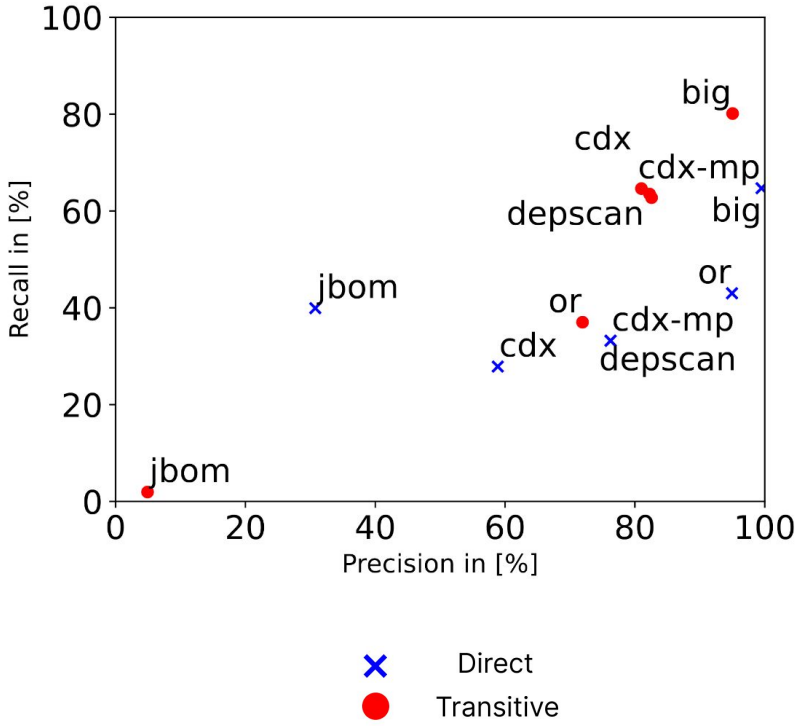
SBOM Producer	Checksums	Hierarchy	Deterministic	Scope
Build-Info-Go	3	✓	✗	✗
cdxgen				✓
cyclonedx-maven-plugin				✓
depscan				✓
jbom				✓
OpenRewrite	0	✓	✓	✓

Takeaway:

Different SBOM producers have distinct properties

Quantitative Analysis

- Compare 6 producers against *the ground truth*
- The average of 26 runs on each datapoint
- Blue crosses are direct dependencies
- Red circles are transitive dependencies



Takeaways of the “Challenges” paper

- Challenges of Producing Software Bill Of Materials for Java, In IEEE Security & Privacy, 2023.

- SBOMs are fundamental for software supply chain security and reliability

- Yet, it is surprisingly challenging to get good SBOMs, need work on the SBOM standards, and on the SBOM producers.

More material

- SBOMs at runtime (SBOM.exe, classport)
  - “NodeShield: Runtime Enforcement of Security-Enhanced SBOMs for Node.js”ACM CCS 2025
- Empirical study on SBOMs
  - Software Bill of Material on Maven Central (MSR 2025)
- Zero-knowledge SBOMs
  - zkSBOM: Privacy-Preserving SBOM Sharing with Zero-Knowledge Sets (2026)

Software Integrity

Software integrity refers to the assurance that software remains complete, unaltered, and consistent with its intended, authorized state throughout its lifecycle: development, distribution, installation, and execution.

1. **Code integrity** — Ensuring source code and compiled binaries have not been tampered with, whether by malicious actors, supply chain attacks, or unintentional corruption.
2. **Data integrity** — Ensuring the data a program processes, stores, or transmits is accurate and has not been altered without authorization.
3. **Execution integrity** — Ensuring the software behaves as designed at runtime, without injected malicious code or unauthorized modification of running processes.
4. **Supply chain integrity** — Ensuring that dependencies, libraries, and build tools used to create the software are themselves trustworthy and unmodified.

SBOM.EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java

Aman Sharma, Martin Wittlinger, Benoit Baudry, Martin Monperrus

Abstract—Software supply chain attacks have become a significant threat as software development increasingly relies on contributions from multiple, often unverified sources. The code from unverified sources does not pose a threat until it is executed. `Log4Shell` is a recent example of a supply chain attack that processed a malicious input at runtime, leading to remote code execution. It exploited the dynamic class loading facilities of Java to compromise the runtime integrity of the application. Traditional safeguards can mitigate supply chain attacks at build time, but they have limitations in mitigating runtime threats posed by dynamically loaded malicious classes. This calls for a system that can detect these malicious classes and prevent their execution at runtime. This paper introduces SBOM.EXE, a proactive system designed to safeguard Java applications against such threats. SBOM.EXE constructs a comprehensive allowlist of permissible classes based on the complete software supply chain of the application. This allowlist is enforced at runtime, blocking any unrecognized or tampered classes from executing. We assess SBOM.EXE's effectiveness by mitigating 3 critical CVEs based on the above threat. We run our tool with 3 open-source Java applications and report that our tool is compatible with real-world applications with minimal performance overhead. Our findings demonstrate that SBOM.EXE can effectively maintain runtime integrity with minimal performance impact, offering a novel approach to fortifying Java applications against dynamic classloading attacks.

Publicly-available repository - https://github.com/chains-project/sbom.exe

Index Terms—Software Supply Chain, Software Bill of Materials, Dynamic Classloading

1 Introduction

Developers reuse a lot of third-party dependencies [1], [2], [3] to build software applications. The process of building the application using the set of all dependencies is known as the software supply chain of an application. While this practice is good as it avoids reinventing the wheel [4], it is challenging for developers to keep track of the reliability, maintainability and security of their software supply chain [5]. In the latter case, it has recently been observed, with high-profile attacks, that third-party libraries can be exploited by malicious actors, leading to so-called "software supply chain attacks" [6], [7], [8].

Software supply chain attacks are a significant threat to the software security landscape as acknowledged by ENISA [9] and the White House [10]. In 2023, there were twice as many software supply chain attacks as in 2019-2022 combined [11]. A few reactive and proactive techniques have been proposed in the literature to mitigate software supply chain attacks [6]. Reactive techniques are based for example on bots [12] to update dependency regularly and tools to scan for Common Vulnerabilities and Exposures (CVE)s [13] in dependencies. However, these only act after the exploit has been discovered and reported. The main proactive technique to help prevent software supply chain attacks is to systematically create Software Bill of Materials (SBOM). In a nutshell, an SBOM is a complete list of dependencies and tools used to build a software application. SBOMs increase transparency in the software supply chain, and are good for accountability [10], but they cannot prevent attacks at runtime.

In this paper, we propose a novel technique to mitigate a class of software supply chain attacks based on code injection in third-party dependencies at runtime [7]. This type of attack received major attention when the high-profile attack `Log4Shell` (CVE-2021-44228 [14]) was released, demonstrating that the dependency `Log4j`, which is part of millions of Java applications' software supply chain [15], was vulnerable. The attack was possible because the dependency had a vulnerability enabling remote code execution when processing a malicious input. More importantly, the attack was entirely based on dynamic class loading facility that exists in Java.

Conceptually, if an attacker knows that a dependency uses Java dynamic features, then they can exploit these features to compromise any dependent application. The solution is obviously not to forbid these dynamic features, because they are used in virtually all mission and business critical applications, incl. all Spring Java web apps. Also, neither SBOM, reproducible builds, code signing, nor version pinning would prevent these attacks as the malicious code in these attacks appears only while the application is running.

We propose SBOM.EXE, a system that ensures the integrity of the Java runtime, by monitoring the software supply chain of an application at runtime, in order to detect and prevent the execution of injected malicious code. SBOM.EXE works by building a comprehensive allowlist of

- *A. Sharma and M. Monperrus are with the KTH Royal Institute of Technology, Stockholm, Sweden*
  Email: {amansha, monperrus}@kth.se
- *M. Wittlinger is with the HDI Group, Cologne, Germany*
  Email: martin.wittlinger@hdi.de
- *B. Baudry is with the Université de Montréal, Montréal, Canada*
  Email: benoit.baudry@umontreal.ca

SBOM.EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java

Technical report 2407.00246, arXiv, 2024.

PhD Thesis Aman Sharma

https://github.com/ASSERT-KTH/sbom.exe/

open / source / insights

Search for open source packages, advisories and projects | Maven

Maven artifact

org.apache.logging.log4j:log4j-api

2.24.3

Overview | Dependencies | Dependents | Compare | Versions

Direct

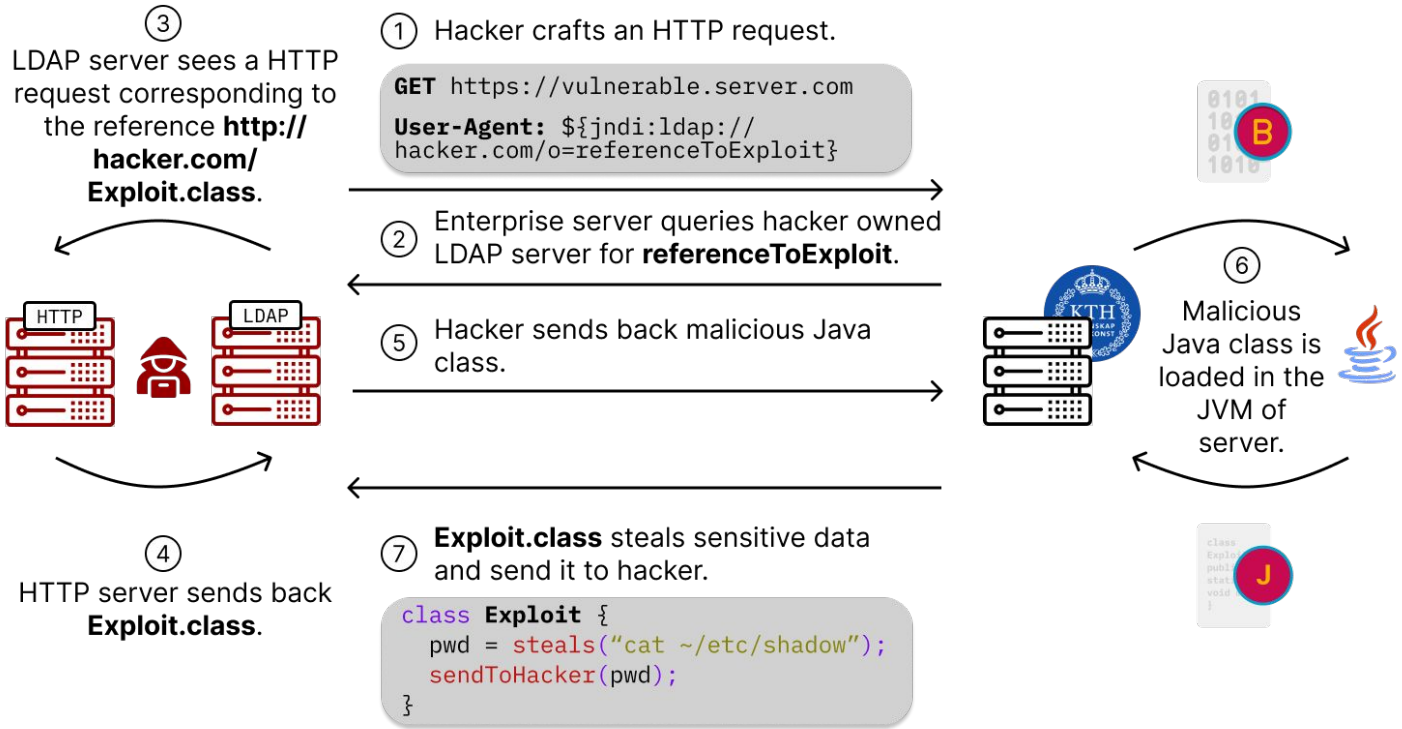
Indirect

Total unique dependents for org.apache.logging.log4j:log4j-api: 15378

Note: Due to the large number of dependents, we show only a sample in the list below.

Artifact	Version	Relation
com.bbossgroups.activiti:activiti-bpmn-converter	5.13.11	Direct
com.bbossgroups.activiti:activiti-bpmn-layout	5.13.11	Direct
com.bbossgroups.activiti:activiti-engine	5.13.11	Direct
com.bbossgroups.rpc:bboss-rpc	6.2.7	Direct
com.bbossgroups.security:bboss-security	6.2.7	Direct

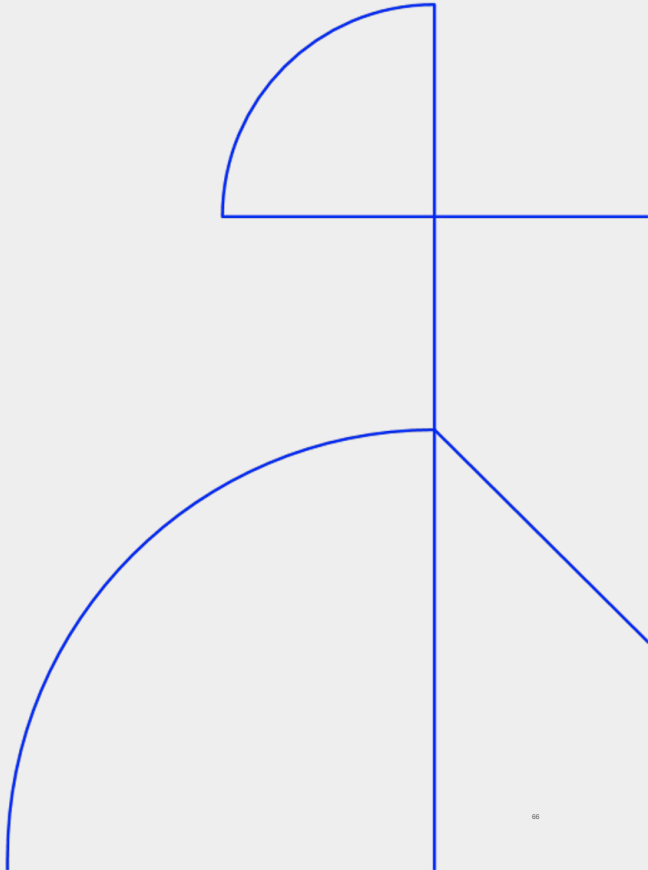
Log4Shell – a software supply chain attack at runtime



Problem statement: libraries can trigger download or generation of malicious code.

Solution: Create an allowlist of dependencies and their classes and only load those classes

That’s SBOM.exe



Act IV - Software Integrity

Step 1: Indexing

Problem: how to index built-in classes?

Solution: we scan all classes using classgraph [20].

Problem: what about app classes and dependencies?

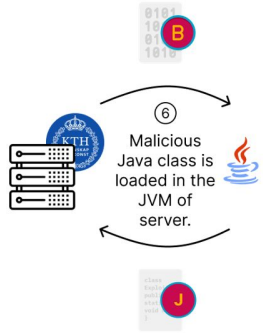
Solution: we use the Software Bill of Materials

Problem: and runtime generated code?

Solution: if we execute the code, we can capture them. We run run tests.

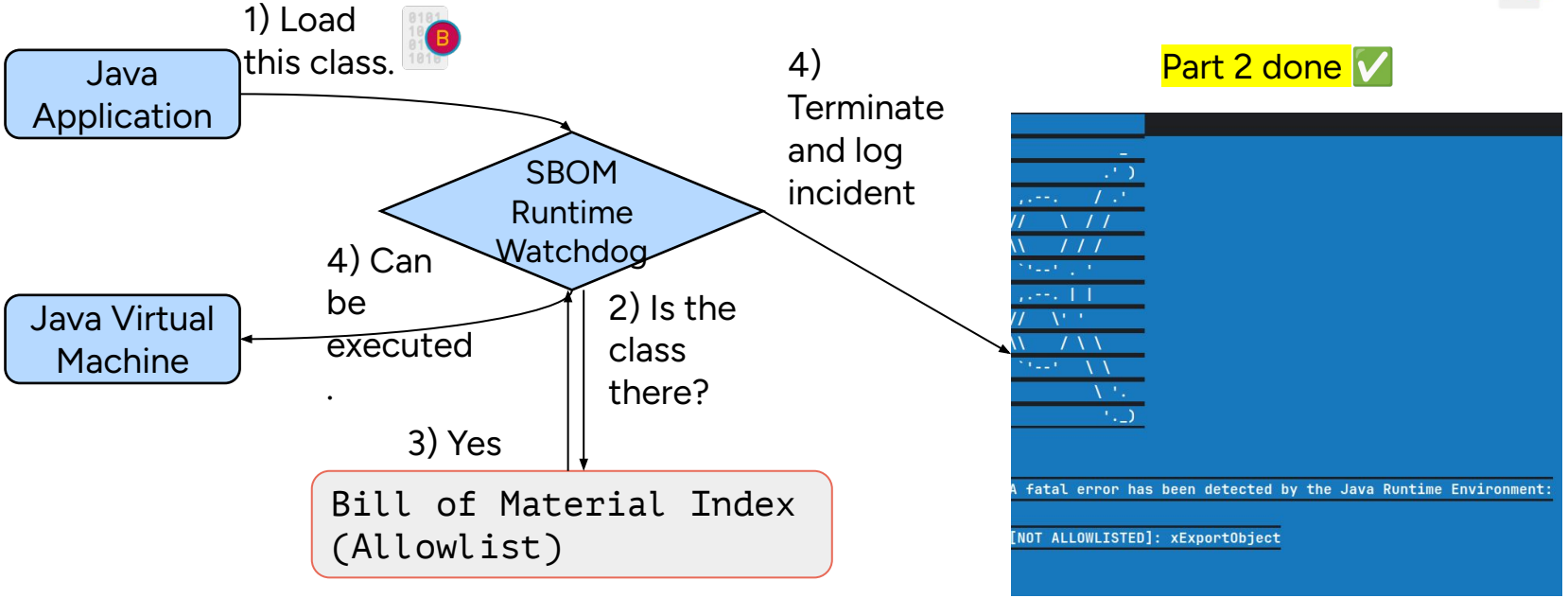
[20] L. Hutchison, Classgraph, GitHub.com, 2024

Step 2: Enforcement



Problem: Java class is simply loaded without any integrity.

Solution: We intercept loading and then verify it.

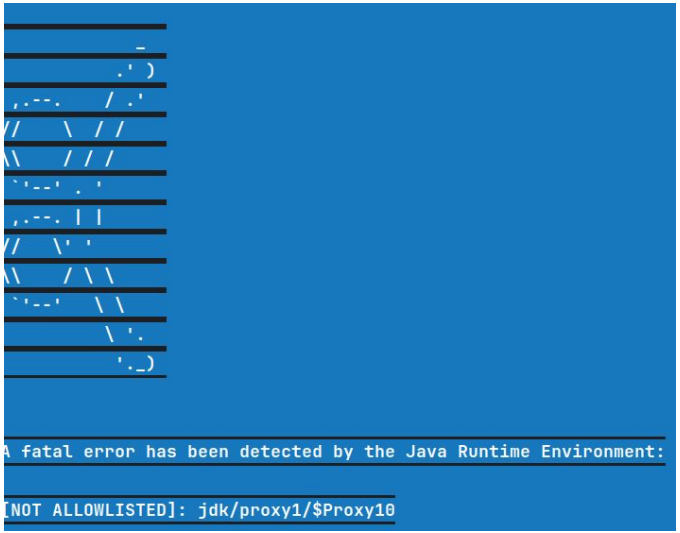


Canary run

Okay, we seem to be done. Let’s see what happened initially.

Problem 1: There seems to be false-positives. This class was in the allowlist.

Problem 2: There seems to be non-determinism in runtime generated code.



Solution: Java Bytecode Canonicalization

- Classnames could change across different executions.
- The type references change.
- The order of method is not fixed.
- SBOM.exe canonicalize all of these.

```
- public class $Proxy10 {
+ public class $Proxy7 {
-     private static $Proxy10.x;
+     private static $Proxy7.x;
-    m1 () {}
+    m3 () {}
-    m3 () {}
+    m1 () {}
}
```

Novel Concepts Summarised

Problem: what to index?

Solution: 3 indexers for built-in classes source code, dependencies, and dynamic code.

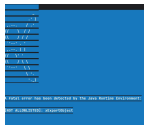
Problem: how to load class with verification

Solution: SBOM Runtime Watchdog is a novel tool to intercept Java classloading and verify integrity of each Java class.

Problem: non-determinism of Java bytecode.

Solution: Bytecode Canonicalization.

Problem: Attackers compromise production systems

Solution: 

Evaluation of SBOM.EXE

Aman Sharma | amansha@kth.se

RQ1: What is the scale of BOMI for all the study subjects?

- Methodology: Count classes in BOMI for all study subjects.

- Key findings:
  - BOMI-Environment contains around 25,000 classes, varying with JDK vendors and versions.
  - BOMI-SupplyChain in the xx,0000, strongly depends on number of classes in dependencies.
  - BOMI-Runtime constitutes few number of classes, but they are extremely important to detect in order to prevent attacks and to avoid false positives.

RQ2: To what extent can SBOM.EXE mitigate high-profile attacks in Java?

- Methodology:
  - CVE-2021-44228 (Log4Shell), CVE-2021-42392 (H2 database console), and CVE-2022-33980 (Apache Commons Configuration
  - Replicate in a proof-of-concept and then run them with our BOMI and SBOM Runtime Watchdog attached.

- Key findings:
  - It successfully prevent execution of the malicious class loaded at runtime in all 3 CVEs.
  - **CVE-2021-44228 – mitigated**
  - **CVE-2021-42392 – mitigated**
  - **CVE-2022-33980 – mitiagted**

RQ2: To what extent can SBOM.EXE mitigate high-profile attacks in Java?

- Methodology: Replicate CVE-2021-44228 (Log4Shell), CVE-2021-42392 (H2 database console), and CVE-2022-33980 (Apache Commons Configuration) in a proof-of-concept and then run them with our BOMI and SBOM Runtime Watchdog attached.

Takeaway: Mitigated all 3 CVEs successfully.

  - **CVE-2021-42392 – mitigated**
  - **CVE-2022-33980 – mitiagted**

RQ3: Is SBOM.EXE compatible with real-world applications?

- Methodology: Create BOMI for real-world projects – PDFBox, TTorrent, and GraphHopper – and then run these projects with SBOM Runtime Watchdog attached.

- Key findings:
  - SBOM.exe is fully compatible with PDFBox and TTorrent.
    - PDFBox:
      - Additional test for PDFBox Debugger
    - TTorrent:
      - Additional e2e test to simulate torrent downloading
  - SBOM.exe is partially compatible with GraphHopper as one of the class names were randomly generated, despite canonicalization.

RQ3: Is SBOM.EXE compatible with real-world applications?

- Methodology: Create BOMI for real-world projects – PDFBox, TTorrent, and GraphHopper – and then run these projects with SBOM Runtime Watchdog attached.

Takeaway: Compatible with real-world projects.

  - Additional test to simulate torrent downloading
  - Update java bytecode for it to be compatible with Java 21
- SBOM.exe is partially compatible with GraphHopper as one of the class names were randomly generated.

Takeaways SBOM.exe

- SBOM.exe mitigates three high-profile CVEs.
- SBOM.exe proposes a novel bytecode canonincalization algorithm which eliminates non-determinism in dynamic classes.
- SBOM.exe work wells on real world projects.

Finale - KUTE

UPPERCASE IS ALL YOU NEED
SIGBOVIK, 2025.

The KUTE Countermeasures

- Know your dependencies (software bill of materials, aka SBOM)
  - https://github.com/CycloneDX
  - https://github.com/chains-project/dirty-waters
- Update your dependencies (regularly and automatically)
  - Dependabot, Renovate, DepFu
- Track updates, vulnerabilities, changes and humans
  - npm audit / govulncheck or $$$$ solutions
  - https://github.com/lightbend-labs/jardiff
- Ensure integrity at build and runtime
  - Reproducible builds
  - https://github.com/step-security/harden-runner
    https://github.com/chains-project/sbom.exe

More Cool Supply Chain Tech

- dependency sandboxing (NodeShield, GoLeash)
- tamper-proof audit trails & build attestations (Sigstore, npm provenance)
- transparency logs (Go package registry)
- lockfiles (maven-lockfiles, EMSE)
- zero-knowledge exploits

Conclusion

- Software supply chain attacks are one of the top cybersecurity threats
- Very active research area, KTH leads CHAINS
- Today: trusting trust attacks, verifiable provenance, runtime integrity
- Register to the Chains mailing list