

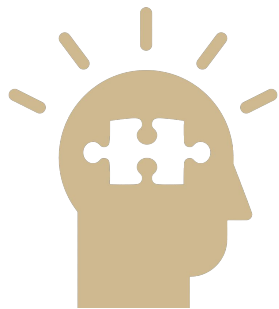
Full-Stack Software Supply Chain Security: From Keyless Signing to Supply Chain Intelligence

Helsinki-Aalto Institute for Cybersecurity (HAIC)

October 8th, 2026

Santiago Torres-Arias

Things we care about



Understand, model and visualize
how software is made

README.md

Catalog of Supply Chain Compromises

This repository contains links to articles of software supply chain compromises. The goal is not to catalog every known supply chain attack, but rather to capture many examples of different kinds of attack, so that we can better understand the patterns and develop best practices and tools.

For definitions of each compromise type, please check out our [compromise definitions page](#)

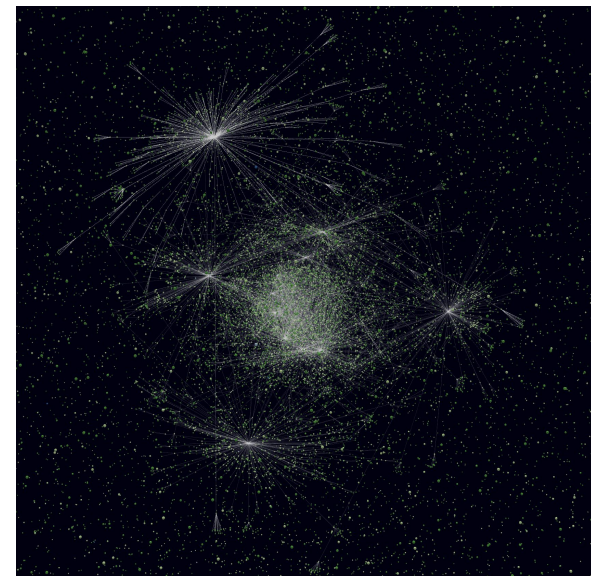
We welcome additions to this catalog by [filing an issue](#) or [github pull request](#)

Contents of this repo and proposed additions are not a statement or opinion on the security stance and/or practices of a given project, of open source, or the community. These articles and stories annotate the communities dedication to rapid response, evolving security practices, transparent disclosure, and enforcement of one of open sources founding principles, "Linux's Law".

When submitting an addition, please review the [definitions](#) page to ensure the Type of Compromise on the details of the incidents as well as the Catalog itself are consistent. If a definition doesn't exist or a new type of compromise needs added, please include that as well.

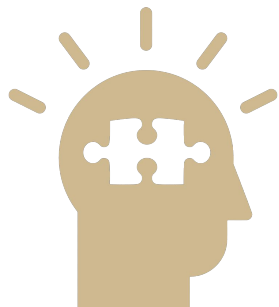
Name	Year	Type of compromise	Link
Docker Hub malicious containers	2022	Publishing Infrastructure	1
Chat100 live chat trojan	2022	Publishing Infrastructure	1
Dropbox GitHub compromise	2022	Attack Chaining	1
Intel Alder Lake BIOS leak	2022	Source Code	1
PEAR PHP Package Manager compromise	2022	Dev Tooling	1
npm Library 'node-ipc' Sabotaged with npm Library 'peacenotwar' in Protest by their Maintainer	2022	Malicious Maintainer	1
npm Libraries 'colors' and 'fast-deep-equal' Compromised by their Maintainer	2022	Malicious Maintainer	1
GCP Golang Buildpack	2022	Source Code	1

Cloud Native Computing Foundation





Things we care about



Understand, model and visualize
how software is made

Ident
(SW, AI, \

```
>> And as a result, when I run the git command `git checkout  
>> refs/tags/v233`, I am tricked into getting v232 instead which contains a  
>> vulnerability.  
>  
> Until there, it's exactly the topic of the presentation linked by  
> Nichood
```

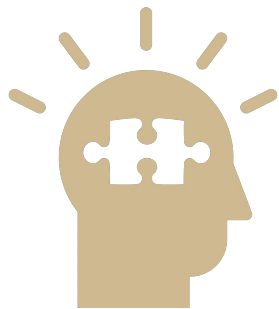
So I was under the impression that git tags encode the tagname in the actual blob, and I didn't see how that attack (rooted in the basic nature of a branch as a lightweight, mutable, *pushable* pointer to a commit) was supposed to work unless of course it was talking about a lightweight tag (which is not really meant for public/permanent use)...

Having actually tested this out, I find myself quite bewildered.

Because, git *does* encode the tagname in the blob, like I thought. And... you *can* simply copy .git/refs/tags/tagname to create a fake tag, and then you see something quite bewildering:



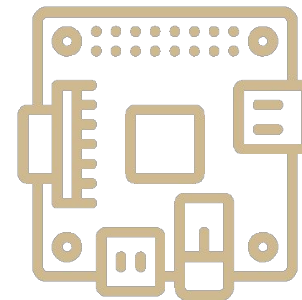
Things we care about



Understand, model and visualize
how software is made



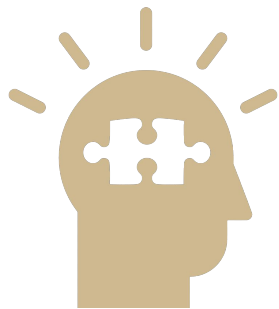
Identify threats and attacks on
various supply chains
(SW, AI, VLSI, Additive Manufacturing)



Build and evaluate systems to
enforce these security
expectations



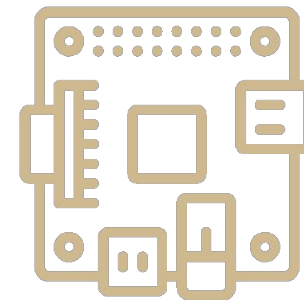
Things we care about



Understand, model and visualize
how software is made



Identify threats and attacks on
various supply chains
(SW, AI, VLSI, Additive Manufacturing)



Build and evaluate systems to
enforce these security
expectations



Background:

Software supply chains & attacks

Background: Software Supply Chains



Development

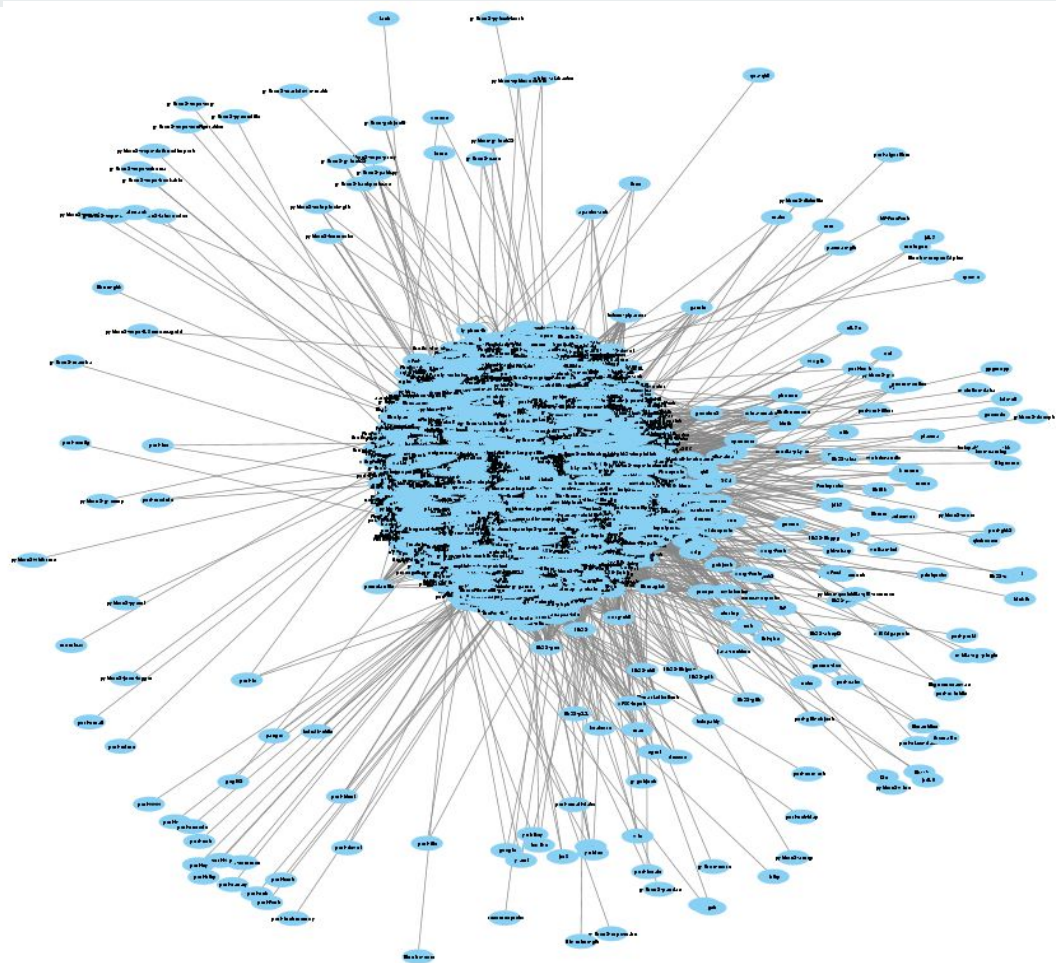
All operations, processes, tools, institutions
and human factors involved in creating a
piece of software.



Production



The arch Linux package repository supply chain.





FEDERAL BUREAU OF INVESTIGATION
PROTECTIVE DIVISION
UPDATE

[Home](#)

[Tech](#)

FT
vu

By: TI

Log4j

found
popul
to millions of

SEC

Supply Chain Attack: Major Linux Distribution ‘Mini Shai-Hulud’ malware compromises hundreds of open-source packages in sprawling supply-chain attack

by the FTC

Urgent data c The campaign hit major registries and hid behind legitimate-looking release signatures, showing how attackers can weaponize the software update process itself.

BY GREG OTTO • MAY 12, 2026



 Listen to this article 7:25 [Learn more.](#)



Regulators are pushing for stronger controls and better risk assessment.

MAY 12, 2021

Executive Order on Improving the Nation's Cybersecurity



BRIEFING ROOM

PRESIDENTIAL ACTIONS

By the authority vested in me as President by the Constitution and the laws of the United States of America, it is hereby ordered as follows:

Section 1. Policy. The United States faces persistent and increasingly sophisticated malicious cyber campaigns that threaten the public sector, the private sector, and ultimately the American people's security and privacy. The Federal Government must improve its efforts to identify, deter, protect against, detect, and respond to these actions and actors. The Federal Government must also carefully examine what occurred during any major cyber incident and apply lessons learned. But cybersecurity requires more

SBOM Requirements in the EU's CRA (Cyber Resilience Act)

September 10, 2024 · 20 min read · Tony Turner

SBOM

Cyber Resilience Act

EU Regulations



Table of Contents

- CRA Background
- Breaking Down the CRA's SBOM-Related Provisions
- CRA SBOM Formats and Data Fields
- What's Next for the CRA

The role of regulation in driving global adoption for [software bill of materials \(SBOM\)](#) may have started with U.S. Executive Order 14028, but make no mistake; this is not a U.S.-only initiative. We have seen regulations from the Payment Card Industry (PCI DSS 4.0), signals from the Australian government, and other global initiatives along with U.S. requirements (like the aforementioned Executive Order along with guidelines from the [Food and Drug Administration](#)).

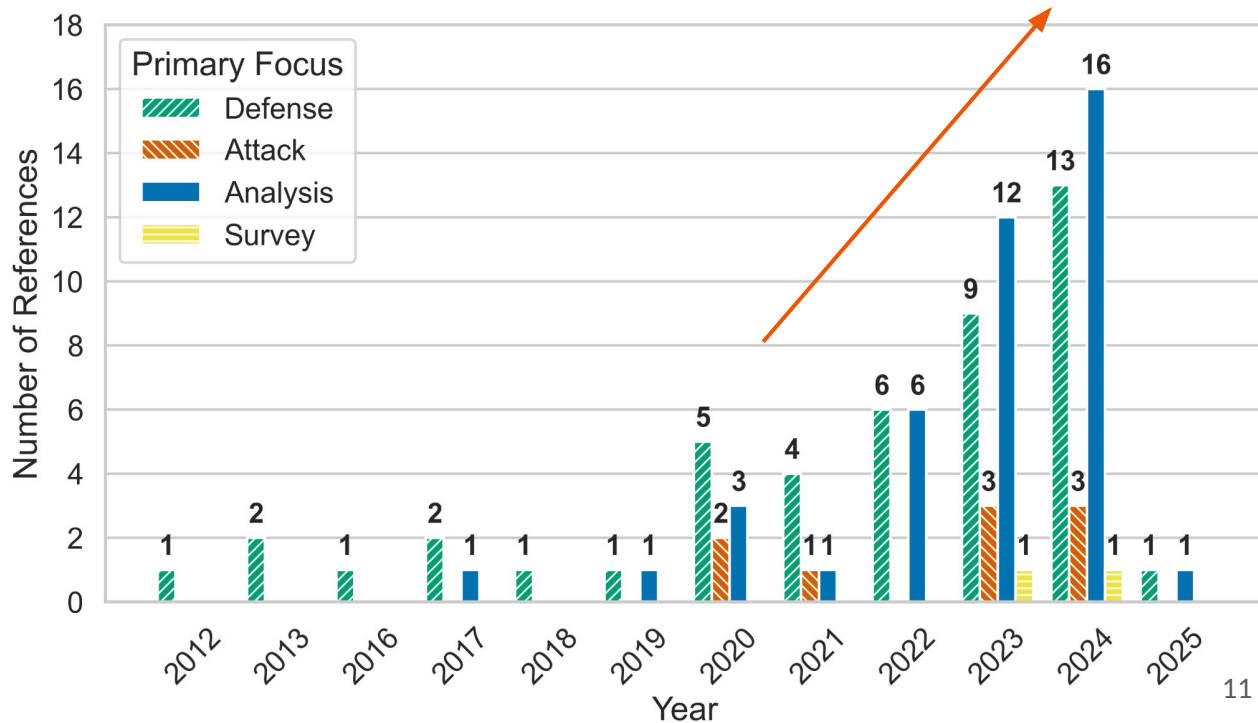
The Cyber Resilience Act (CRA) — a set of rules elevating security standards for digital products in the European Union — is another significant security regulation with a prominent SBOM requirement.

The CRA applies to manufacturers of digital products selling in the EU and any entity such as distributors or importers that market a digital product under their brand. It requires them to share a top-level SBOM with market surveillance authorities as part of the technical documentation provided. Open source software is specifically exempt from these

Academia Efforts

This derived research and innovation in the field of SSC security.

We analyzed 98 papers from 2010 (from [2]).

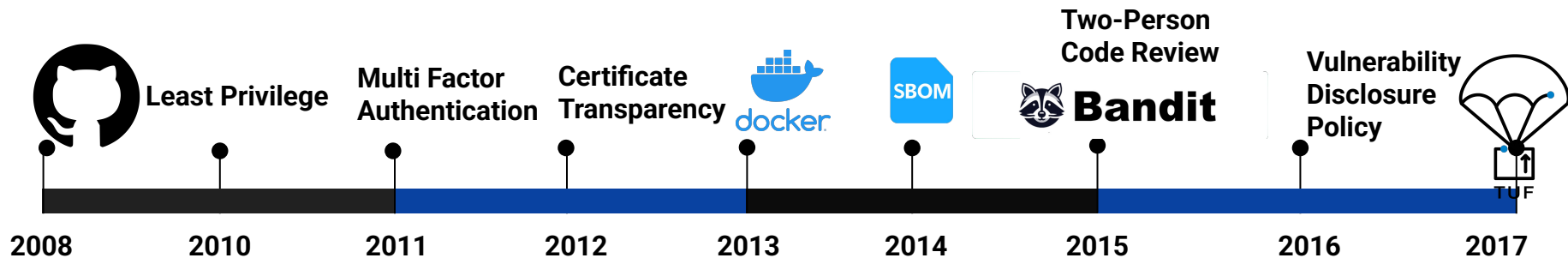
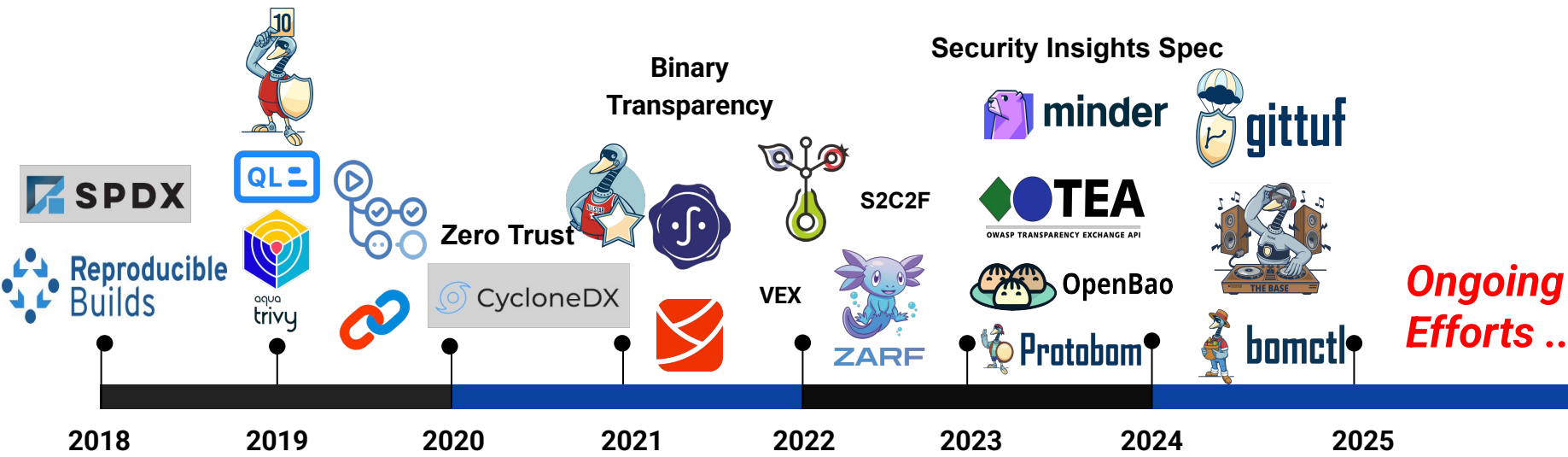




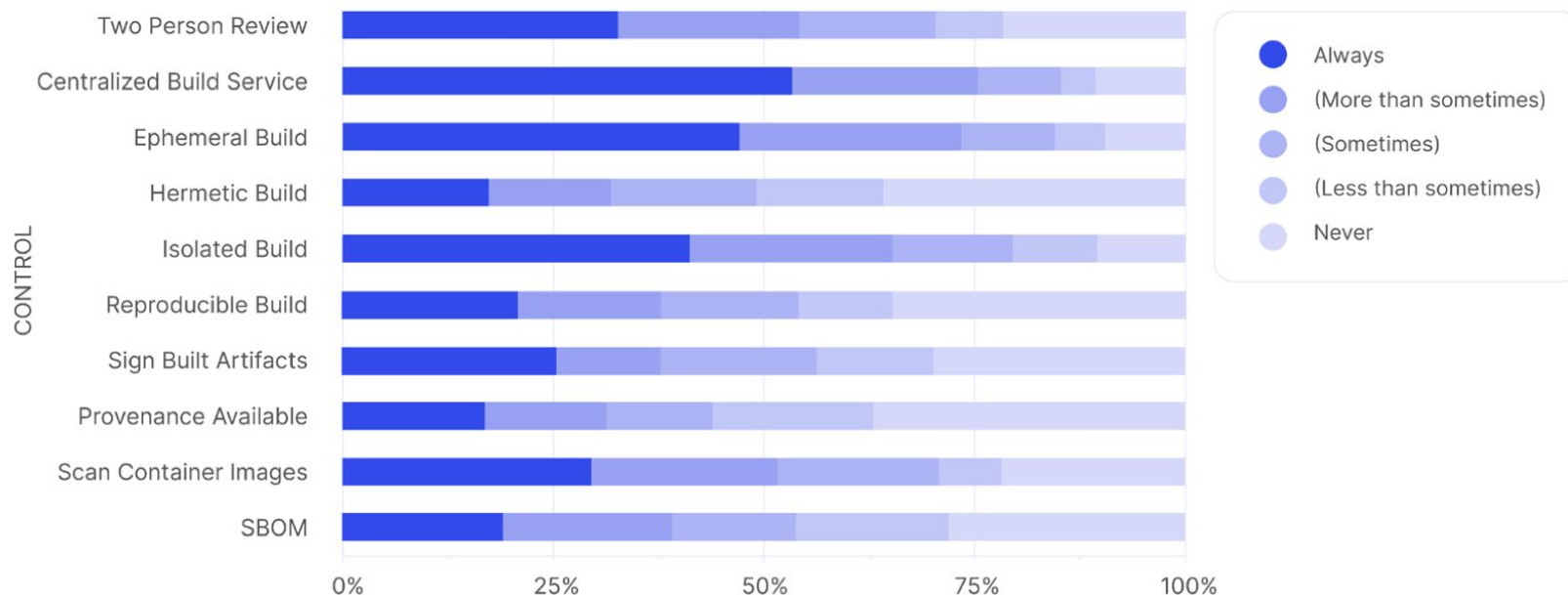
Industry & Open-Source Initiatives

- Proposed multiple frameworks and practices to audit the software development lifecycle (SDLC) and validate open-source and proprietary software.



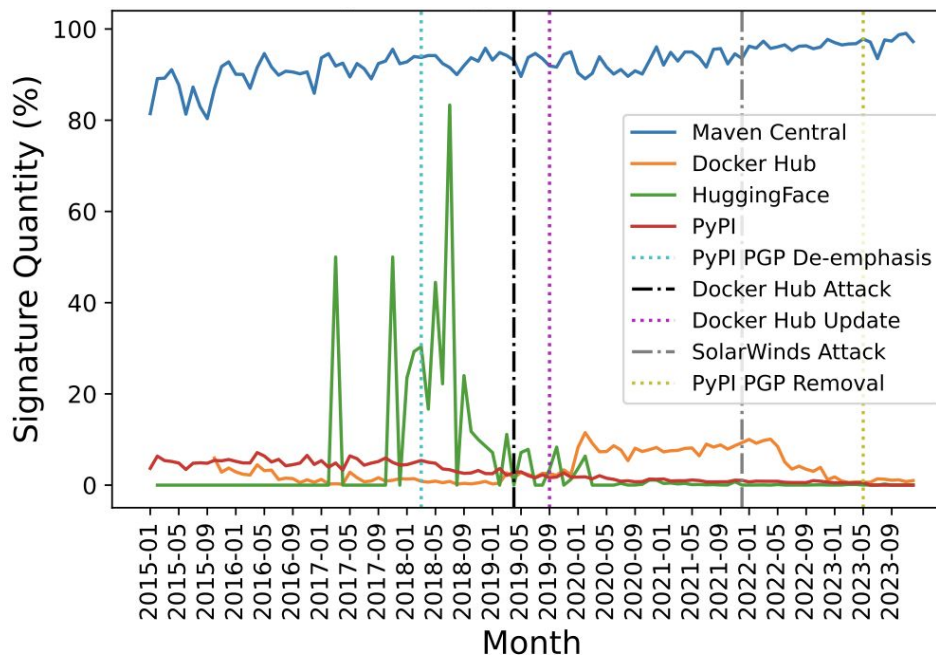


Supply Chain Systems Are Not Widely Adopted

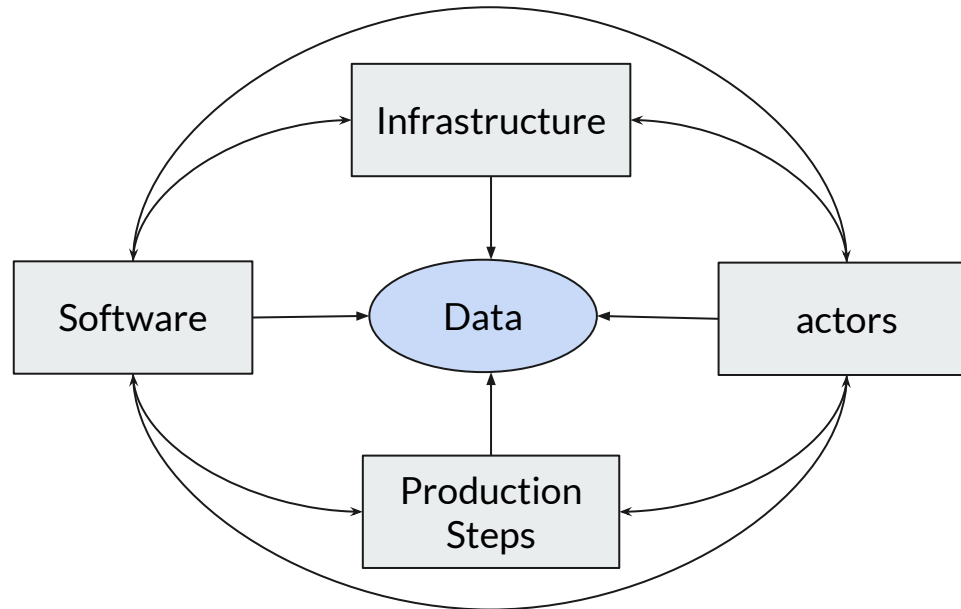


Quantity of Signed Artifacts in Package Registries

- ❖ 8 years of software signing analysis (from [3]).
- ❖ Software signing are barely adopted in some ecosystems.



We need effective tool-design from the ground up





AStRA: Modelling SSCs as Causal Graphs

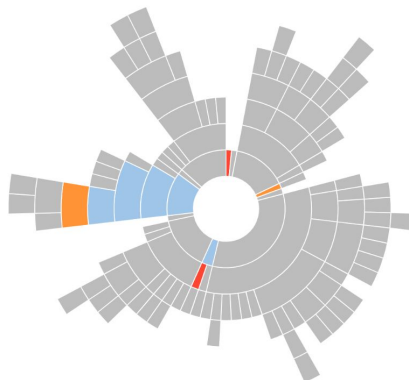
Can an abstract supply chain model help guide design?

Models of The Supply Chain

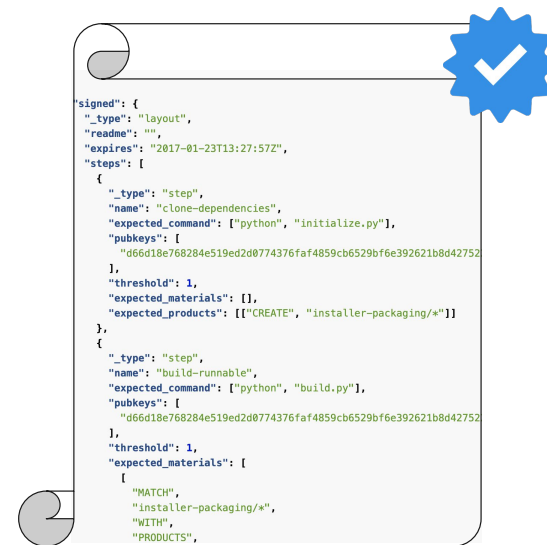
(All models are wrong, but some are useful)

```
root@46ca1c89e977:/astra-tests# apt-rdepends nginx
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
nginx
  Depends: iproute2
  Depends: libc6 (>= 2.34)
  Depends: libcrypt1 (>= 1:4.1.0)
  Depends: libpcre2-8-0 (>= 10.22)
  Depends: libssl3 (>= 3.0.0)
  Depends: nginx-common (>= 1.22.1-9+deb12u8)
  Depends: zlib1g (>= 1:1.1.4)
iproute2
  Depends: debconf (>= 0.5)
  Depends: debconf-2.0
  Depends: libbbpf1 (>= 1:0.6.0)
  Depends: libbsd0 (>= 0.0)
  Depends: libc6 (>= 2.34)
  Depends: libcap2 (>= 1:2.10)
  Depends: libcap2-bin
  Depends: libdb5.3
  Depends: libelf1 (>= 0.131)
  Depends: libmnl0 (>= 1.0.3-4~)
  Depends: libselinux1 (>= 3.1~)
  Depends: libtirpc3 (>= 1.0.2)
  Depends: libxtables12 (>= 1.6.0+snapshot20161117)
debconf
```

Adjacency
(e.g., dependency graphs)



Claim-based
(e.g., knowledge graphs)



flow-based
(e.g., In-toto layout)

Adjacency-based

Strengths:

- Useful for ecosystem modelling

Weaknesses:

- Speculative
- They can not know which alternative is used

Examples:

Cappos et al., CCS '08, Zimmermann et al., USENIX Security '19, Duan et al., NDSS '21, Zahan et al., ICSE-SEIP '22, Wang et al., USENIX Security '23, Derr et al., CCS '17, Neupane et al., USENIX Security '23

```
root@46ca1c89e977:/astra-tests# apt-rdepends nginx
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
nginx
  Depends: iproute2
  Depends: libc6 (>= 2.34)
  Depends: libcrypt1 (>= 1:4.1.0)
  Depends: libpcre2-8-0 (>= 10.22)
  Depends: libssl1.3 (>= 3.0.0)
  Depends: nginx-common (>= 1.22.1-9+deb12u8)
  Depends: zlib1g (>= 1:1.1.4)
iproute2
  Depends: debconf (>= 0.5)
  Depends: debconf-2.0
  Depends: libbpf1 (>= 1:0.6.0)
  Depends: libbsd0 (>= 0.0)
  Depends: libc6 (>= 2.34)
  Depends: libcap2 (>= 1:2.10)
  Depends: libcap2-bin
  Depends: libdb5.3
  Depends: libelf1 (>= 0.131)
  Depends: libmnl0 (>= 1.0.3-4~)
  Depends: libselinux1 (>= 3.1~)
  Depends: libtirpc3 (>= 1.0.2)
  Depends: libxtables12 (>= 1.6.0+snapshot20161117)
debconf
```

Claim-based

They describe a series of facts about software.

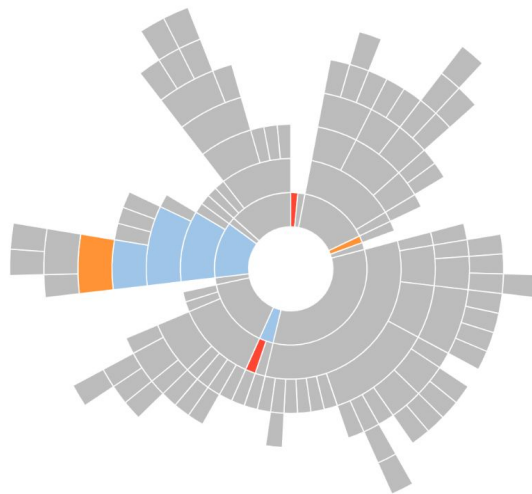
Examples: GUAC

Strengths:

- Useful for inventory, composition queries.

Weaknesses:

- Claims do not directly encode execution causality
- Hard to reason about root cause and risk propagation





Flow-based

Strengths :

- Helps identify unauthorized modifications

Weaknesses:

- No notion of active vs passive materials.

Examples:

Torres-Arias et al., USENIX Security '19,

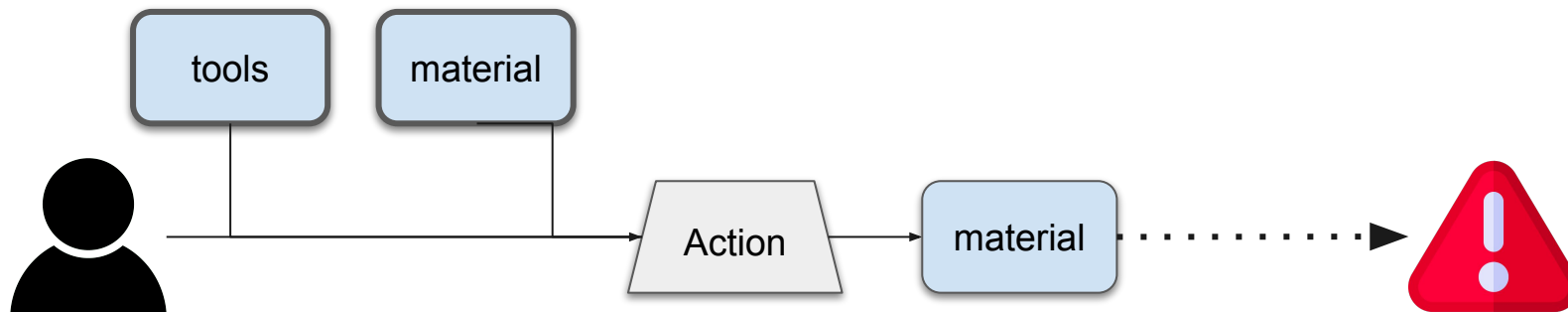
Nikitin et al., USENIX Security '17,



```
'signed': {
  "_type": "layout",
  "readme": "",
  "expires": "2017-01-23T13:27:57Z",
  "steps": [
    {
      "_type": "step",
      "name": "clone-dependencies",
      "expected_command": ["python", "initialize.py"],
      "pubkeys": [
        "d66d18e768284e519ed2d0774376faf4859cb6529bf6e392621b8d42752"
      ],
      "threshold": 1,
      "expected_materials": [],
      "expected_products": [{"CREATE", "installer-packaging/*"}]
    },
    {
      "_type": "step",
      "name": "build-runnable",
      "expected_command": ["python", "build.py"],
      "pubkeys": [
        "d66d18e768284e519ed2d0774376faf4859cb6529bf6e392621b8d42752"
      ],
      "threshold": 1,
      "expected_materials": [
        [
          "MATCH",
          "installer-packaging/*",
          "WITH",
          "PRODUCTS",
```

We need a model that:

- ❑ Capture all SSC elements, passive+active.
- ❑ Capture causality :
 - ❑ Needed for root cause analysis
 - ❑ Facilitates post-mortem analysis
 - ❑ Allows transitive risk reasoning



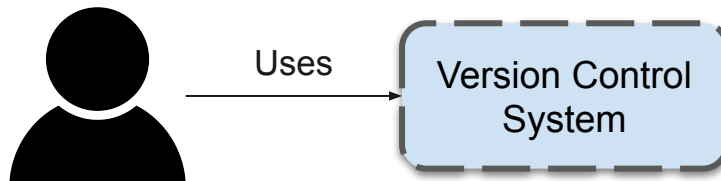
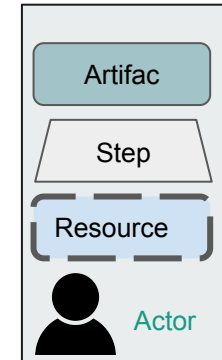
A short horizontal bar with a gold segment on the left and a black segment on the right.

AStRA Nodes

- Artifacts: What was produced or consumed? Actual software, Metadata.
- Steps: How did it happen? Build, test, commits ...
- Resources: Where did it happen? Git, Github actions, OS ...
- **Actors**: Who was involved? Developers, organizations, bots ...
- Topology: When do all nodes relate?

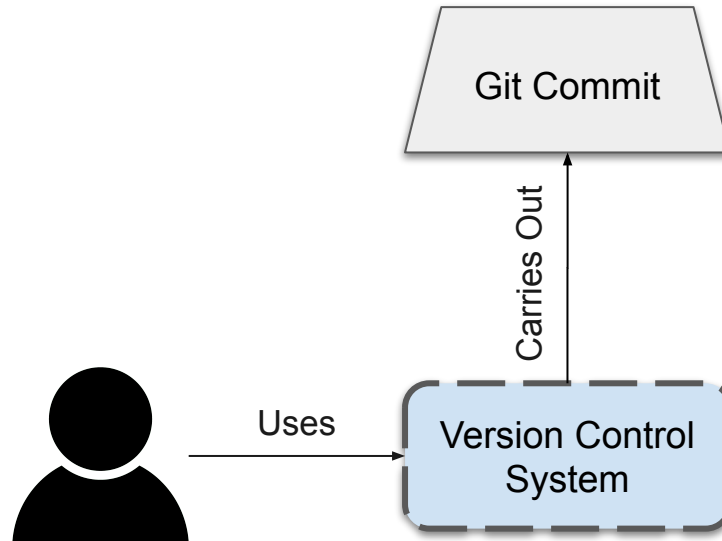
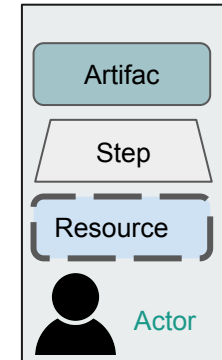
AStRA Relationships

- AStRA captures three types of causal relationships between supply chain elements:
- Actors, identified by credentials, use resources;



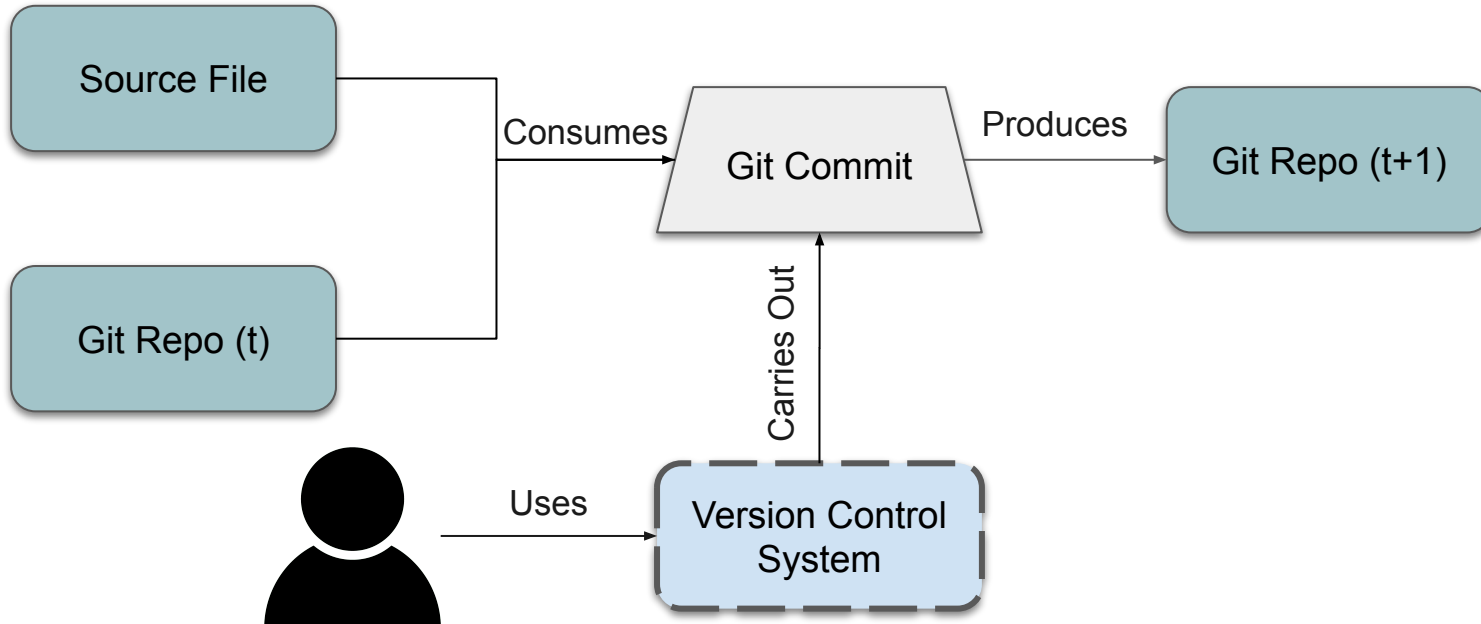
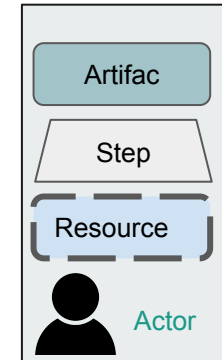
AStRA Relationships

→ Resources carry out steps;

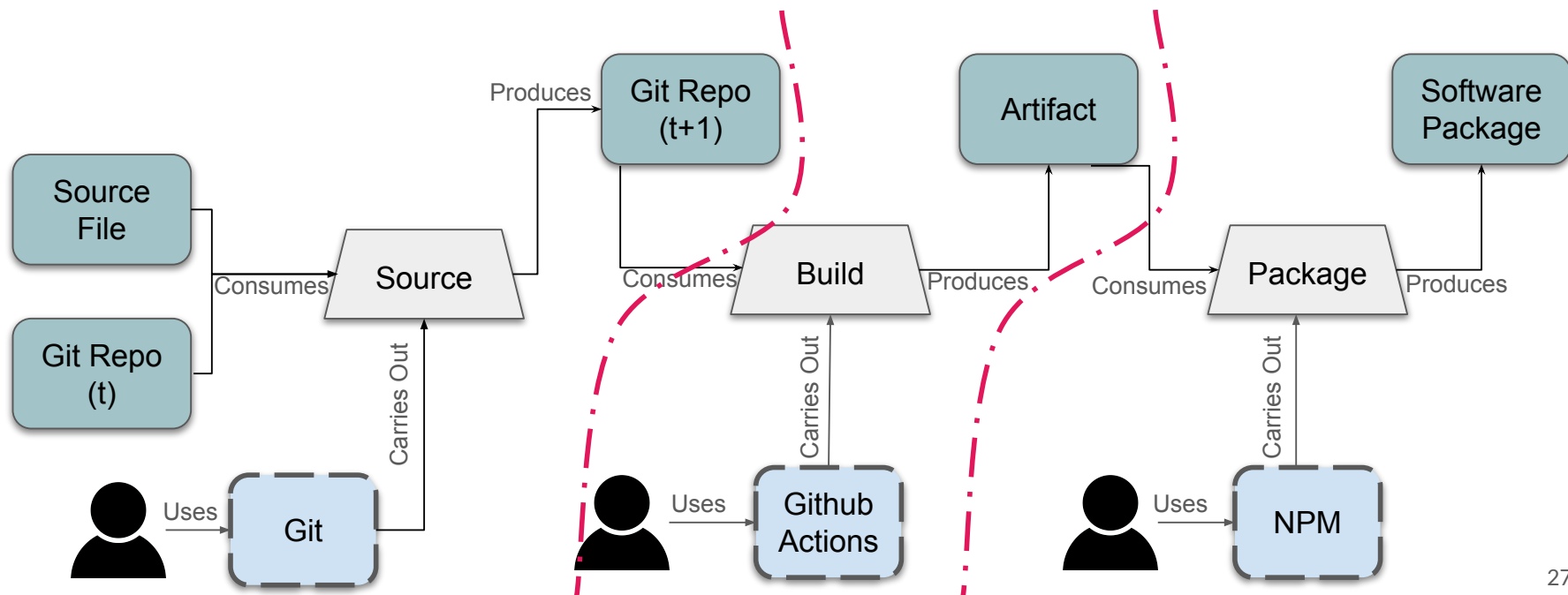


AStRA Relationships

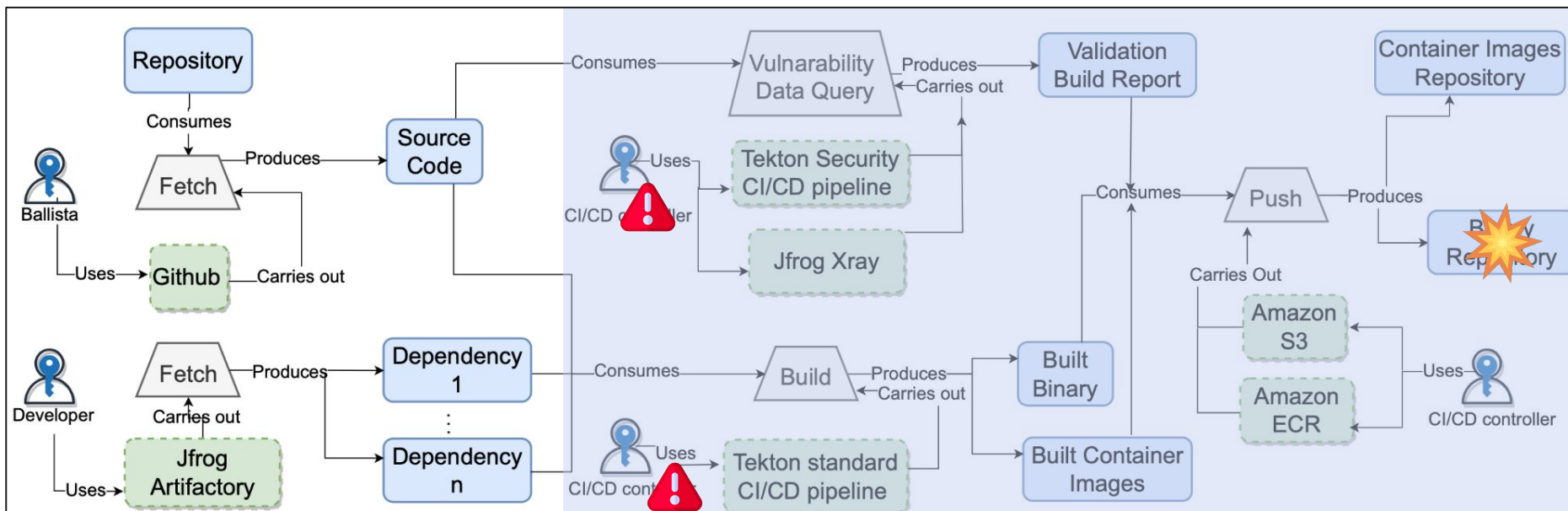
→ Steps consume and produce artifacts;



Multi-Step AStRA Topology



Case Study: SolarWinds SSC



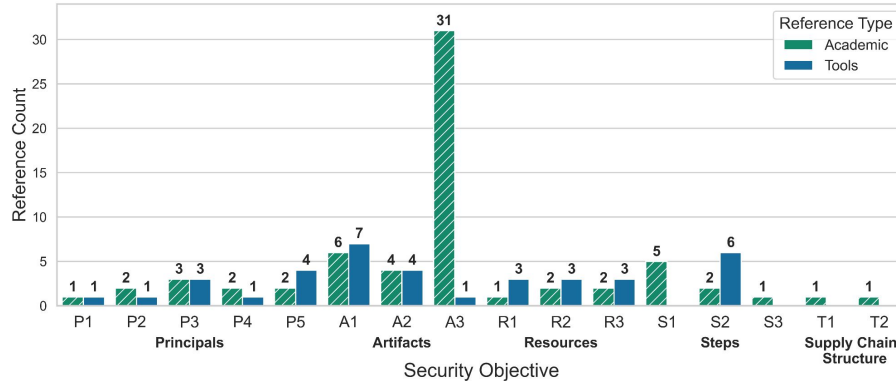
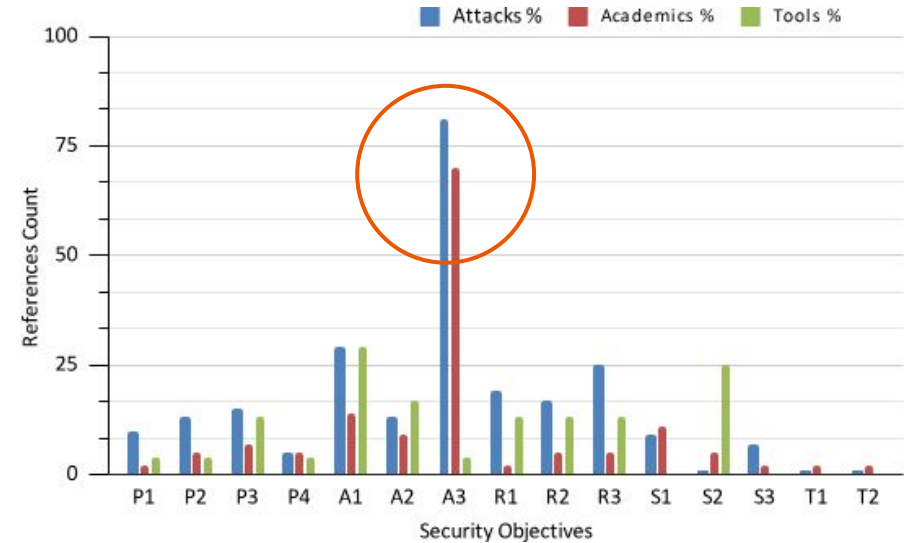
Analyzing 76 SSC defenses

- There is no single tool solving all SSC attacks
- Tools have various deployment requirements
- AStRA allows practitioners to identify tool combinations and deployment expectations

AStRA Element	Supply Chain Specific?	Academic Examples	Industry Examples	Security Objective	Defense Pattern					
					Transparency	Integrity	Graceful degra	Redundancy	Limitation	Correctness
Principals				P1 P2 P3 P4						
ephemeral credentials		[121]		●					●	
secret scanning		[200]		●		●				
credential transparency		[121]	[1]	●	●	●				
role separation		[92]	[62, 126]	●		●	●		●	
multi-factor authentication			[195]	●		●			●	
separation of privilege			[38]	●				●		
threshold authorization		[121, 168]		●			●	●		
Artifacts				A1 A2 A3						
hashing		[175]		●		●				
artifact signing		[121]	[60, 61]	●	●	●				
metadata signing	✓	[88]	[47]	●	●	●				
artifact transparency	✓	[113, 121]	[133, 150]	●	●	●				
ledger-based systems		[18]		●	●	●				
static analysis		[77, 78, 100]	[162]	●						●
patch propagation	✓	[50, 106, 107]		△		●				●
dynamic analysis		[75, 136, 174]		△		●				●
ML-based analysis		[84, 85, 147]		△		●				●
Resources				R1 R2 R3						
mandatory access control			[38, 172]	●	●	●		●		
filtering		[14]	[52]	●	●	●			●	
virtualization		[69]	[13, 173]	●	●	●			●	
Steps				S1 S2 S3						
deterministic steps	✓	[41, 119, 191]		●						●
step transparency	✓	[168]	[24, 81]	●	●	●	●			
step consensus	✓	[54, 123]	[26]	●	●	●	●			
static analysis		[112, 188]		●						●
ML-based analysis		[73]		●						●
Supply Chain Structure				T1 T2						
supply chain layout integrity	✓	[168]		●		●				●
supply chain reproducibility	✓	[191]		●						●

Are we focusing on the right things?

Distribution of Academic References and Tools Across Software Supply Chain Security Objectives



Astra's Impact

NIST Special Publication 800
NIST SP 800-204D

Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines

Ramaswamy Chandramouli
Frederick Kautz
Santiago Torres-Arias

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.SP.800-204D>

NIST SP 800-204D
February 2024

Software Supply Chain Security
in DevSecOps CI/CD Pipelines

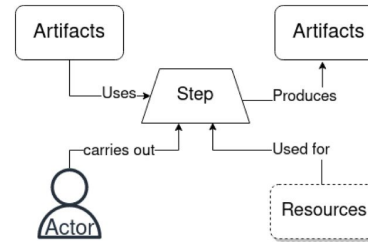


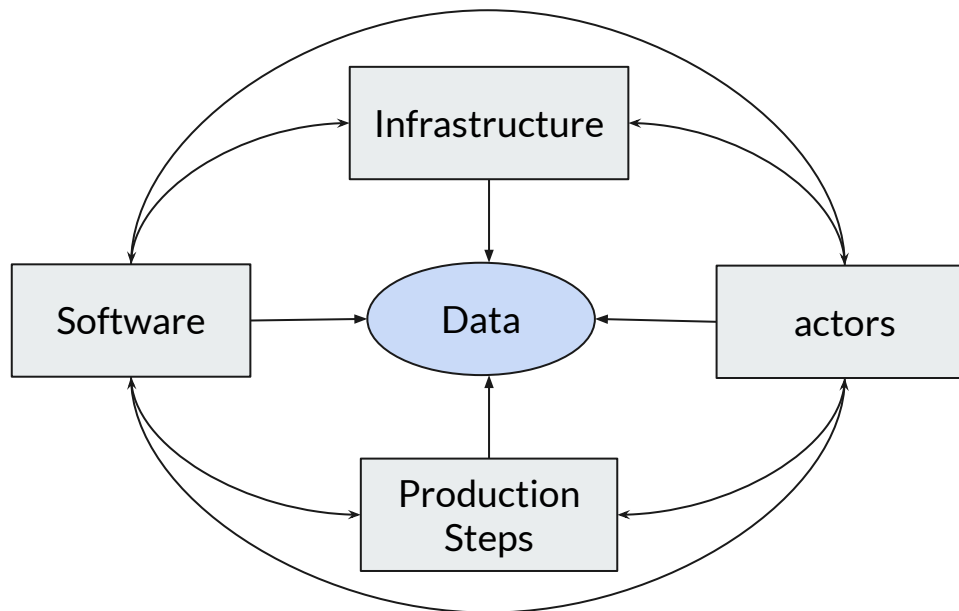
Fig. 1. Interaction between the different elements¹ of a software supply chain (SSC) step²

2.4.1. Software Supply Chain Defects

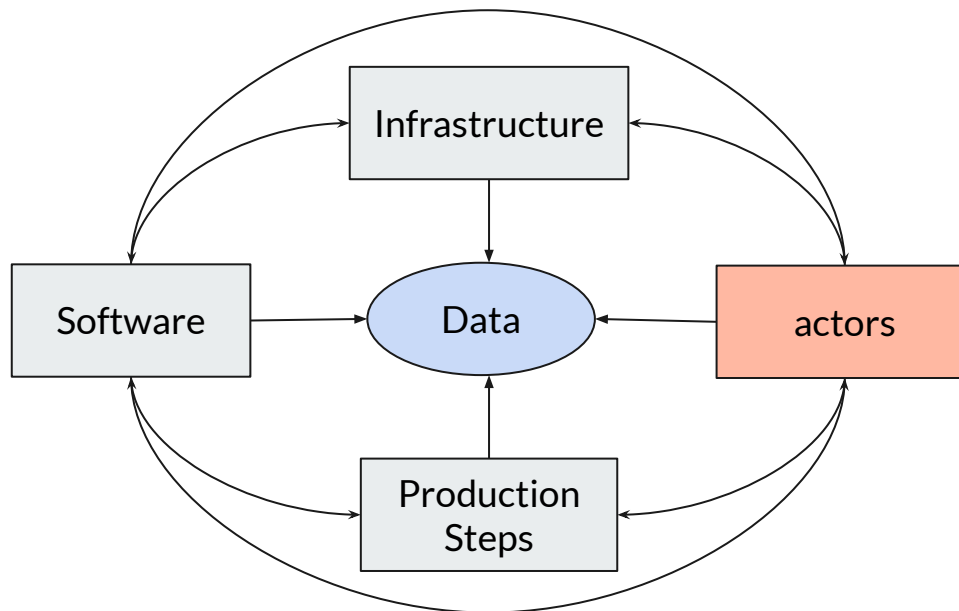
Much like software defects (i.e., bugs), defective artifacts can propagate throughout an SSC and affect its security posture. A noteworthy example of such a defect is Log4Shell [5], where a vulnerability in a highly-used software artifact allowed attackers to compromise a large number of targets with very little effort.

If software is used in a manner that it was not originally intended or configured for, it may result in an insecure state. However, while the line between a defect and an attack is often blurred in the context of SSC, the guiding principle is that of intent — that is, whether or not the upstream actor intended to exploit that defect. In the context of software engineering, not all defects are vulnerabilities, regardless of intent. Vulnerabilities may be present for other reasons, and that presence does not guarantee exploitation, which is what defines an attack. Malicious actors complete the defect-attack chain by intentionally introducing weaknesses that they can later exploit.

What do we need to defend SSCs?



What do we need to defend SSCs?





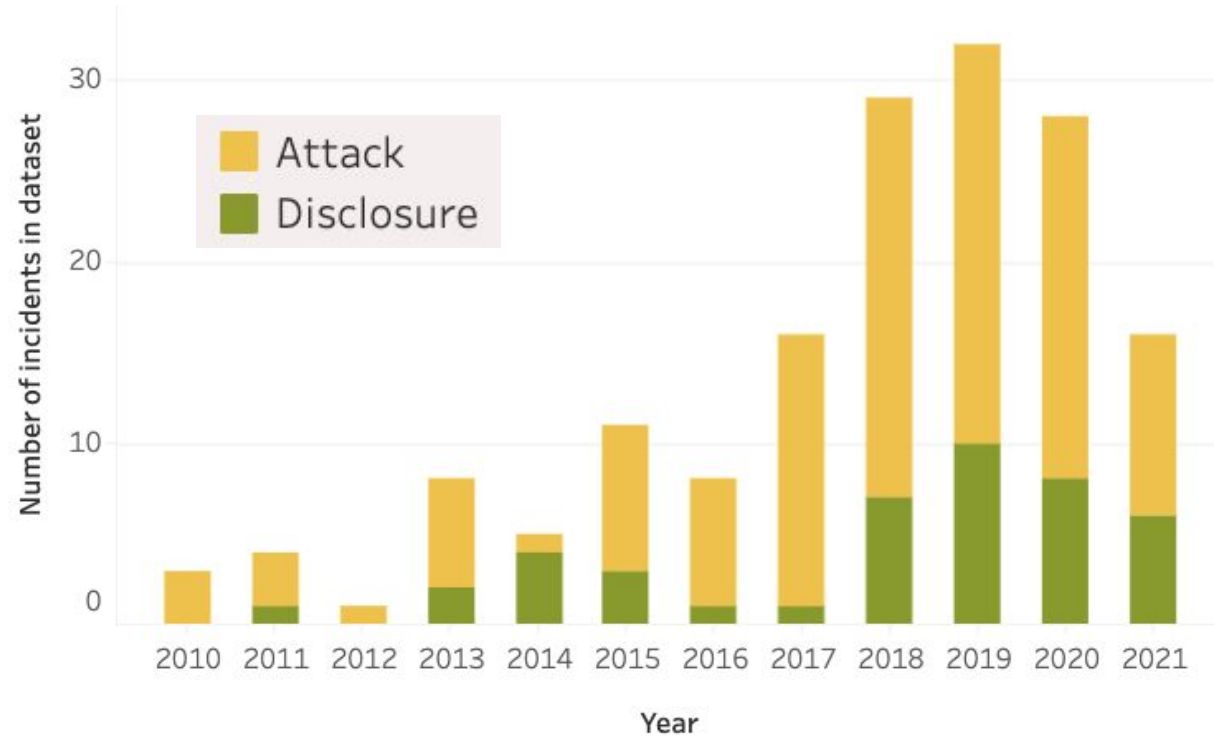
Sigstore: software signing for everybody

“Sigstore: Software Signing for Everybody,” Computer and Communications Security ‘22,
<https://dl.acm.org/doi/10.1145/3548606.3560596>.

Problem

Attacks on the software supply chain have increased in the past decade.

Changes in attacks and disclosures over the years



Atlantic Council, 2021, “Breaking Trust,” available at <https://www.atlanticcouncil.org/commentary/trackers-and-data-visualizations/breaking-trust-the-dataset/>.

Software Signing

Part of the solution.

- Someone (not the expected author) publishes software
 - Compromises of account credentials
 - Compromises of package repositories.
- Not all attacks!
 - Normal vulnerabilities
 - Build system compromises
 - “Hypocrite commits”
 - Threatening authors
- *If you know who’s supposed to sign a package, signing helps.*

donald stufft

Why Package Signing is not the Holy Grail

23 July 2013

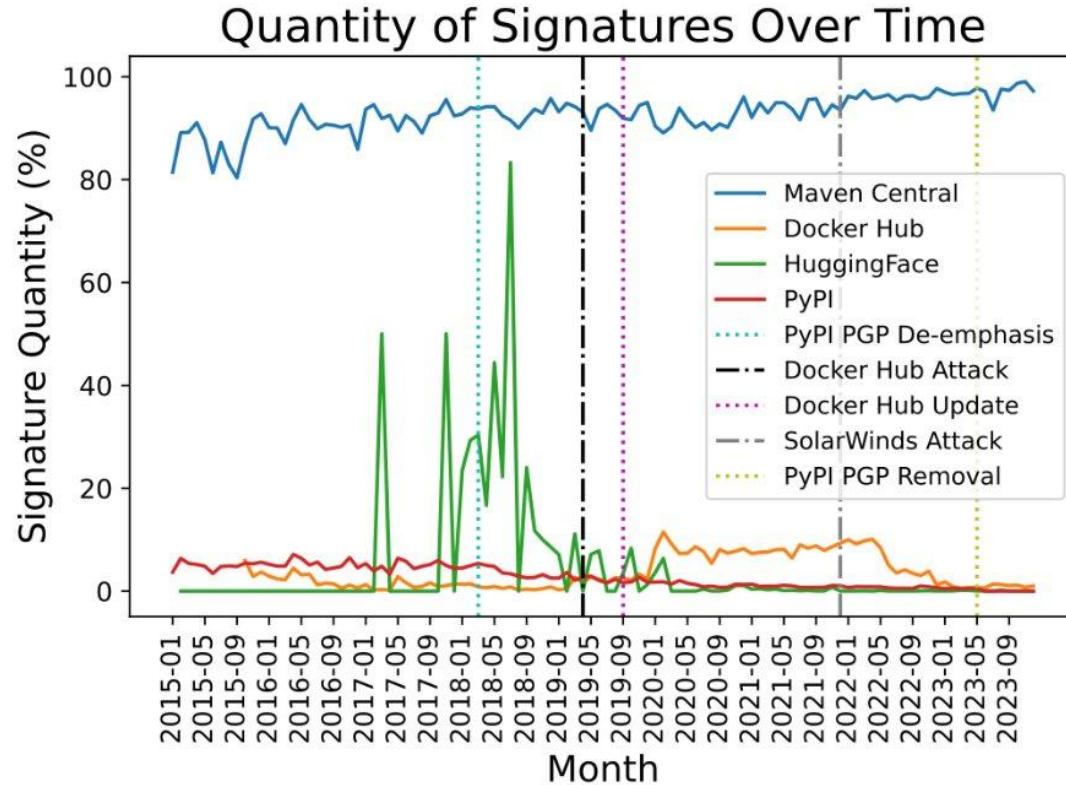
Typically any time the topic of security and software packages, in my case typically Python packages, comes up someone seems to come up with the “helpful” suggestion of “Just Use X!”, where X is typically GPG but can be any of a wide range of signing technologies. Quite often the people suggesting it have latched onto signing packages as some sort of voodoo you can throw at the problem and magically get “security”.

Software Signing

Nobody does it.

- Many package repositories support it: PyPI, RubyGems, NuGet, ELPA
 - <4% adoption on PyPI
 - Tools don't check signatures by default
- Exception that proves the rule: Maven Central
 - Java is an “enterprise-y” language
 - Teams are able to jump through hoops
 - Packages namespaced by domain (`com.example.foo`)

Software Signing

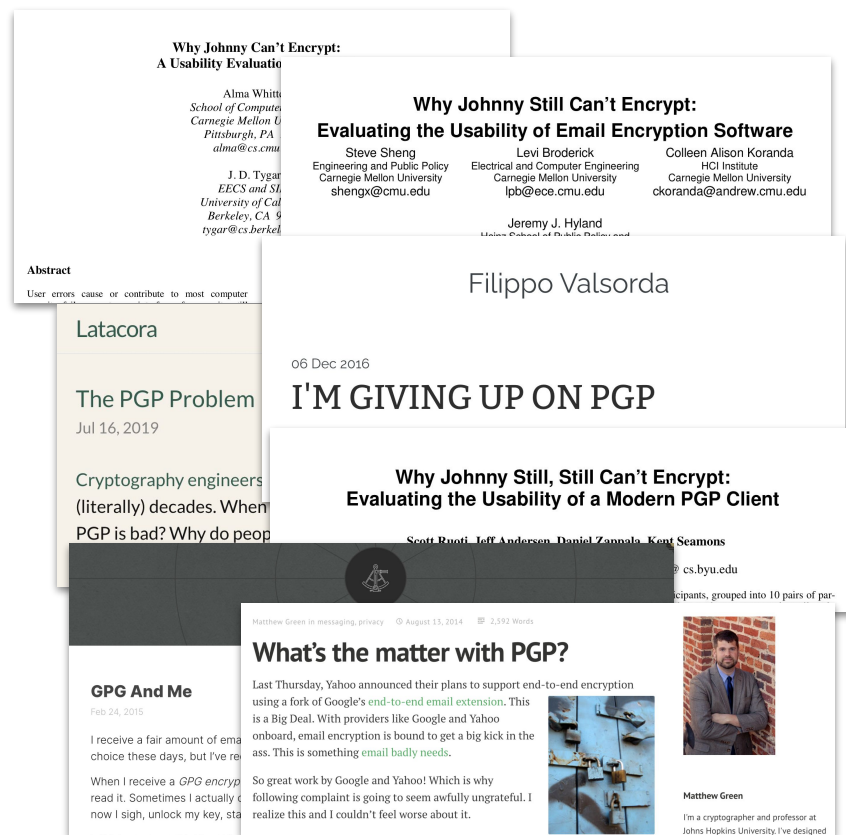


"Signing in Four Public Software Package Registries: Quantity, Quality, and Influencing Factors" IEEE Security & Privacy, 2024

Software Signing

Nobody does it.

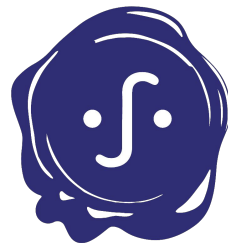
- Why not?
 - Well-known usability issues
 - Key management
 - “What do you mean I can’t push from my laptop on vacation?”
 - Developers expect account recovery
- Exception: mobile apps
 - App stores check signatures on publish
 - Keys can be replaced by store



Sigstore

Goals

- Meet developers where they are
 - Support old-fashioned signatures
 - And hardware tokens
- Accept centralization, with accountability (like web PKI)
- Free developers from key management
- Integrate with other tools
 - Software supply chain solutions
 - Developer tools and infrastructure

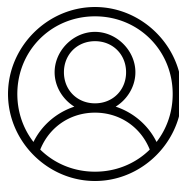


sigstore

Client Flow

Signing:

1. Get auth token (“OAuth dance”)
2. Generate key pair
3. Send token, precertificate to Fulcio
4. Get X.509 certificate
5. Sign artifact
6. Send signature, metadata to Rekor
7. Publish: artifact, signature, certificate, signed timestamp

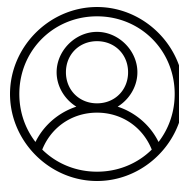


(As identity provider)

Client Flow

Signing:

1. Get auth token (“OAuth dance”)
2. Generate key pair
3. Send token, precertificate to Fulcio
4. Get X.509 certificate
5. Sign artifact
6. Send signature, metadata to Rekor
7. Publish: artifact, signature, certificate, signed timestamp

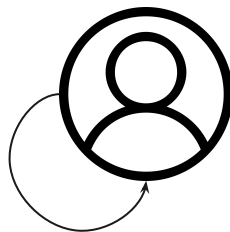


(As identity provider)

Client Flow

Signing:

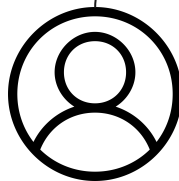
1. Get auth token (“OAuth dance”)
2. **Generate key pair**
3. Send token, precertificate to Fulcio
4. Get X.509 certificate
5. Sign artifact
6. Send signature, metadata to Rekor
7. Publish: artifact, signature, certificate, signed timestamp



Client Flow

Signing:

1. Get auth token (“OAuth dance”)
2. Generate key pair
3. **Send token, precertificate to Fulcio**
4. Get X.509 certificate
5. Sign artifact
6. Send signature, metadata to Rekor
7. Publish: artifact, signature, certificate, signed timestamp

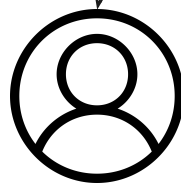


(As identity provider)

Client Flow

Signing:

1. Get auth token (“OAuth dance”)
2. Generate key pair
3. Send token, precertificate to Fulcio
4. **Get X.509 certificate**
5. Sign artifact
6. Send signature, metadata to Rekor
7. Publish: artifact, signature, certificate, signed timestamp

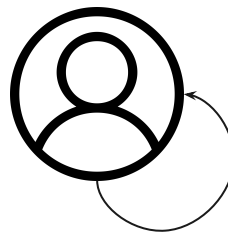


(As identity provider)

Client Flow

Signing:

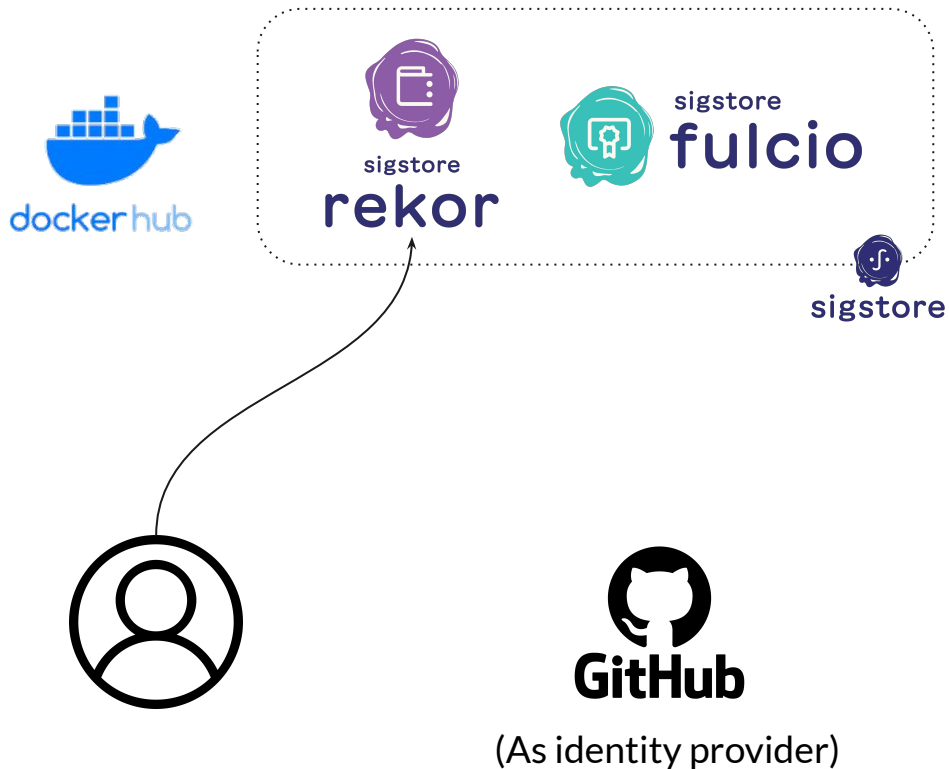
1. Get auth token (“OAuth dance”)
2. Generate key pair
3. Send token, precertificate to Fulcio
4. Get X.509 certificate
5. **Sign artifact**
6. Send signature, metadata to Rekor
7. Publish: artifact, signature, certificate, signed timestamp



Client Flow

Signing:

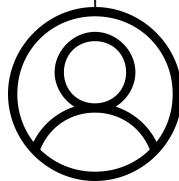
1. Get auth token (“OAuth dance”)
2. Generate key pair
3. Send token, precertificate to Fulcio
4. Get X.509 certificate
5. Sign artifact
6. **Send signature, metadata to Rekor**
7. Publish: artifact, signature, certificate, signed timestamp



Client Flow

Signing:

1. Get auth token (“OAuth dance”)
2. Generate key pair
3. Send token, precertificate to Fulcio
4. Get X.509 certificate
5. Sign artifact
6. Send signature, metadata to Rekor
7. **Publish: artifact, signature, certificate, signed timestamp**

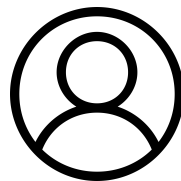


(As identity provider)

Client Flow

Verifying:

1. (Download artifact, certificate, timestamp)
2. Check certificate:
 - a. Signed by Fulcio
 - b. Subject: expected signer?
3. Check signature:
 - a. Valid for artifact
 - b. Timestamp during cert validity
4. Check transparency log membership

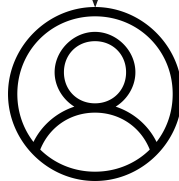


(As identity provider)

Client Flow

Verifying:

1. (Download artifact, certificate, timestamp)
2. Check certificate:
 - a. Signed by Fulcio
 - b. Subject: expected signer?
3. Check signature:
 - a. Valid for artifact
 - b. Timestamp during cert validity
4. Check transparency log membership

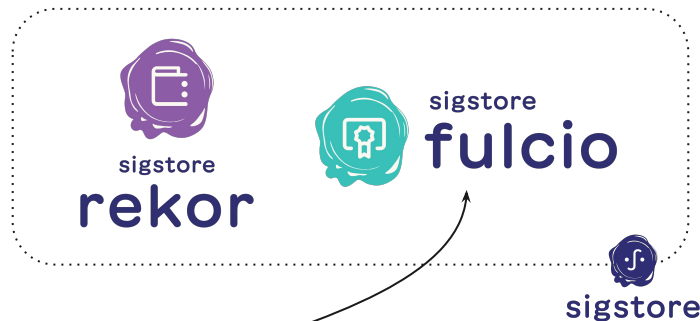


(As identity provider)

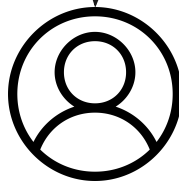
Client Flow

Verifying:

1. (Download artifact, certificate, timestamp)
2. **Check certificate:**
 - a. Signed by Fulcio
 - b. Subject: expected signer?
3. **Check signature:**
 - a. Valid for artifact
 - b. Timestamp during cert validity
4. Check transparency log membership



(no communication needed)

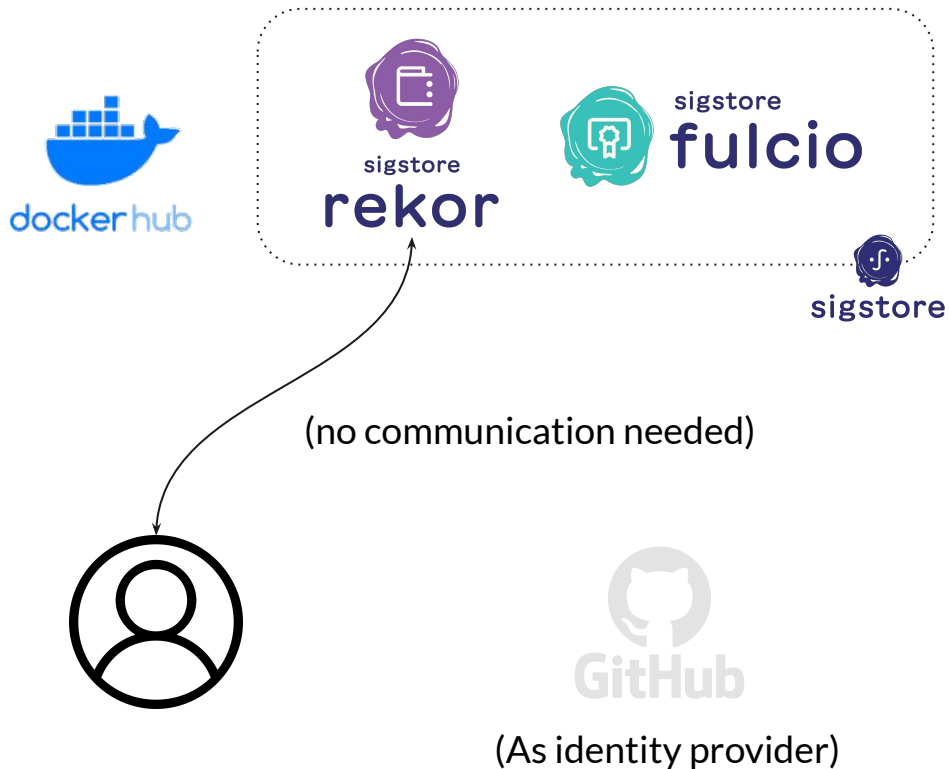


(As identity provider)

Client Flow

Verifying:

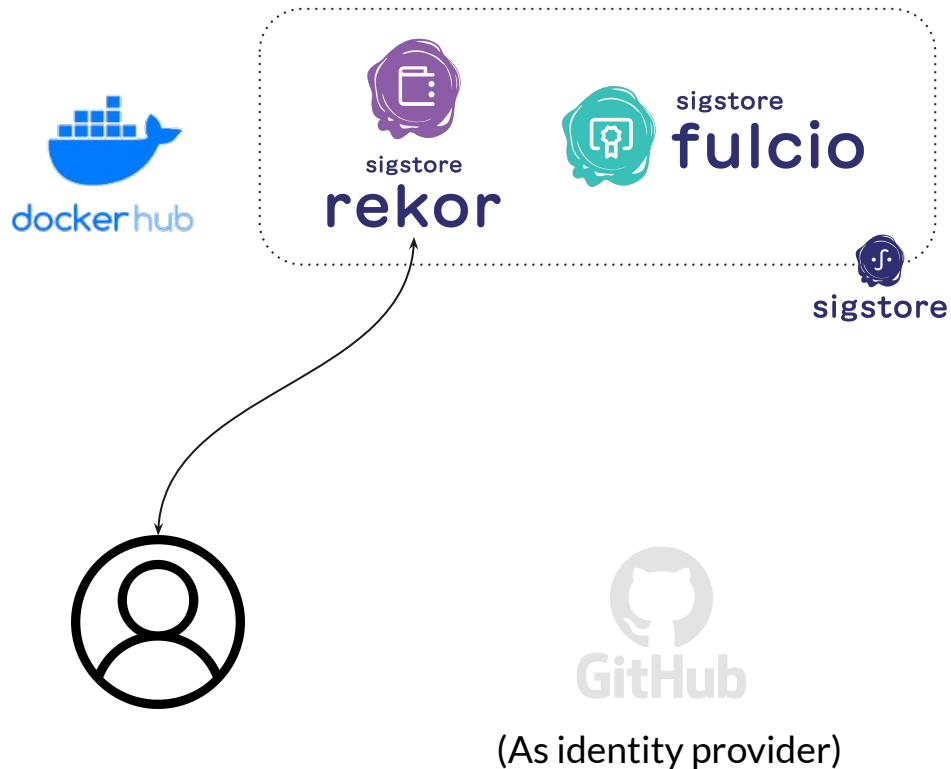
1. (Download artifact, certificate, timestamp)
2. Check certificate:
 - a. Signed by Fulcio
 - b. Subject: expected signer?
3. **Check signature:**
 - a. Valid for artifact
 - b. Timestamp during cert validity
4. Check transparency log membership

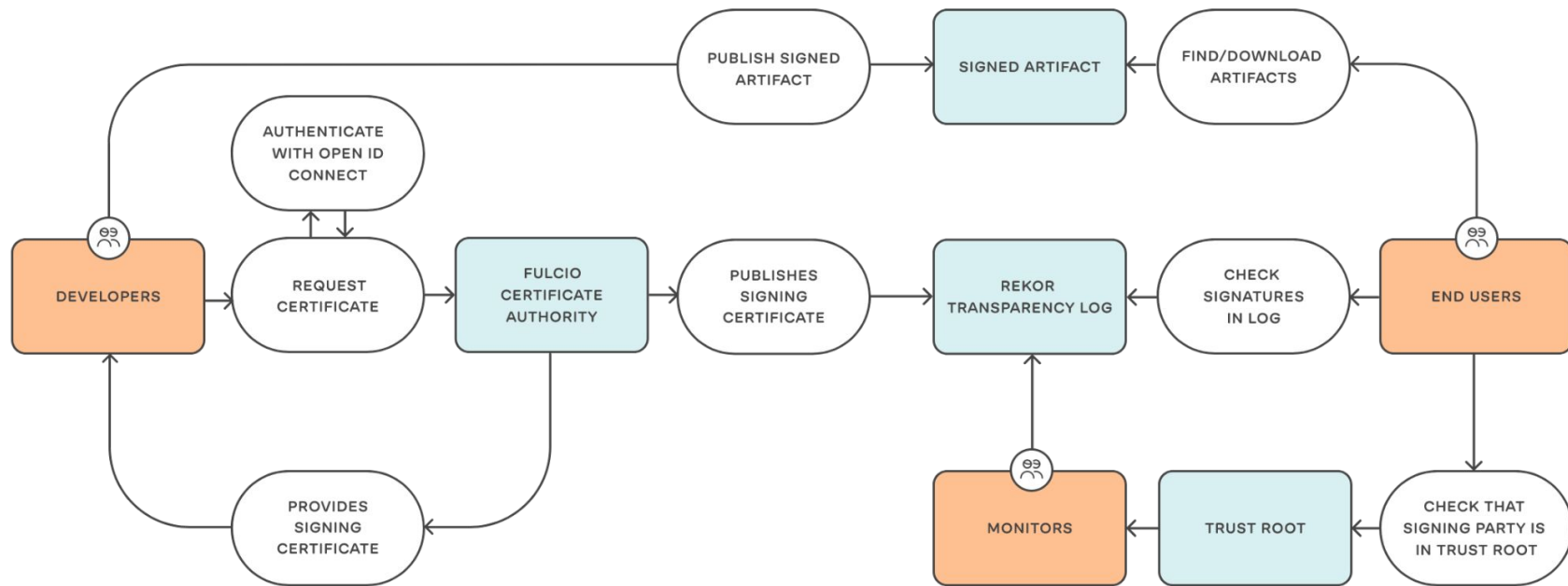


Client Flow

Verifying:

1. (Download artifact, certificate, timestamp)
2. Check certificate:
 - a. Signed by Fulcio
 - b. Subject: expected signer?
3. Check signature:
 - a. Valid for artifact
 - b. Timestamp during cert validity
4. **Check transparency log membership**





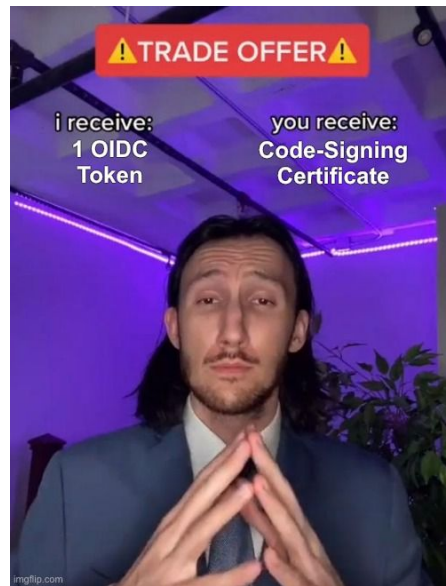
Sigstore

Let's take that step by step!

1. Identity
2. Signature timestamping
3. Transparency



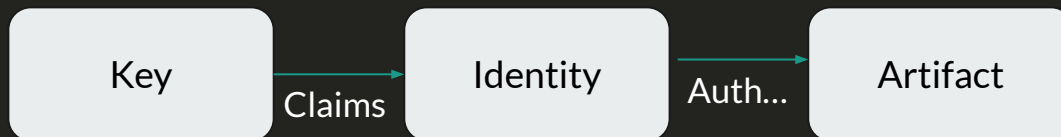
- “A new kind of CA”
- Users authenticate
 - OpenID Connect (OIDC: “login with Facebook”)
 - Secure Production Identity Framework for Everyone (SPIFFE)
 - Extensible: Decentralized Identifiers (DIDs)? SAML?
- Issues X.509 code signing certificate
 - SubjectAlternativeName: “Joel's Facebook account”
- Certificate is *ephemeral* (10 minute validity)



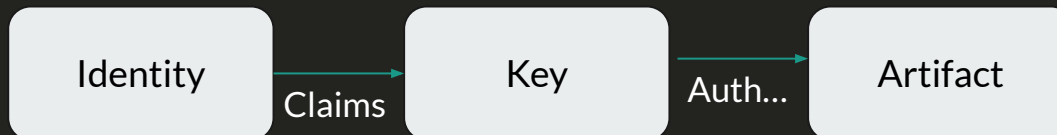


Idea: rely on identity providers to manage cryptographic material using ephemeral keys

Traditional models (e.g., PGP)



Next-gen signing (e.g., Sigstore, OPK)



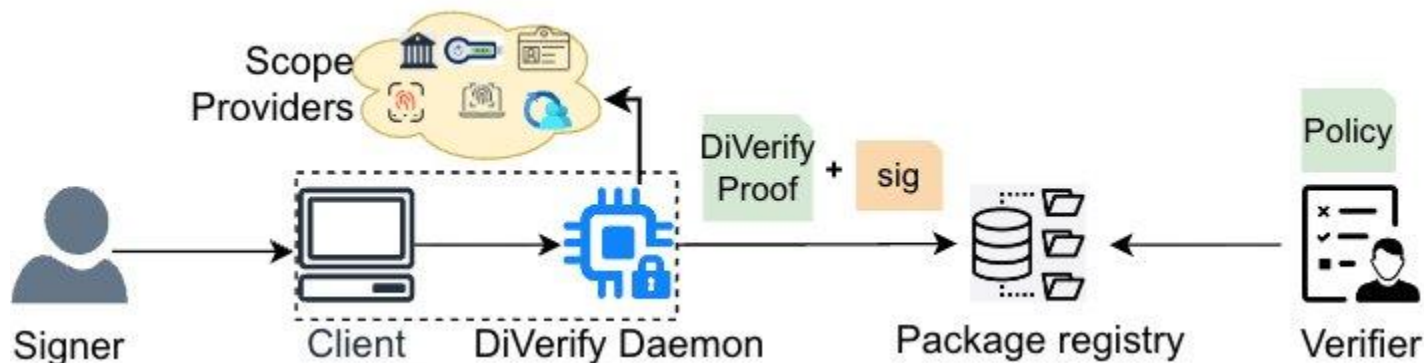


DiVerify: Extending Sigstore beyond identity providers

“Context-Aware Trust Verification for Identity-Based Software Signing” Automated Software Engineering ‘26

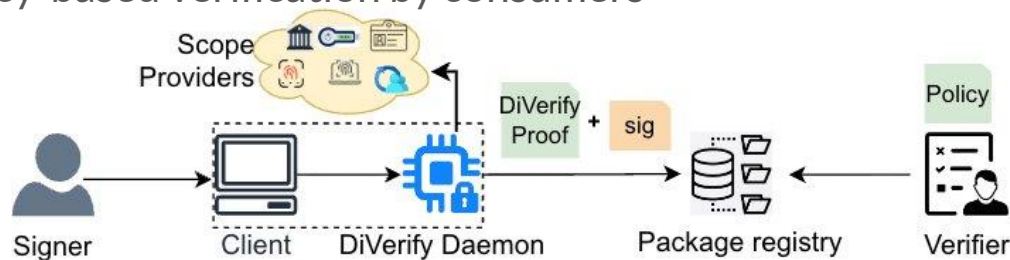
Context-aware signing

- Extend signing with verifiable context
- Attach attested evidence about signing conditions
- Enable policy-based verification by consumers



Context-aware signing

- Extend signing with verifiable context
- Attach attested evidence about signing conditions
- Enable policy-based verification by consumers



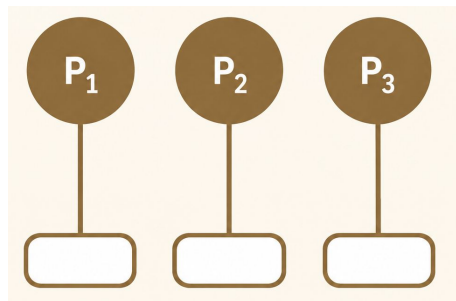
Verifying who signed



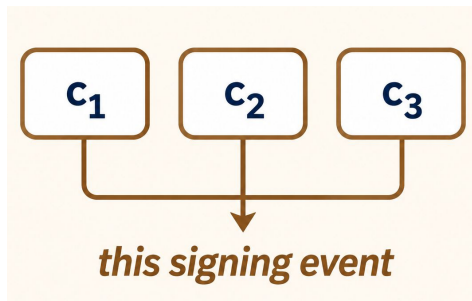
Verifying who signed + under what conditions signing occurred

Secure Provider Composition

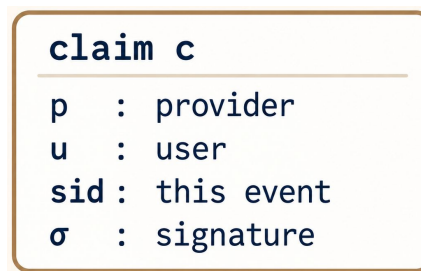
Provider Independence



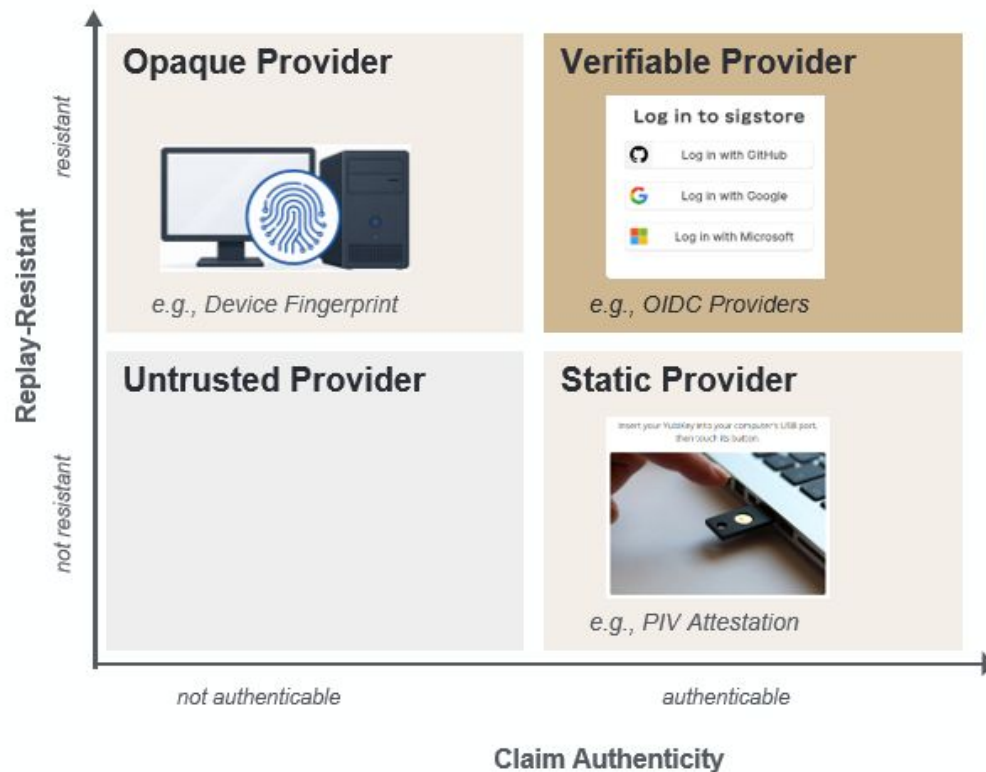
Session Binding



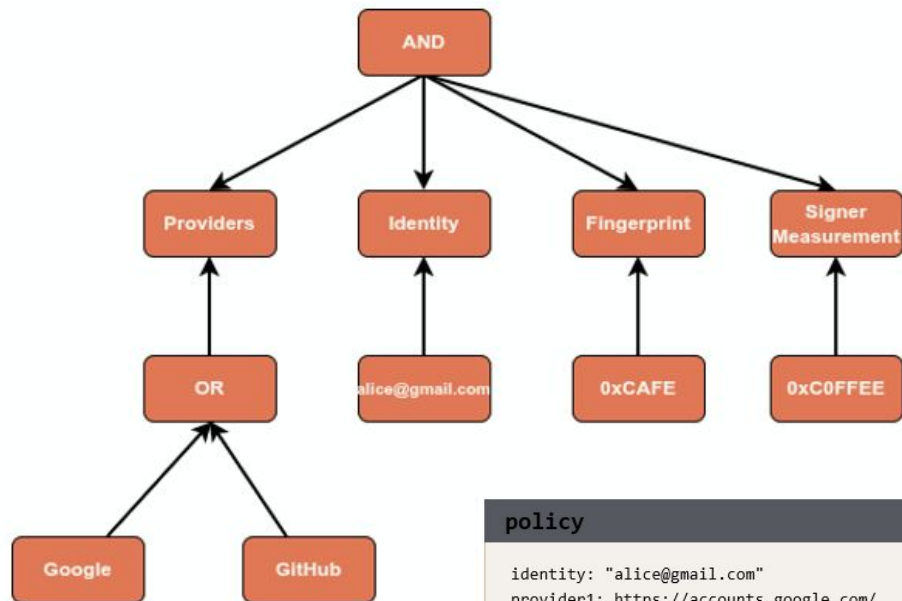
Claim Authenticity



Scope Provider Taxonomy



DiVerify Policy Governance



policy

```
identity: "alice@gmail.com"
provider1: https://accounts.google.com/
provider1: https://oauth.github.com/
device_fingerprint: "443d32cb2f8d7..."

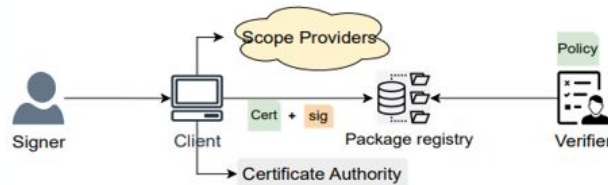
signer_measurement: "e823445d37f80a69350ff..."
rule: "identity AND (provider1 OR provider2) AND
      device_fingerprint AND signer_measurement"
```



Deployment Variants

Legacy-Compatible Mode

Multi-scope verification without client attestation

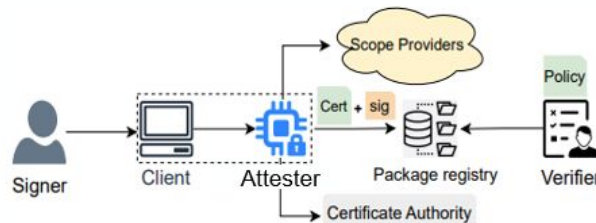


Legacy-Compatible Mode (no client attestation)



Trusted Authentication Mode

Extends to multi-scope retrieval, with client attestation. Client integrity is enforced by the CA

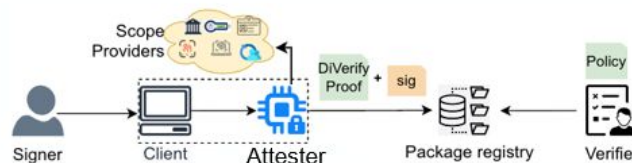


Trusted Authentication Mode



Core Mode

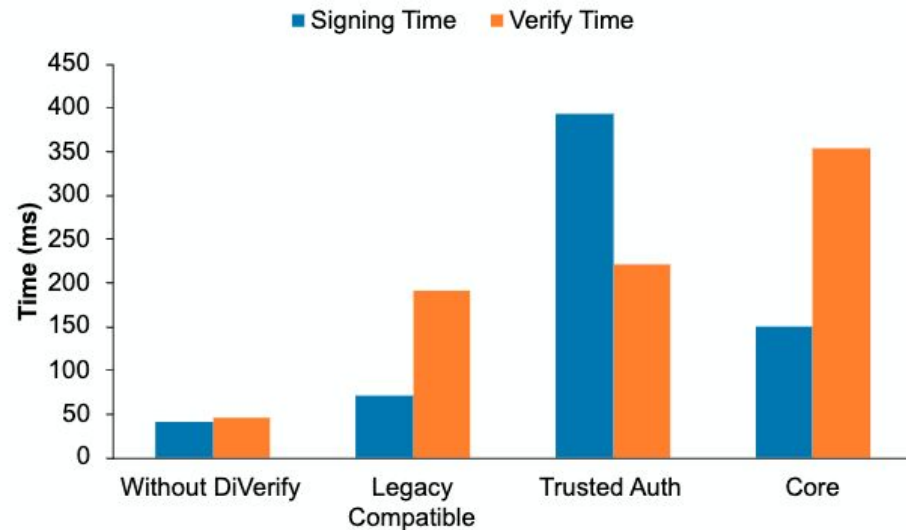
Extends to multi-scope retrieval, with client attestation. Client integrity is enforced by the verifier



Core Mode

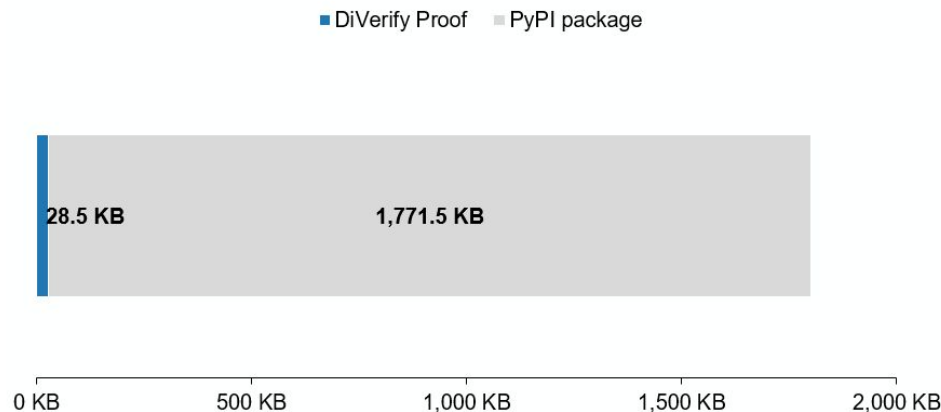
DiVerify Overhead: Signing vs Verify Latency

Adds overhead but remains practical, with signing time under 400 ms



DiVerify Proof Storage Cost

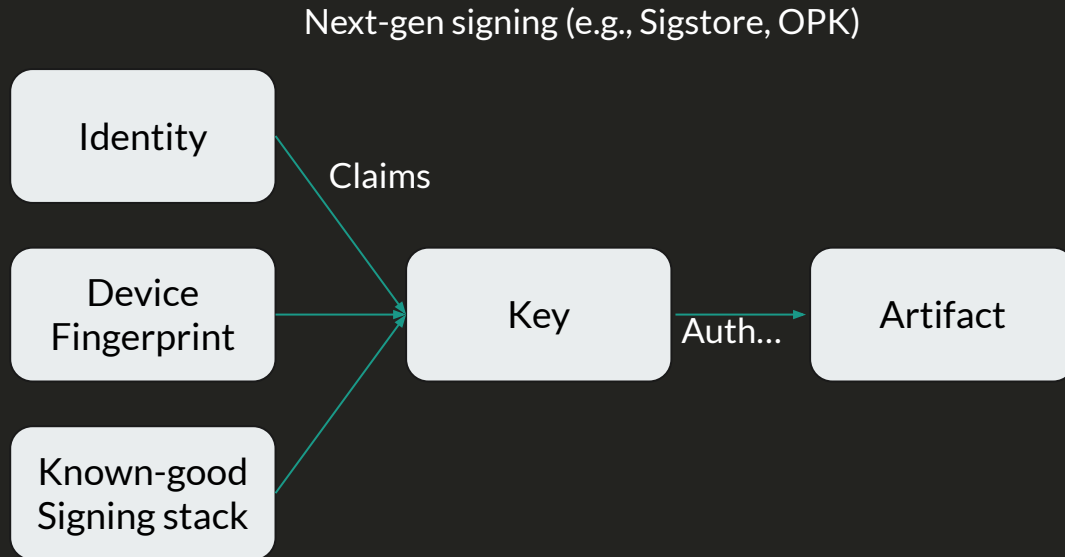
Worst-case overhead is <1.6% of the average PyPI package size.



*Average PyPI package $\approx$ 1.8 MB
DiVerify proof worst case (Trusted Auth, 28.5 kb) = $\sim$ 1.6% overhead*



Idea: We can decompose trust beyond a single element for robust protection





Speranza: providing actor confidentiality for the software supply chain

"Speranza: Usable, privacy-friendly software signing", ACM Computer and
Communications Security '23
<https://doi.org/10.1145/3576915.362320>

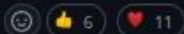


Motivation: OSINT on supply chain data

We can't prevent people from including their name in the source code of their crate, but [as Ashley pointed out in another comment](#) having the field in the `Cargo.toml` encourages people to fill it in when they're publishing a new crate and they're looking at what fields are available.

I'm sympathetic to a person wanting to change their name in the authors field, but I don't follow why they can't also use the "yank and publish an update" system we already have for other problems with published crate content.

@Lokathor unfortunately yanking can't prevent people from being harassed or doxxed. Both of those things happened multiple times already, and I want avoid crates.io becoming a tool to harass as much as possible.



This also only removes their name from the internet, but not the contents they created. Is this really meaningful in that sense? In particular, what if for example, their names for some reason got into the code section of another person's crate?

This is anecdotal evidence, but I have had access to help@crates.io for almost two years, and all of the cases where personal information needed to be deleted were related to `package.authors` , not the source code of the crates. Of course we can't prevent people from intentionally adding their name in the source code, but not forcing them to do so will address most of the issues.



Insight: authors do not need to disclose their identity to prove they are allowed to sign artifacts or metadata

Instead:

- Other actors & infrastructure can commit to a "view" of the permissions of such actor
- An actor can use a single-use key (recall Sigstore & next-gen software signing) to create a single, transparent signature
- A verifier can verify that the signature at time (t) satisfies the commitments from other actors



Insight: authors do not need to disclose their identity to prove they are allowed to sign artifacts or metadata

Instead:

- Other actors & infrastructure can commit to a "view" of the permissions of such actor
- An actor can use a single-use key (recall Sigstore & next-gen software signing) to create a single, transparent signature
- A verifier can verify that the signature at time (t) satisfies the commitments from other actors

(this can be generalized to prove more granular & universal signer properties, but that's a separate paper)



Fundamental building block: commitment schemes

1. $\text{Generate}(\lambda) \rightarrow \text{pp}$: generate public parameters.
2. $\text{Commit}(\text{pp}, m) \rightarrow (c, r)$: create commitment c to message m with randomness r .
3. $\text{Verify}(\text{pp}, m, c, r) \rightarrow \text{yes/no}$: check the commitment.



New building block: co-commitment schemes

1. $\text{CoCo.Generate}(\lambda) \rightarrow \text{pp} = (\text{pppdrsn}, \text{ppnizk})$: create public parameters for co-commitments.
 - a. Return $(\text{Pedersen.Generate}(\lambda), \text{Pedersen.GenerateEq}(\lambda))$.
2. $\text{CoCo.Commit}(\text{pp}, \text{id}) \rightarrow (c, r)$: create a commitment.
 - a. Return $\text{Pedersen.Commit}(\text{pppdrsn}, \text{id})$
3. $\text{CoCo.Prove}(\text{pp}, \text{id}, c1, r1, c2, r2) \rightarrow \pi \text{ or } \perp$: prove $c1$ and $c2$ are co-commitments.
 - a. Return $\text{Pedersen.ProveEq}(\text{pppdrsn}, \text{ppnizk}, \text{id}, c1, r1, c2, r2)$
4. $\text{CoCo.Verify}(\text{pp}, \text{id}, c, r) \rightarrow \text{yes/no}$
 - a. Return $\text{Pedersen.Verify}(\text{pppdrsn}, \text{id}, c, r)$
5. $\text{CoCo.VerifyEq}(\text{pp}, c1, c2, \pi) \rightarrow \text{yes/no}$
 - a. Return $\text{Pedersen.VerifyEq}(\text{pppdrsn}, \text{ppnizk}, c1, c2, \pi)$



FillGraph

We also define FillGraph, which creates a “co-commitment graph”: a graph where nodes are commitments to the same identity, and edges are proofs of equality between them.

$\text{FillGraph}(\text{pp}, G, m) \rightarrow H$: for graph structure G and message m , returns an instantiated graph H where nodes are commitments to m computed as $\text{Pedersen.Commit}(\text{pp}, \text{drsn}, m)$, and edges are equality proofs between their connected nodes computed as

$$e_{ij} = \text{Pedersen.ProveEq}(\text{pp}, m, c_i, r_i, c_j, r_j).$$



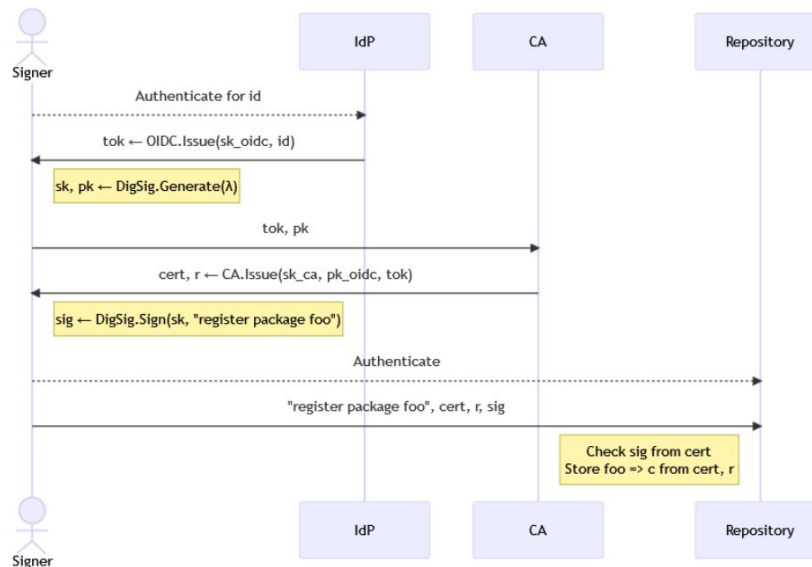
Linkability

If two commitments are co-commitments, even transitively, then they are commitments to the same identity.

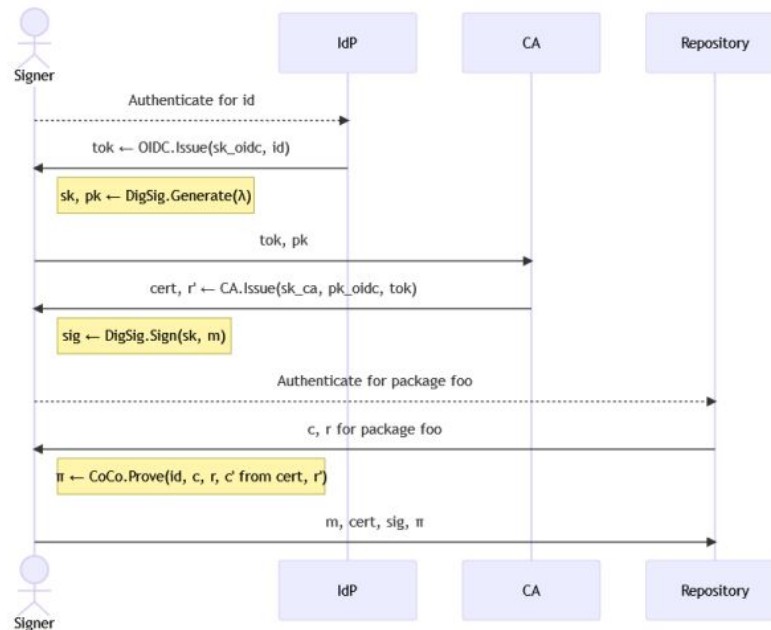
This means that no adversary should be able to generate a connected graph of co-commitments such that all of the following hold:

- The set of nodes contain commitments to at least two distinct identities ($|\{id_i\}| > 1$)
- All of the commitment equality proofs on the edges verify
- All of the commitments on the nodes verify

Speranza: package registration



Speranza: package signing





Speranza: Verification

1. Verifier retrieves the package, the certificate, the signature over the package, and the equality proof from the package repository.
2. Verifier retrieves the commitment associated with the package from the authorization record.
3. Verifier checks that the certificate is valid; if not, reject.
4. Verifier checks that the signature over the package is valid; if not, reject.
5. Verifier checks that the identity co-commitment is valid; if not, reject. If so, accept.



Speranza: Evaluation (costs)



Repository	Packages	Source
npm	"2 million"	npmjs.org
PyPI	451,913	pypi.org
RubyGems	176,365	rubygems.org
Arch User Repository	84,624	aur.archlinux.org
Ubuntu 23.04	35,587	repology.org
Hackage	15,903	hackage.haskell.org
Arch Linux (official)	13,889	archlinux.org

Table 2: Scale of real-world repositories



Sigstore's Impact

WIRED [SUBSCRIBE](#)

LILY HAY NEWMAN SECURITY JUN 18, 2021 9:00 AM

A New Tool Wants to Save Open Source From Supply Chain Attacks

Sigstore will make code signing free and easy for software developers, providing an important first line of defense.

[f](#) [t](#) [e](#) [b](#)

Fortune 500 (10% of)



Architecture/Spec



Projects



Deployments



l.openssf.org



Sigstore is a new standard for signing, verifying, and protecting software. It is an OpenSSF project and this landscape is intended as a map to explore the Sigstore ecosystem.

Integrations



Language Clients



Signed With



Case Studies



Architecture/Spec

Projects

Integrations

Signed With

Open Source Security

Introducing npm package provenance

How to verifiably link npm packages to their source repository and build instructions.

Version
20.4.6

Built and signed on
GitHub Actions

New feature

Introducing npm package provenance

Source Commit
github.com/npm/pkg@ddc3c4

Build file
[github/workflows/release.yml](https://github.com/workflows/release.yml)

Public ledger
[Transparency log entry](#)

Authors

Brian DeHamer
 Philip Harrison

April 19, 2023

The GitHub Insider Newsletter

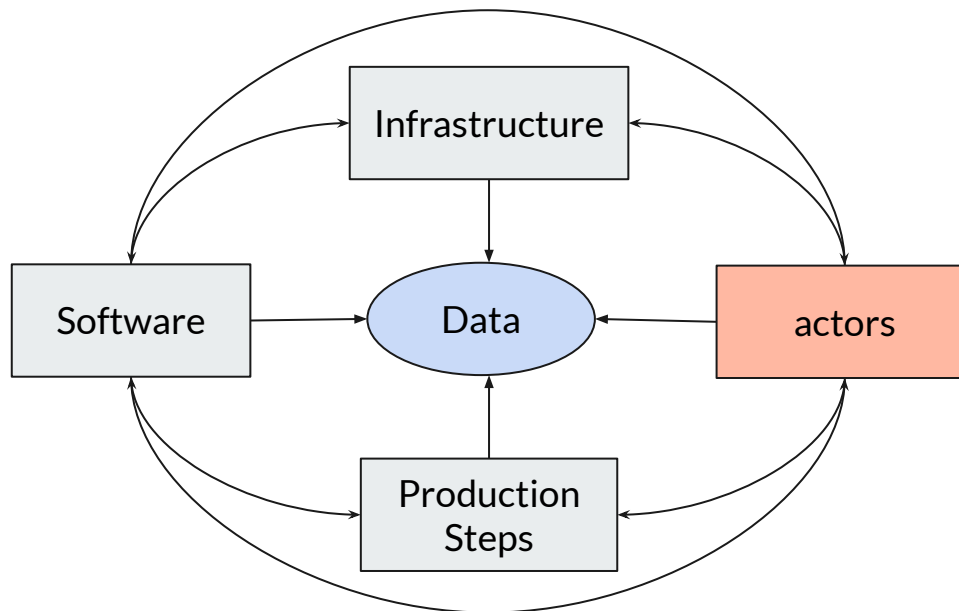
OpenSSF.org

Sigstore is a new standard for signing, verifying, and protecting software. It is an OpenSSF project and this landscape is intended as a map to explore the Sigstore ecosystem.

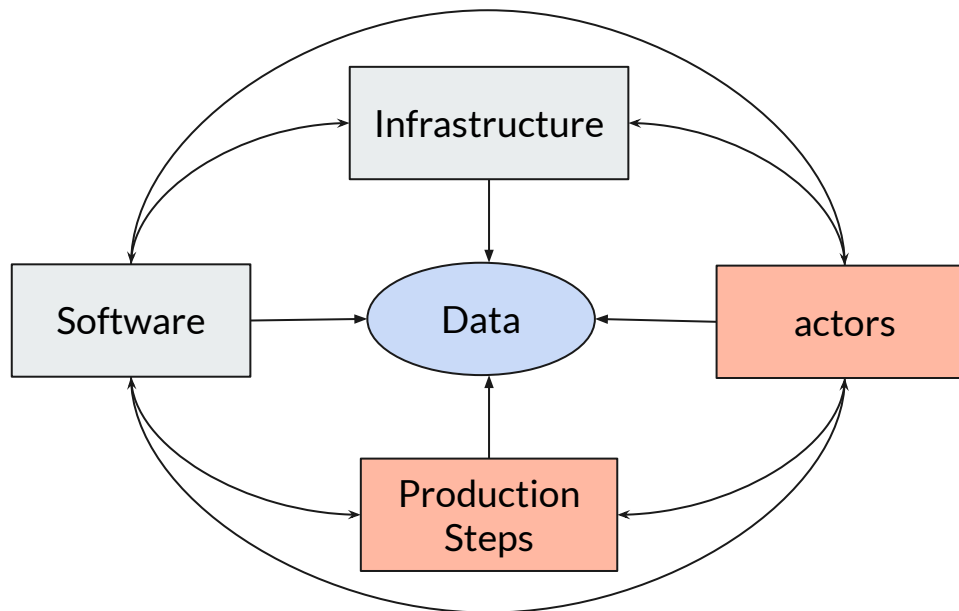
Language Clients

Case Studies

What do we need to model SSCs?



What do we need to model SSCs?



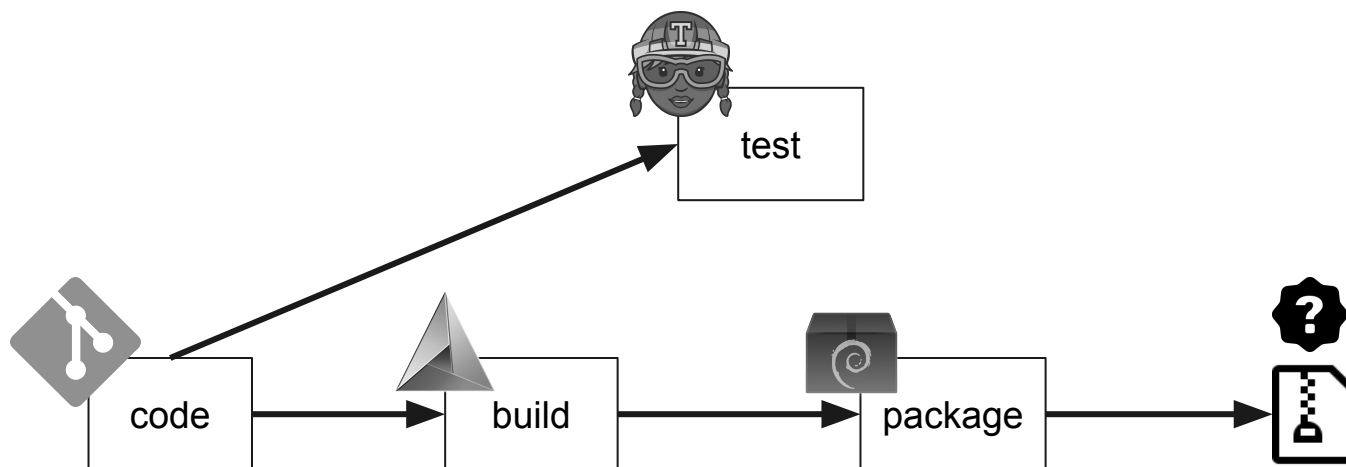


In-toto: providing farm-to-table guarantees for bits and bytes

"In-toto: providing farm-to-table guarantees for bits and bytes" USENIX Security '19

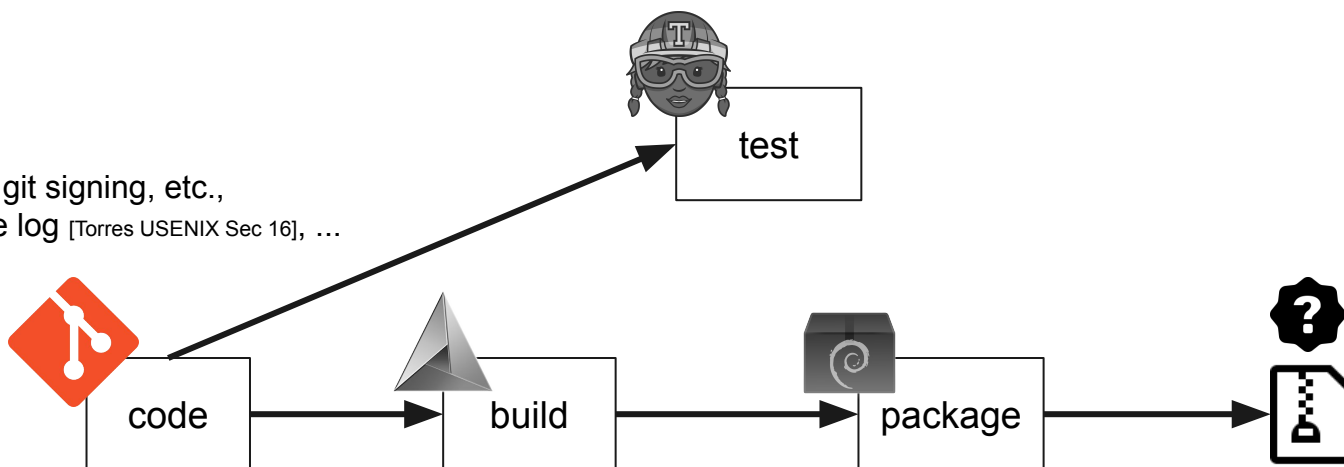
"Over the Shoulder: Improving SBOM Accuracy by Watching the Build" ACM/LF SCORED '26

Many good point solutions



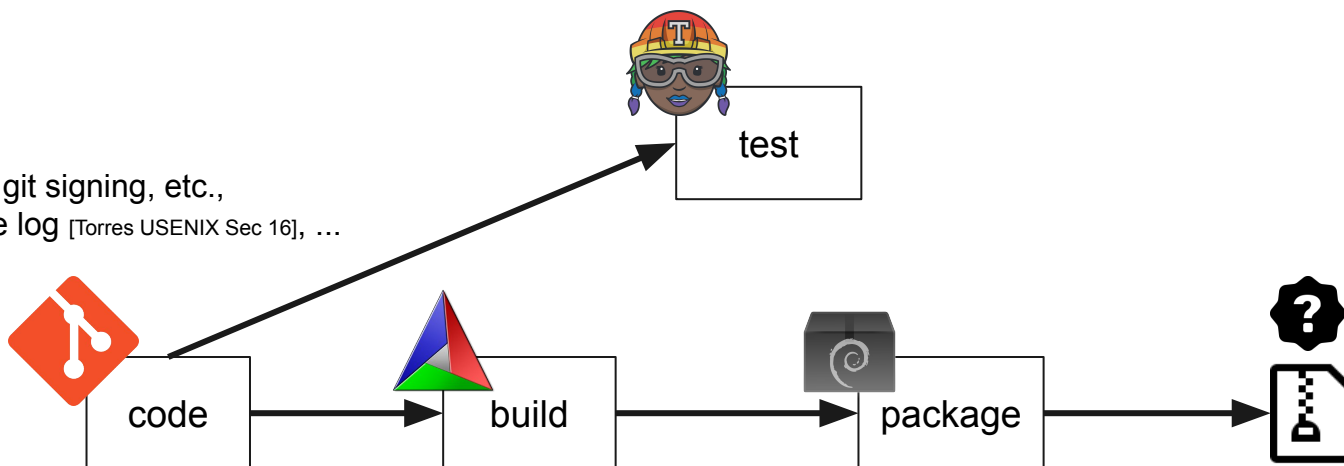
Many good point solutions

→ SVN, CVS, git signing, etc.,
reference state log [Torres USENIX Sec 16], ...



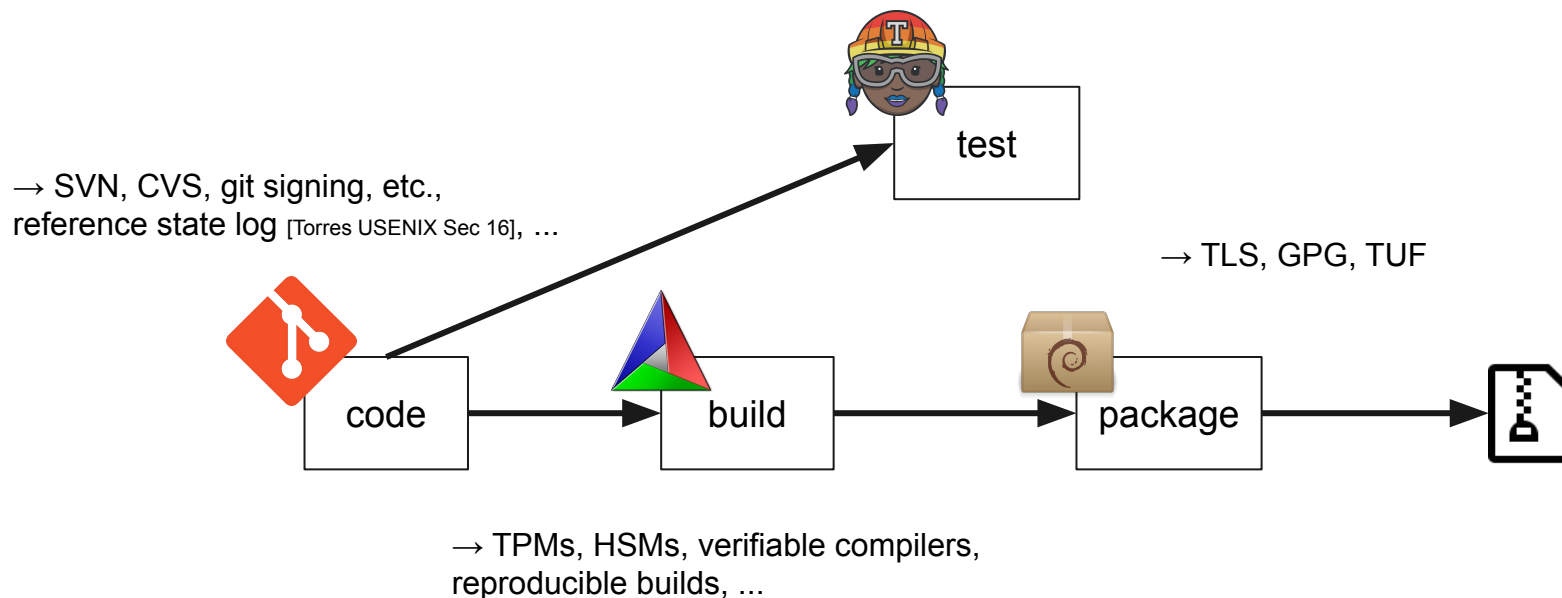
Many good point solutions

→ SVN, CVS, git signing, etc.,
reference state log [Torres USENIX Sec 16], ...

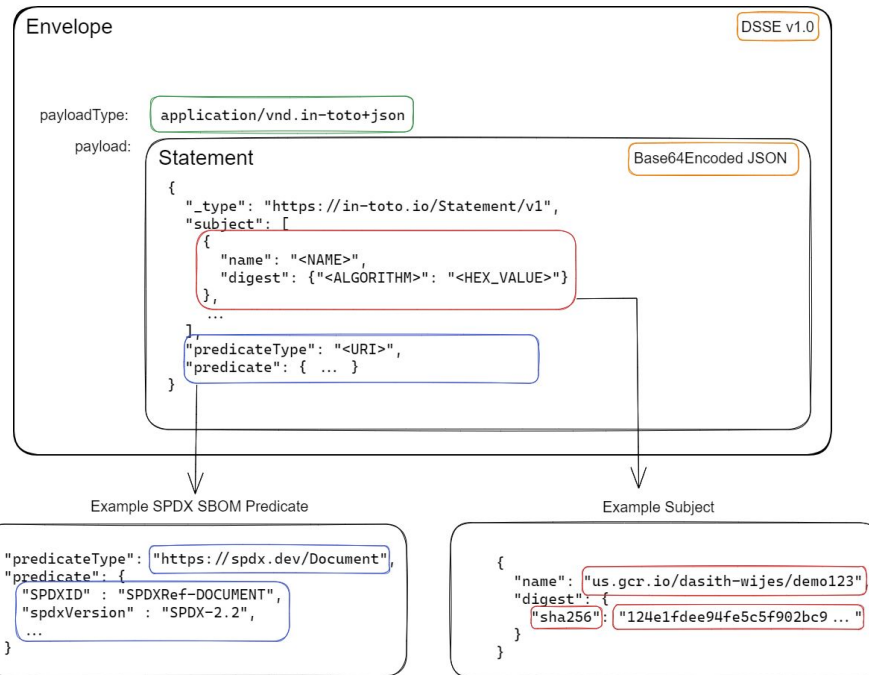


→ TPMs, HSMs, verifiable compilers,
reproducible builds, ...

Many good point solutions



Common Format: in-toto

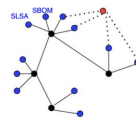


Many Systems for Attestation Generation

Evidence Gathering



Information Discovery



GUAC



SCITT



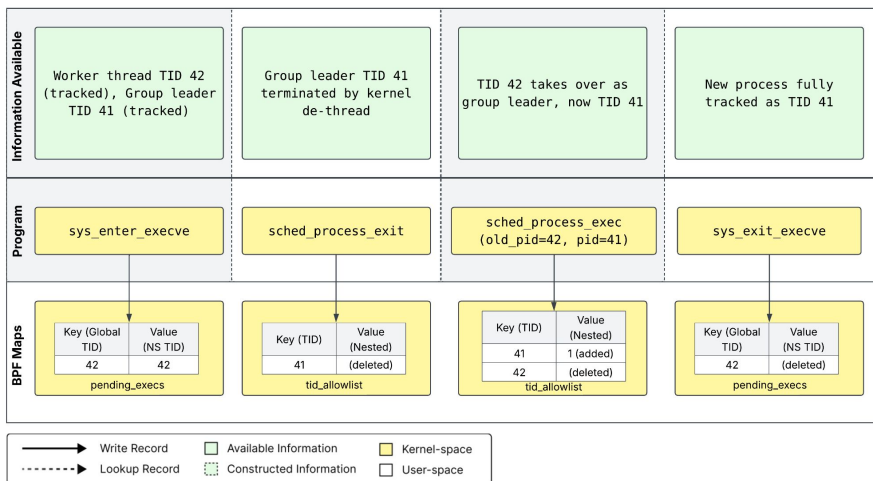
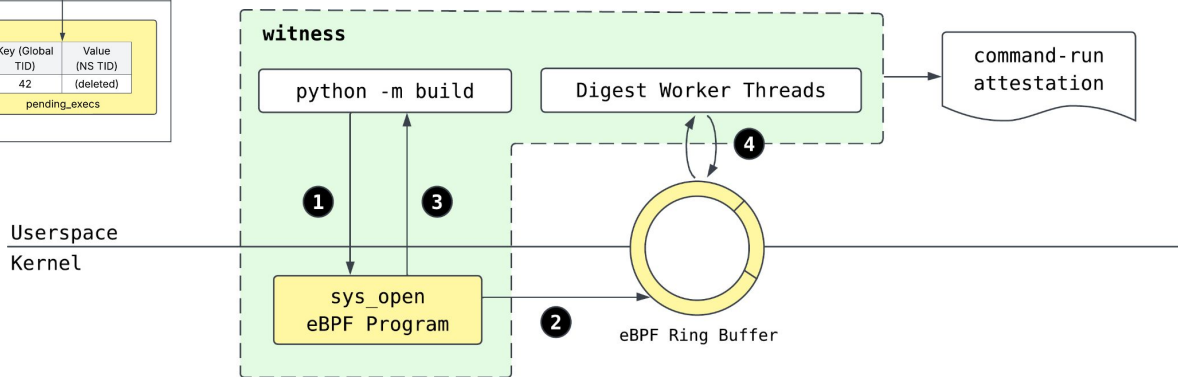
sigstore

Policy / Validation



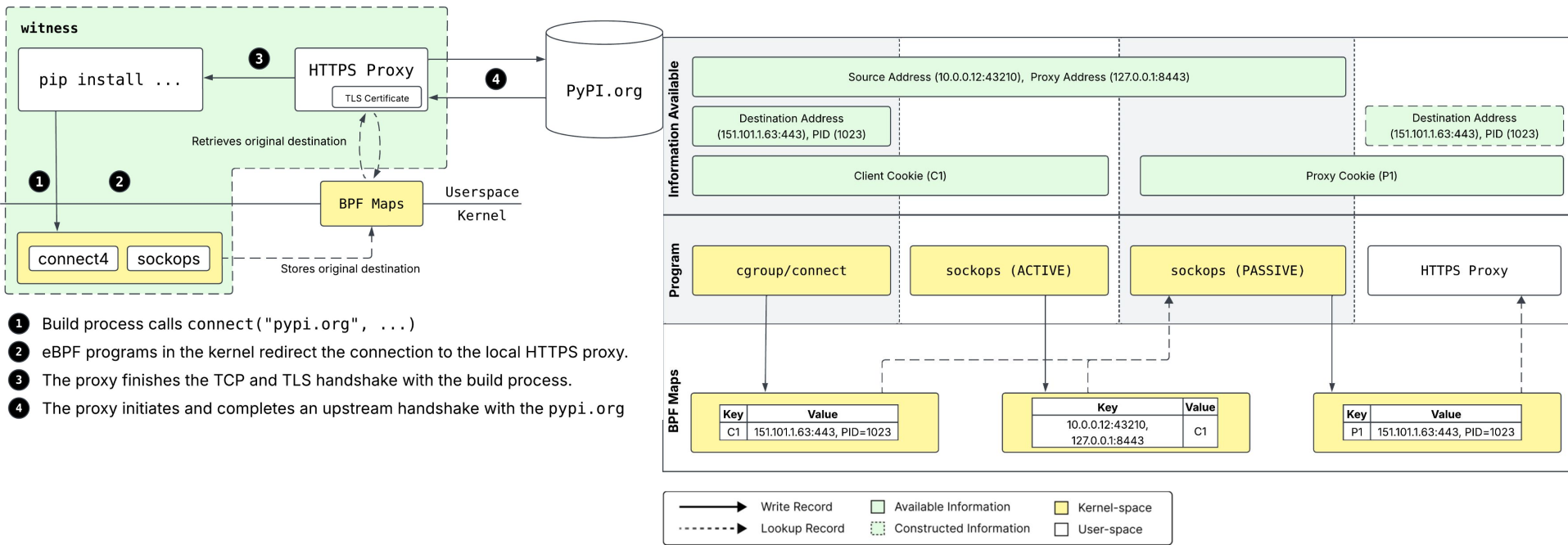
Open Policy Agent

Collecting this data in a generalizable way is hard!



- 1** Build process calls `open(...)`
- 2** The eBPF programs loaded at syscall entry and exit events store an OpenEvent in the ring buffer.
- 3** The open syscall returns, build process continues as expected.
- 4** Worker threads drain the ring buffer, resolve the paths and compute digests for accessed files.

Collecting this data in a generalizable way is hard!



Integrations & Adoptions



Chainloop



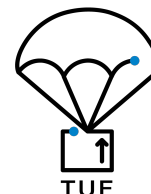
controlplane



GUAC



Reproducible Builds



TUE



DATADOG



debian

 CycloneDX

GUAC

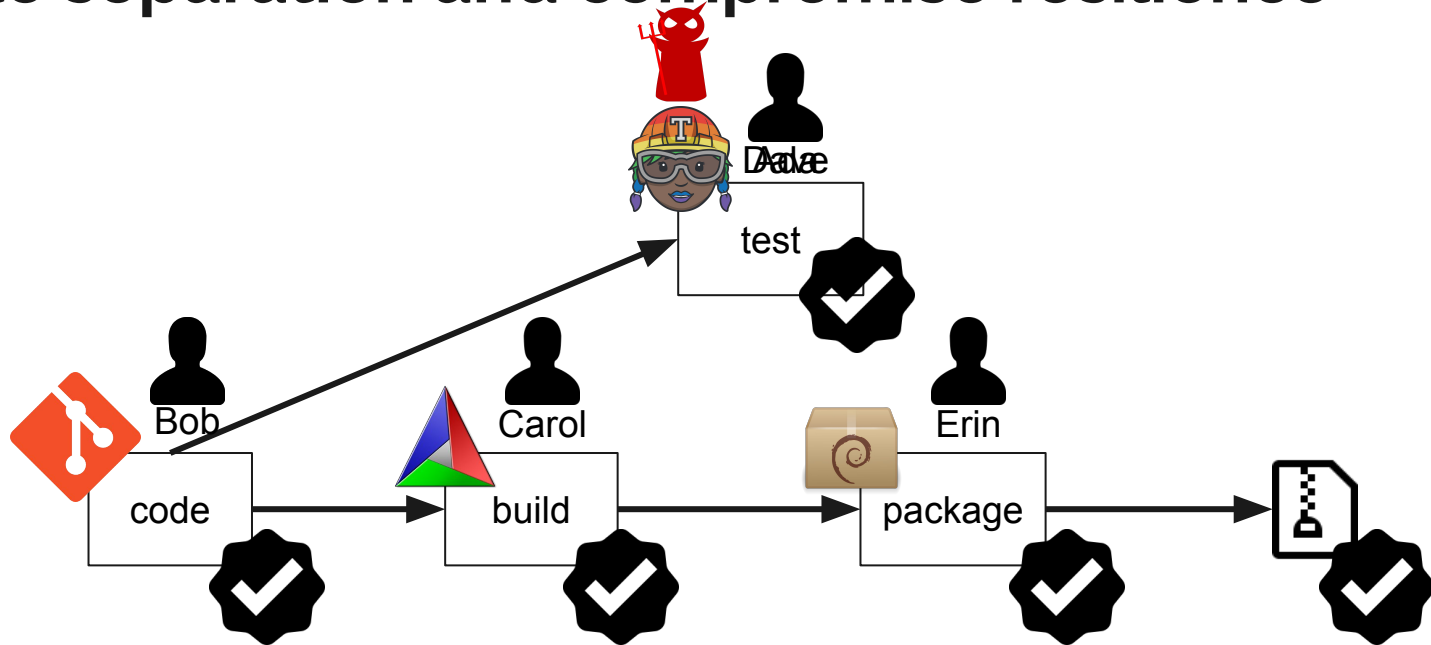


spiffe

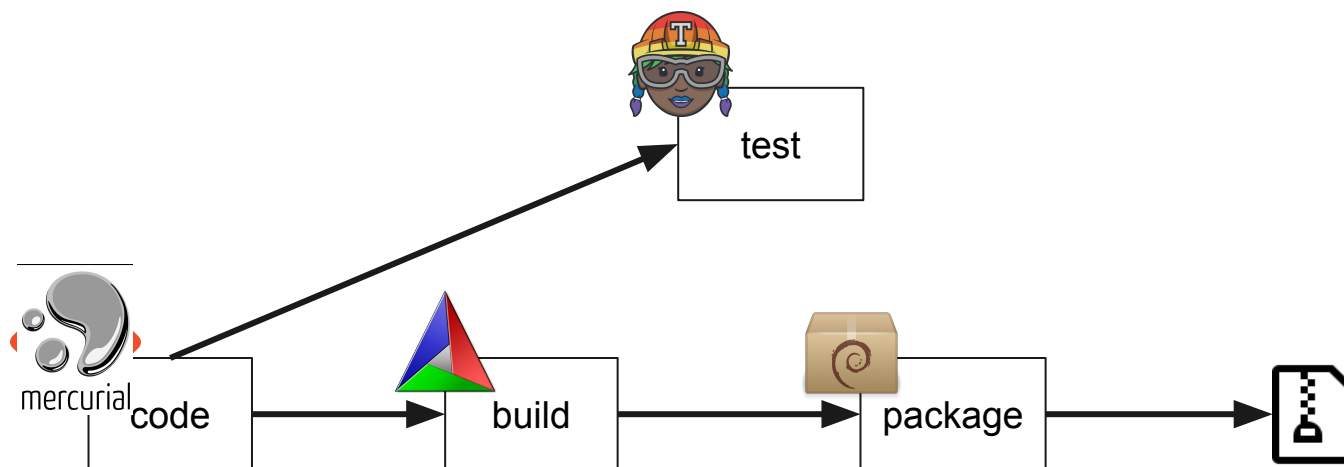


Jenkins

Role separation and compromise resilience

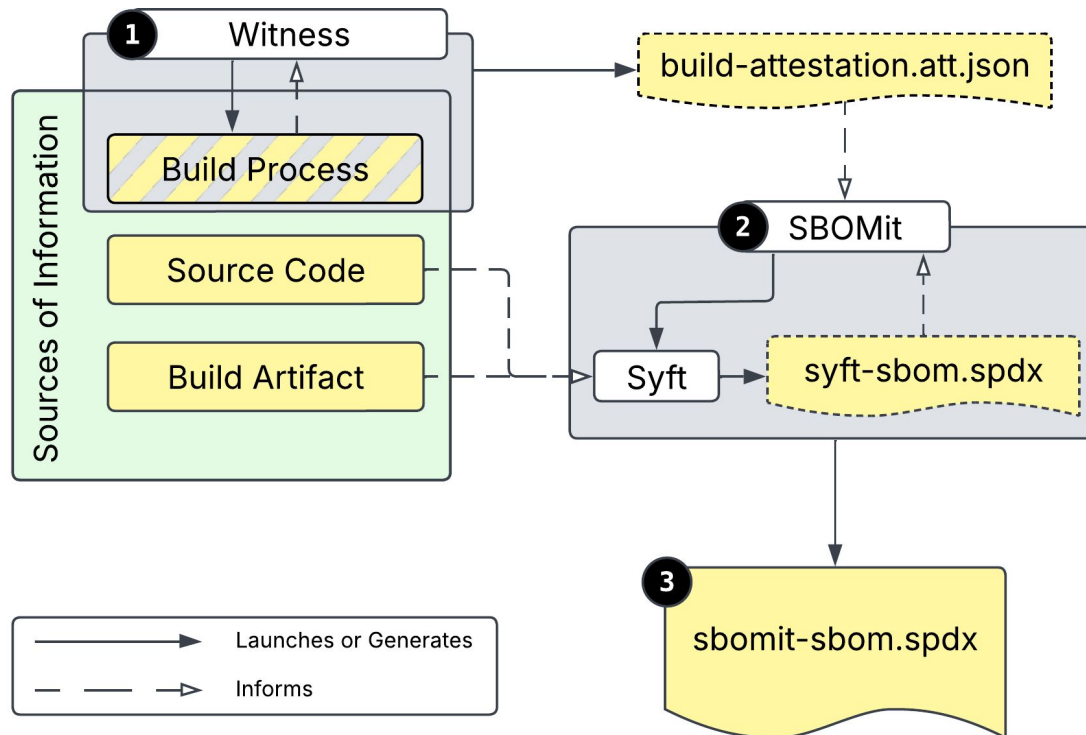


Tool agnostic



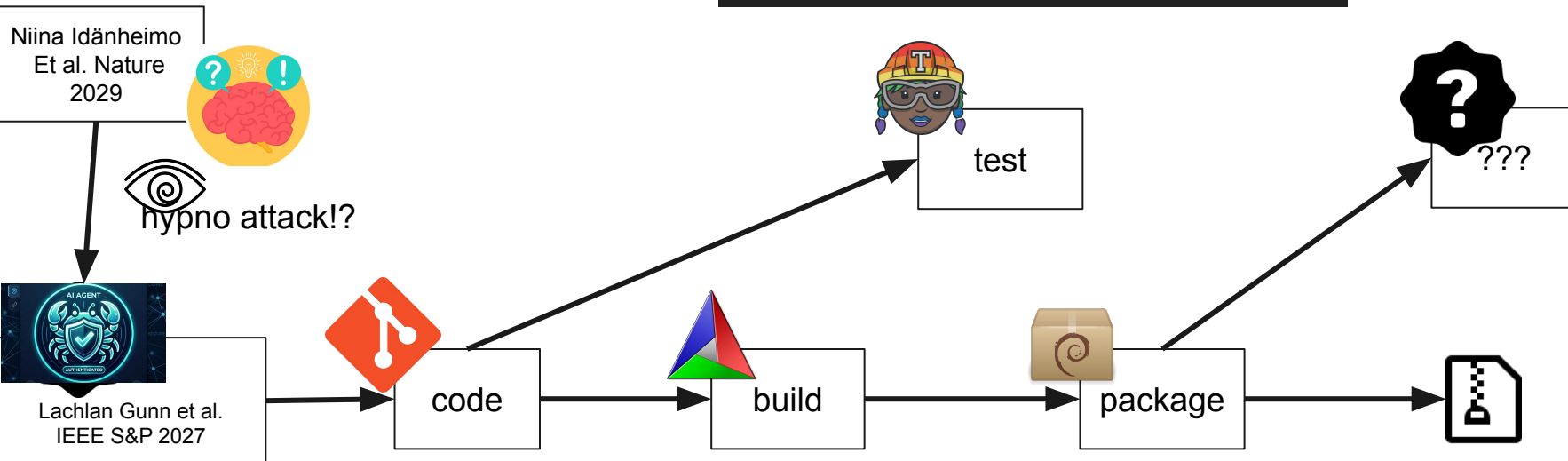


Creating fully-accounted-for builds





All Encompassing





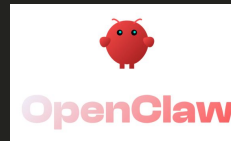
GitLab



DATADOG



In-toto's Impact



CNCF Graduates in-toto, Bolstering Software Supply Chain Security

JUN 23, 2025 • 1 MIN READ

Log in to listen to this article

LIKE



by



Craig Risi

FOLLOW

Software Architect | Game Designer |
Writer | Speaker

Follow us on



Youtube
232K Followers



LinkedIn
26K Followers



Instagram
New



RSS
19K Readers



X
57.1k Followers



Facebook
21K Likes



Bluesky
New

On April 23, 2025, the [Cloud Native Computing Foundation](#) (CNCF) [announced](#) the graduation of [in-toto](#), a framework designed to enforce supply chain integrity by ensuring that every step in the software development lifecycle, such as building, signing, and deployment, is properly authorized and verifiable. This move signifies that in-toto has achieved full maturity and stability, joining other graduated CNCF projects that are widely adopted and ready for production at scale.

[Chris Niszczczyk, CTO of CNCF](#), emphasized its timely impact:

“

in-toto addresses a critical and growing need in our ecosystem, ensuring trust and integrity in how software is built and delivered. As software supply chain threats grow in scale and complexity, in-toto enables organizations to confidently verify their development workflows, reducing risk, enabling compliance, and ultimately accelerating secure innovation.

The graduation follows in-toto's journey from a Sandbox project in 2019 to Incubation in early 2022 to a stable 1.0 specification in mid-2023. It is backed by major US federal agencies, including the [National Science Foundation](#), [DARPA](#), and the [Air Force Research Laboratory](#) have contributed funding and research support. It is already in use by organizations like [Autodesk](#) and [SolarWinds](#), and integrated with standards such as [OpenVEX](#) and [SLSA](#), in-toto is gaining traction across sectors.

RELATED CONTENT

Artifactory Vulnerabilities Under
Exploitation Enable Authentic
Admin Access

SEP 28, 2026

InfoQ Online Cohorts Address AI
Coding Agent Verification

SEP 30, 2026

Cursor Uses S3 WAL to Scale Git
than 300 Pushes per Second

SEP 30, 2026

Context Is the New Code

SEP 30, 2026

Agents Refactor 300K Lines in TH
Practitioners Ask What It Proves

SEP 30, 2026

SvelteKit 3 Reaches Release Can
Config to Vite and Retiring the S

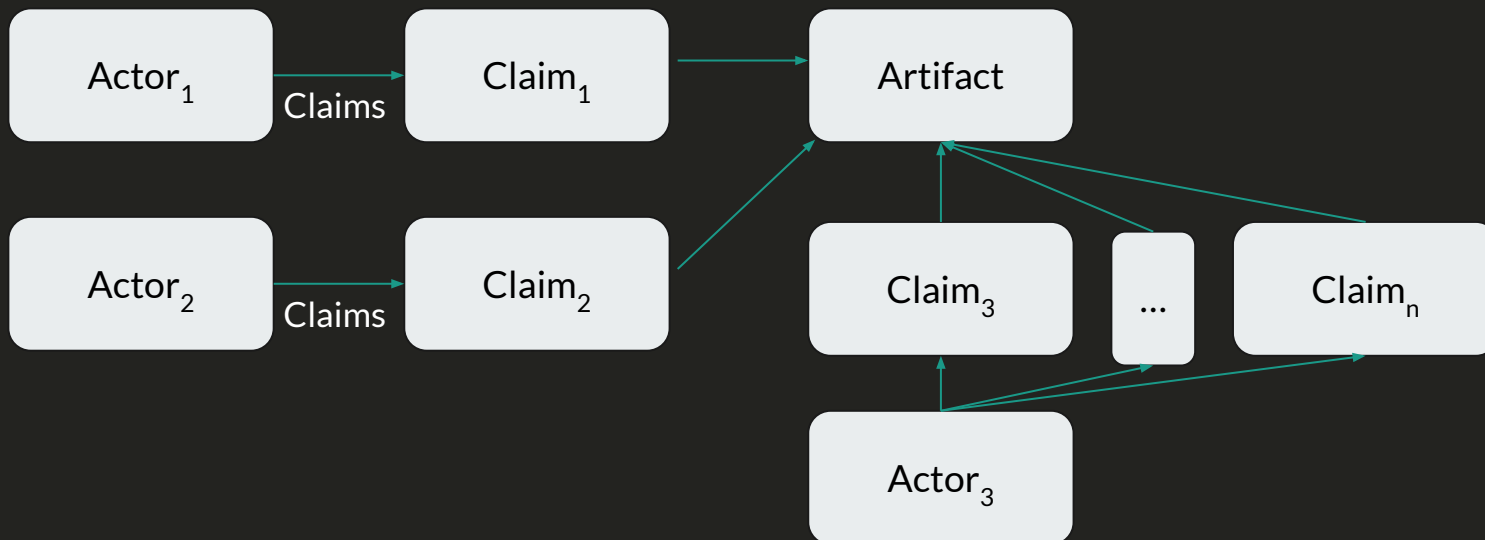
SEP 30, 2026

Amazon CloudWatch Omni Exten
into the Agent Era

SEP 29, 2026

Idea: we can de-compose knowledge about supply chain security to reason about it

As operations are carried out, we learn more about provenance, pedigree, vulnerabilities, etc.



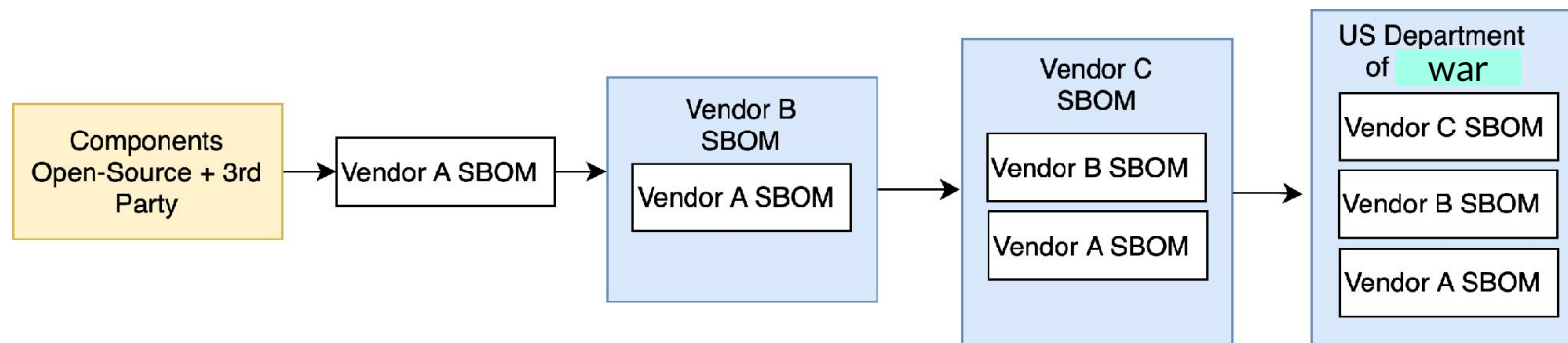


Petra: Confidential SBOM exchange

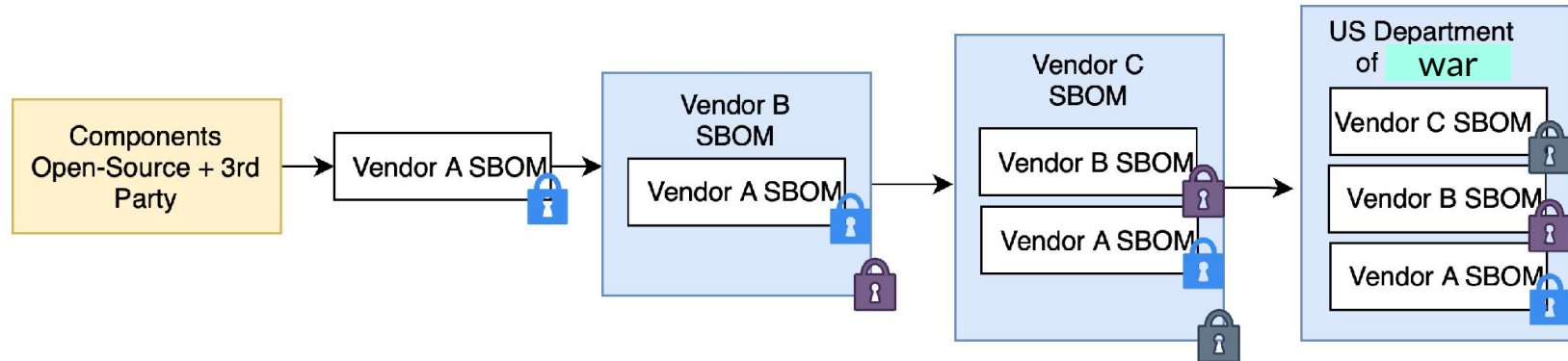
"Trustworthy and Confidential SBOM Exchange", USENIX Security '26

SBOM Exchange Use Case: SBOM Redistribution

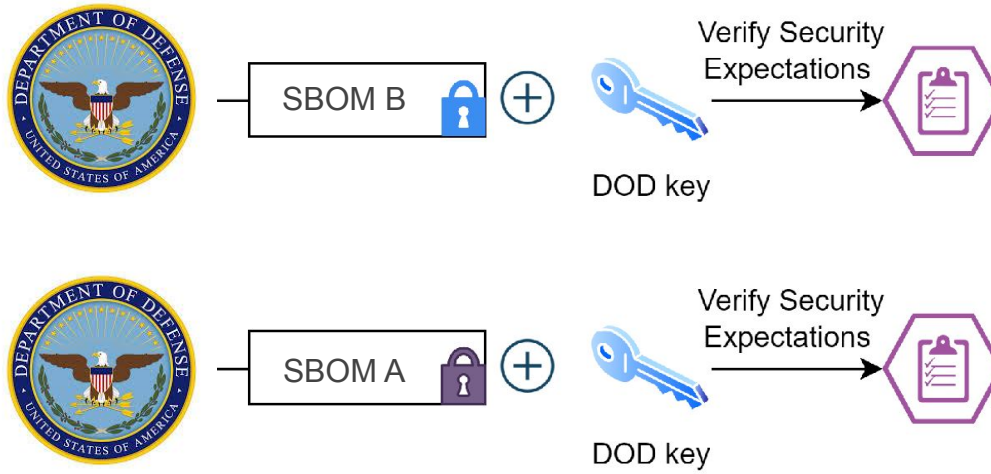
→ Selling software to federal agencies ? SBOM Redistribution is a requirement.



Confidential SBOM Exchange Use Case: SBOM Redistribution



Balancing between Privacy & Transparency



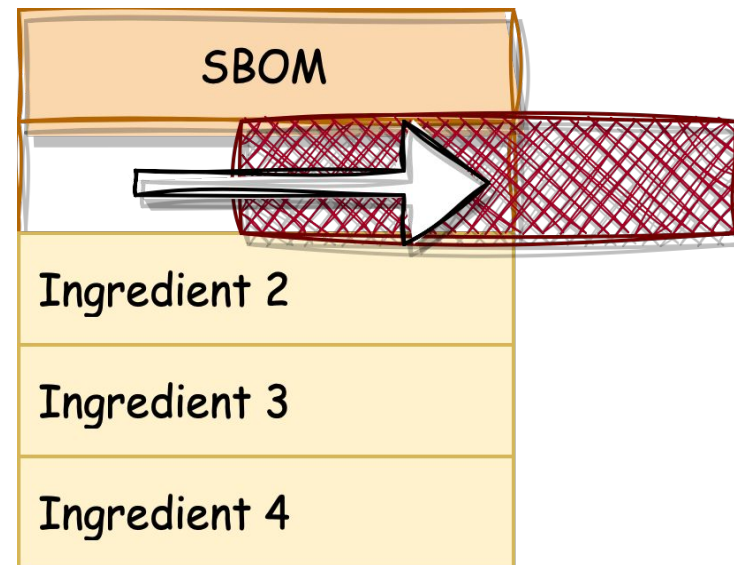
Petra

Selective disclosure 

Verifiable redaction & integrity 

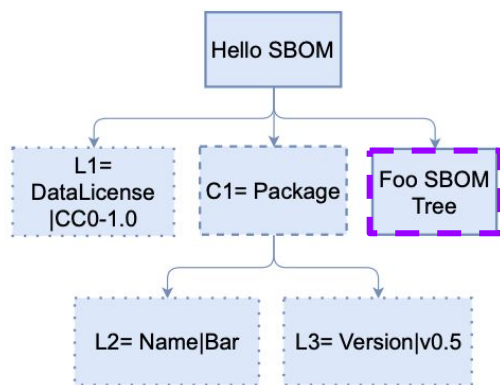
Abstract representation for SBOM consumption 

Practical redistribution system 

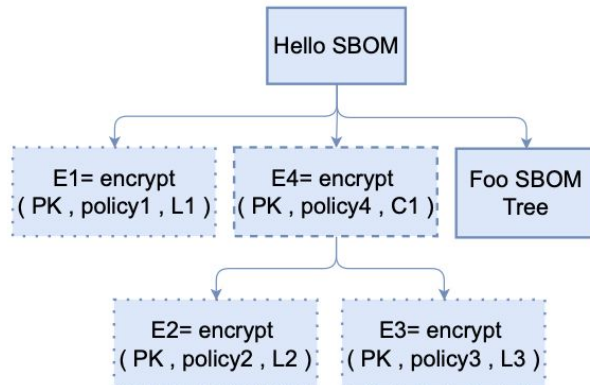


Access control is *bound to the data* !

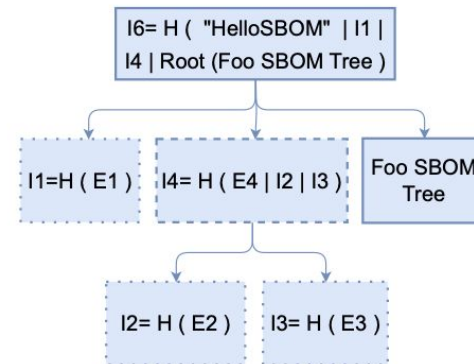
Foundational Techniques (all computing is just trees)



SBOM Trees
(composability)

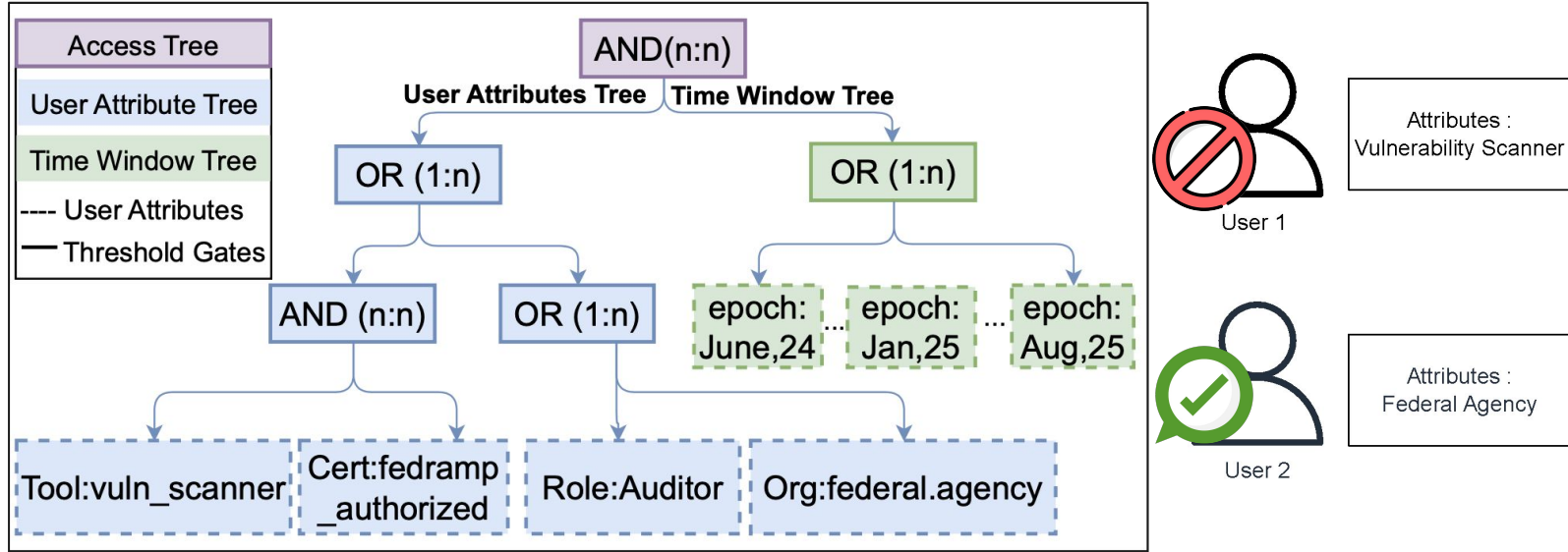


CP-ABE Trees
(selective redaction)



Merkle Trees
(verifiability)

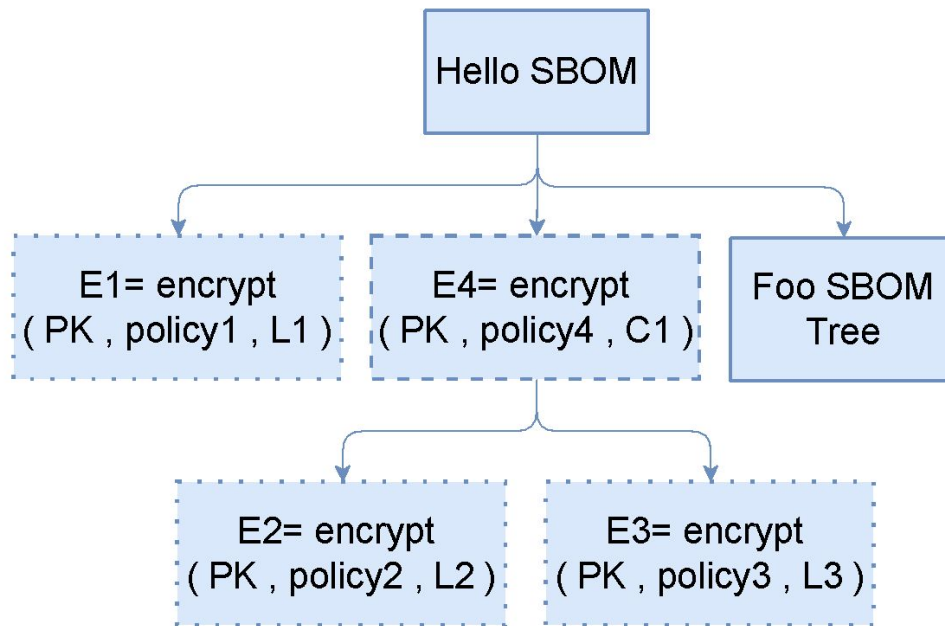
Access Trees



Addresses *Adapting to Different Security Expectations* challenge.

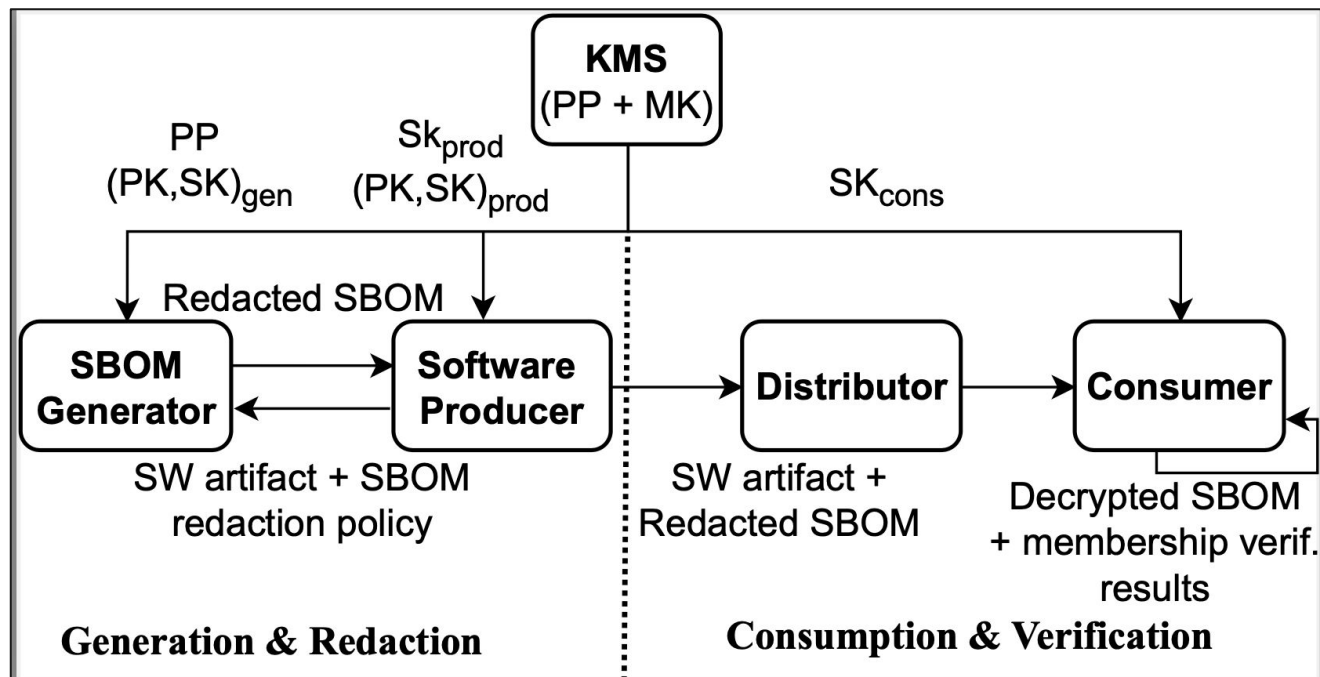
Attribute-Based Redaction

- ❖ **Semantic Confidentiality:** Does not leak contents to parties without explicit access permission.
- ❖ **Collusion Resistance:** Consumers cannot collude to reveal redacted SBOM data

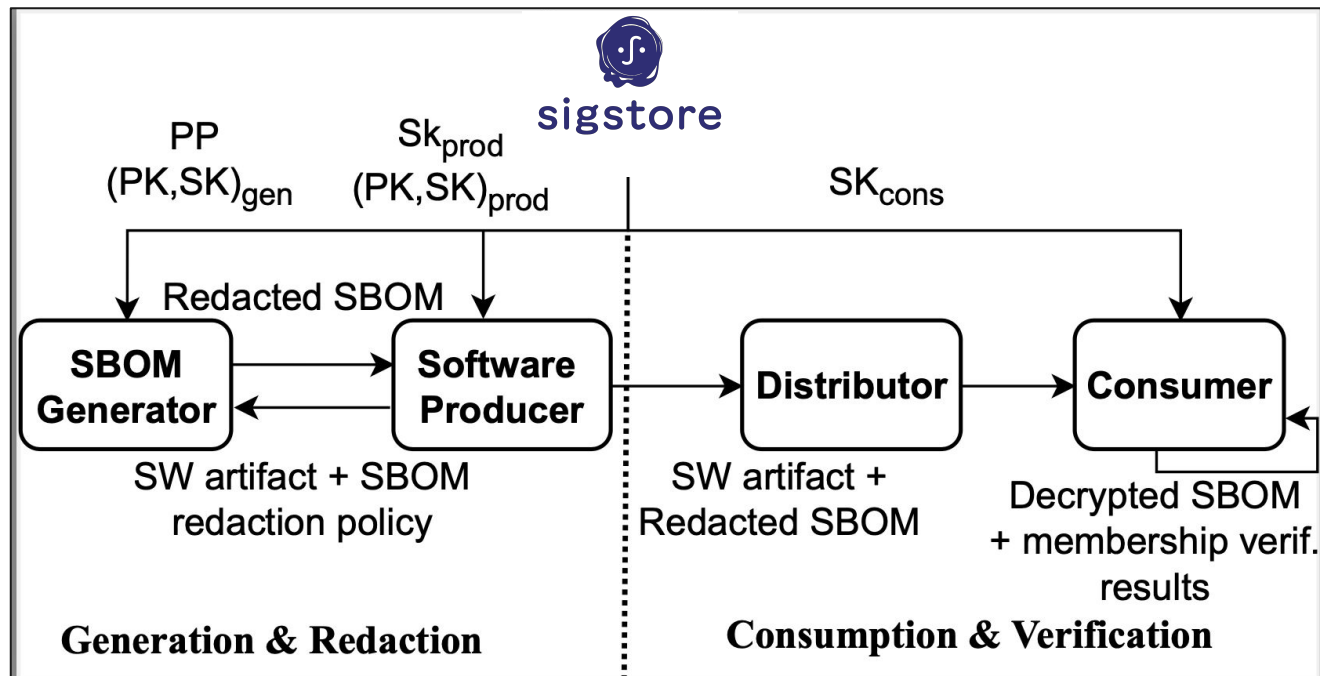


Addresses *Balancing between Privacy & Transparency* challenge.

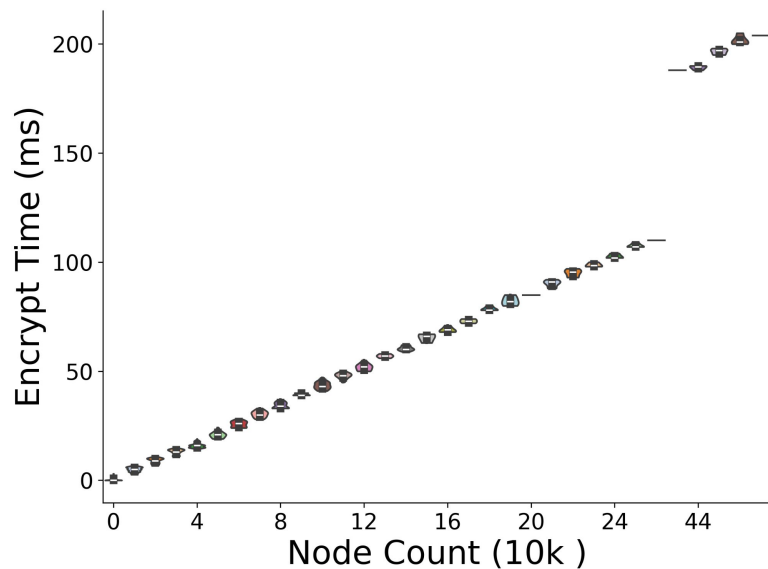
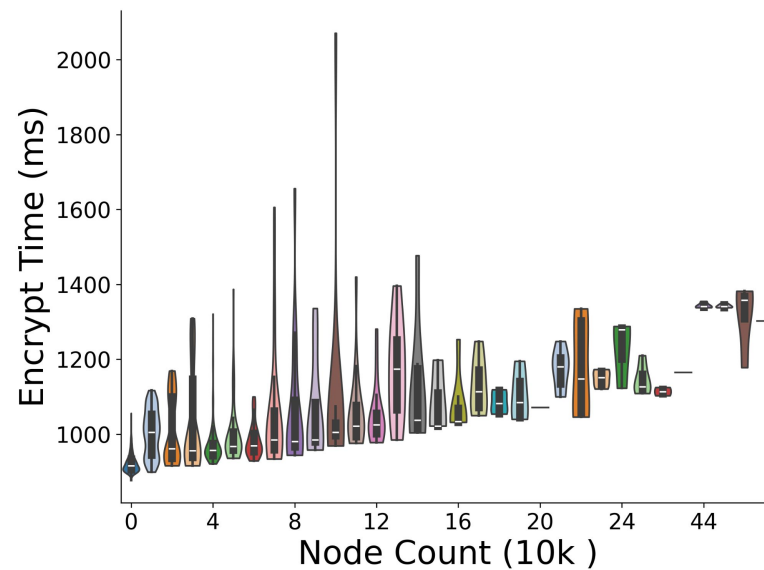
Petra Architecture



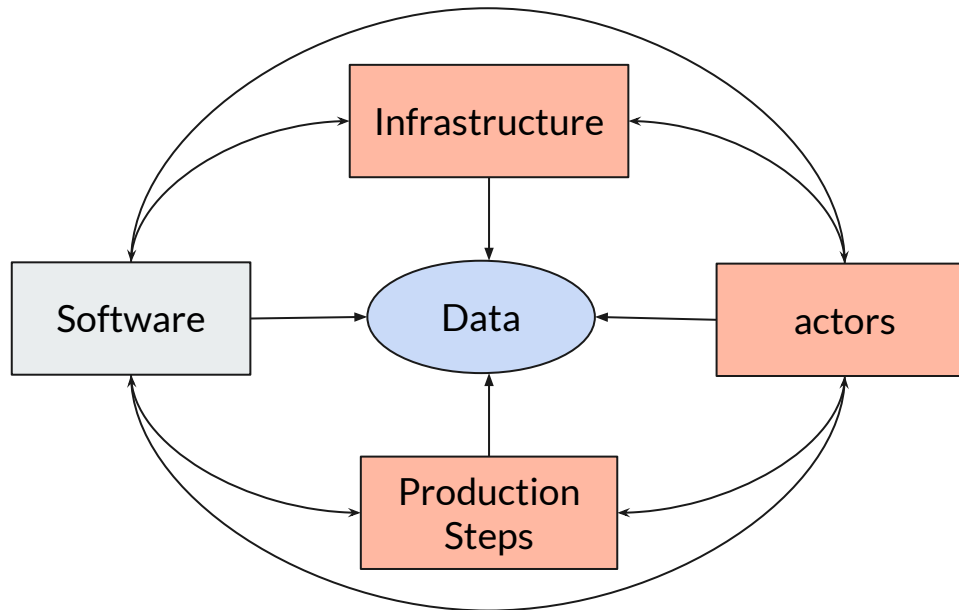
Petra Architecture



Evaluation



What do we need to protect SSCs?

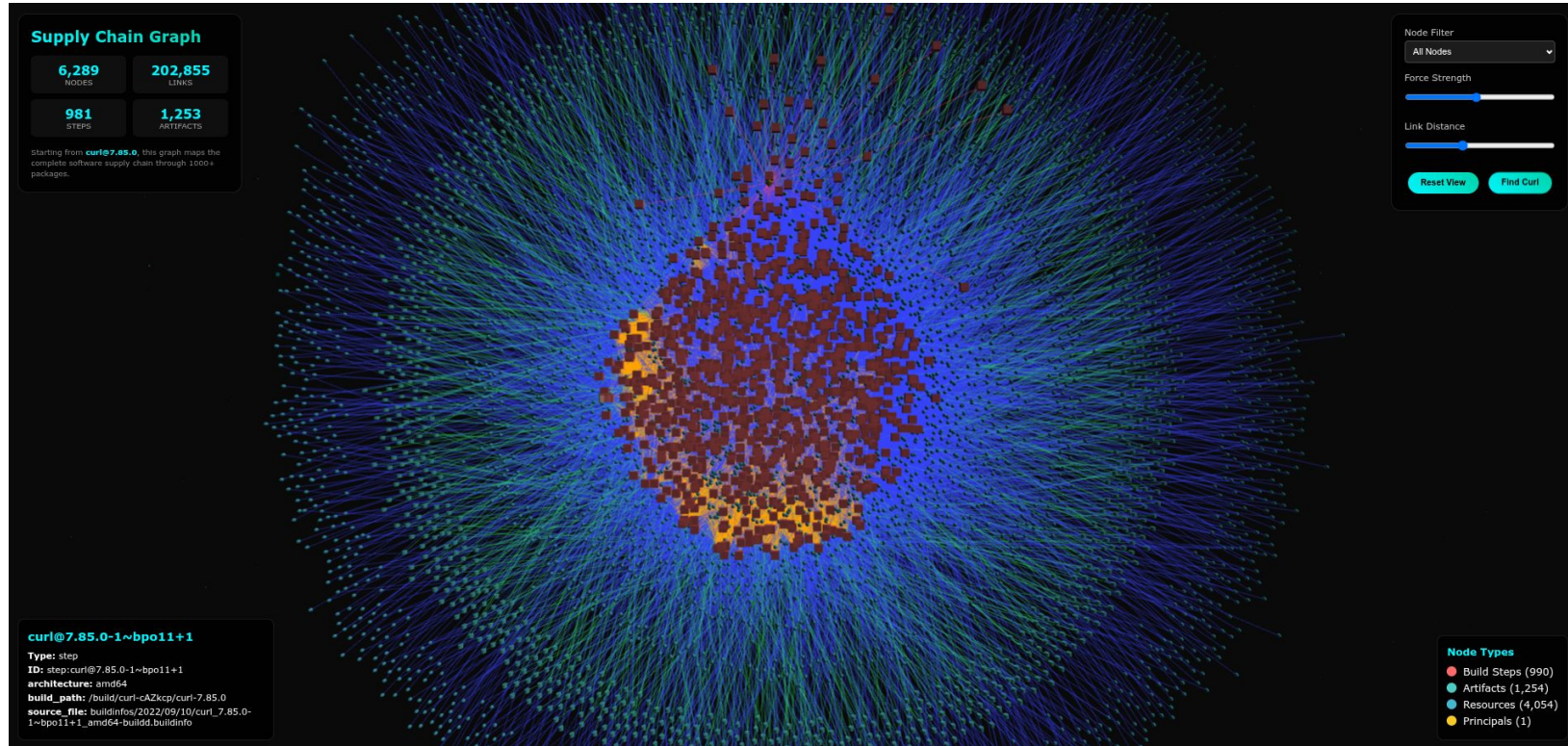




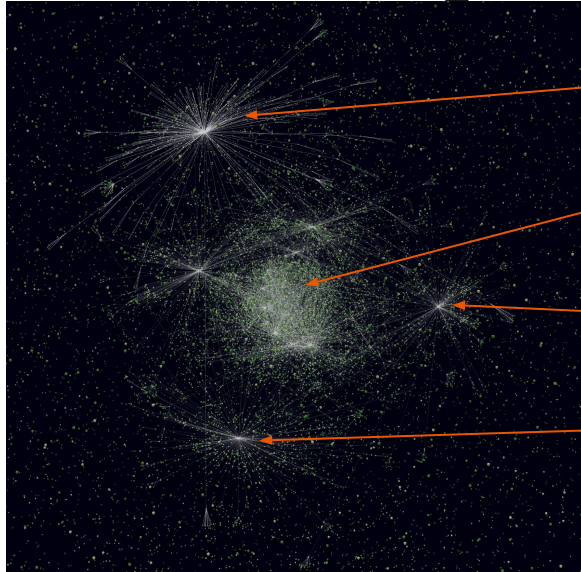
DebTrace

"DebTrace: Tracing, Verifying, and Revisiting a Decade of Debian Reproducibility"
International Conference on Software Maintenance and Evolution (ICSME) '26

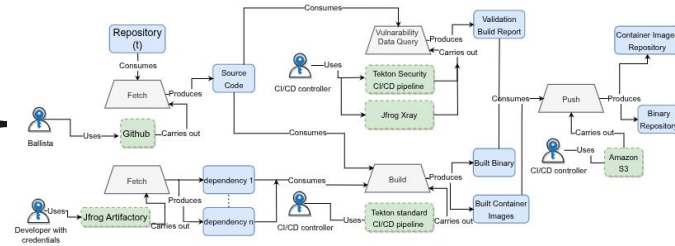
AStRA representation for 981 debian builds



AStRAfy: automate the reconstruction of causal graphs from heterogeneous evidence

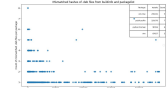
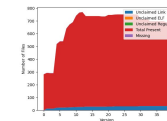


Observe ecosystems
(per ecosystem
mapping frontends)



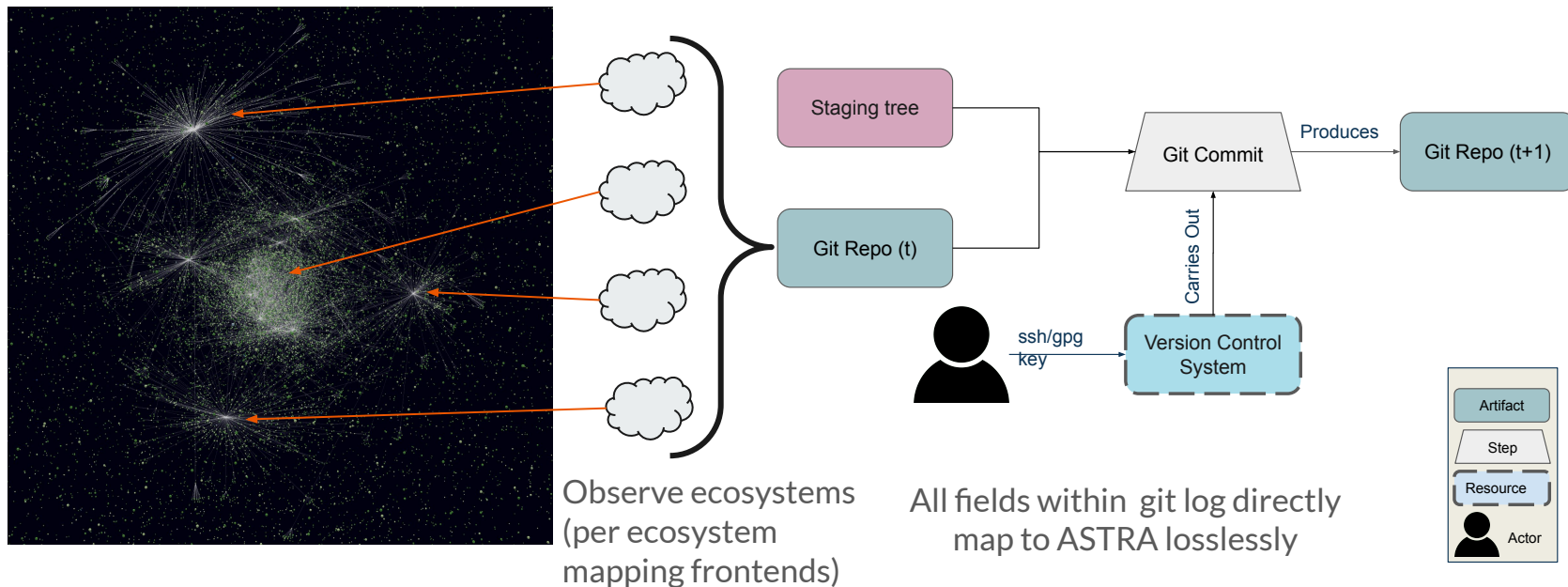
Transpile to ASTRA
Intermediate Representation
(ASTRA IR)

 **SPDX**

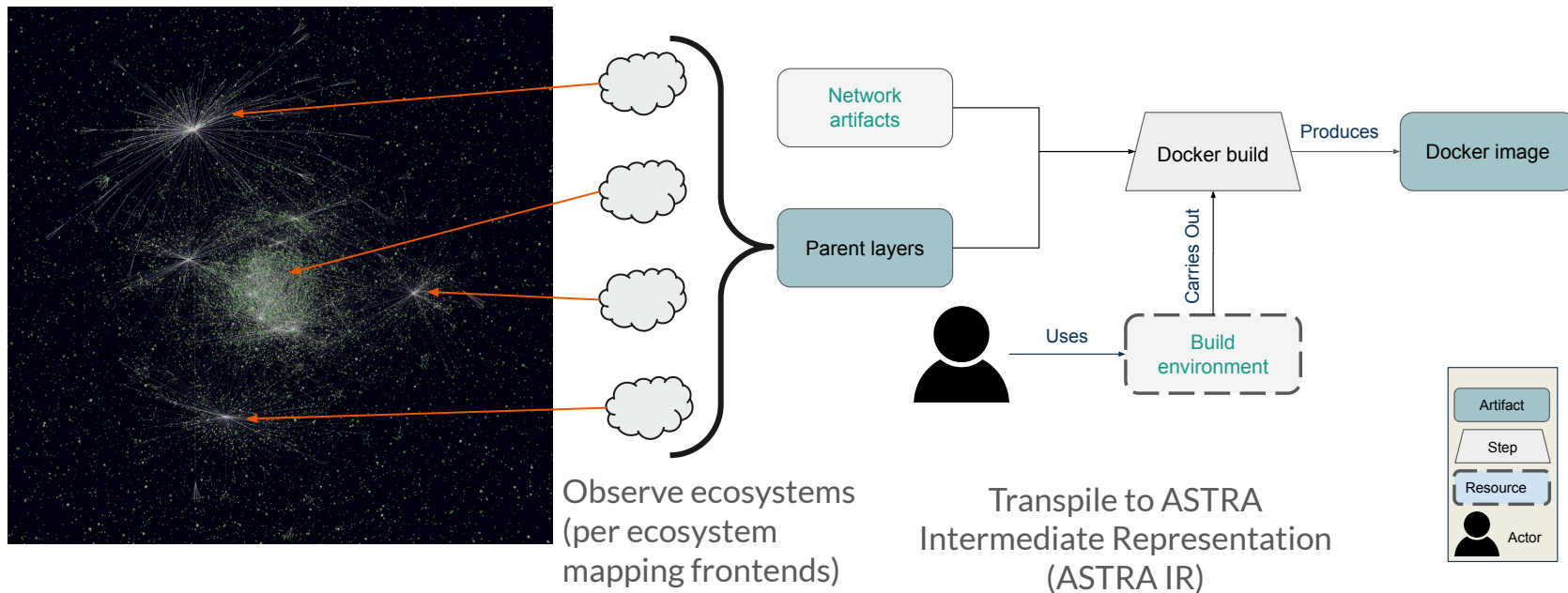


Recode into
actionable data
(backends)

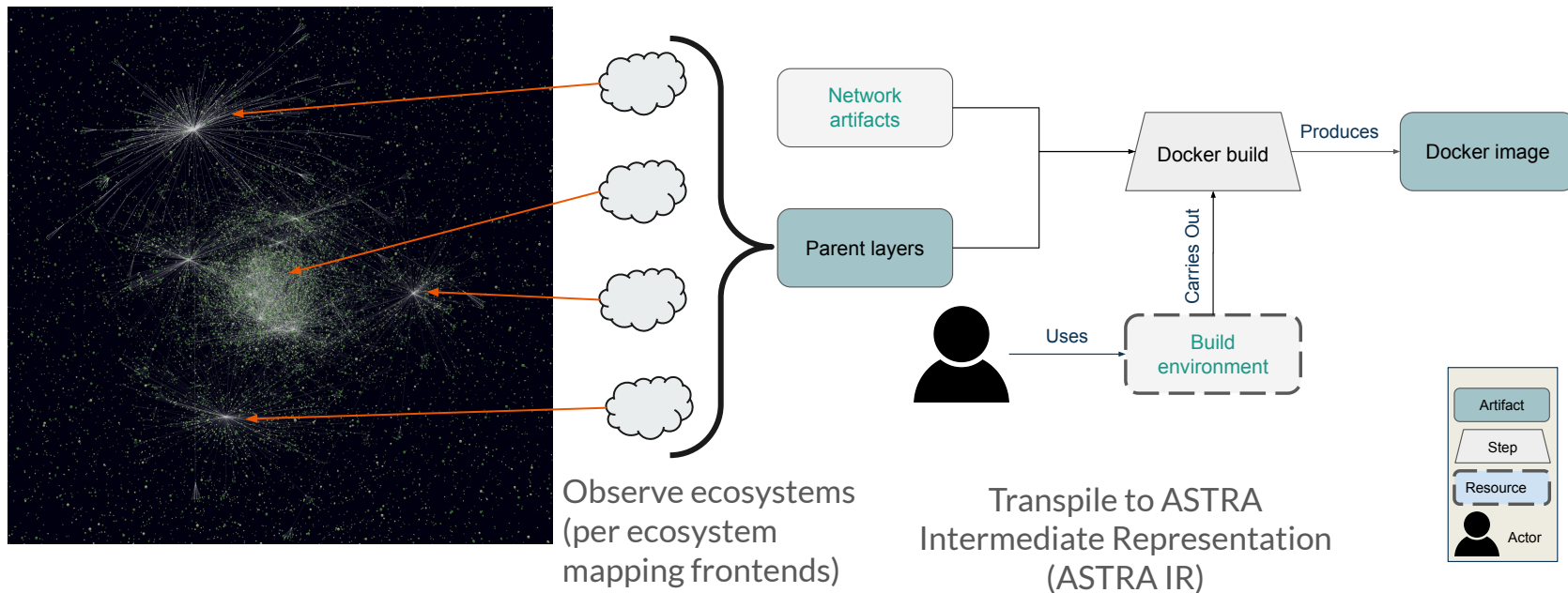
Mapping ASTRA-capable metadata (e.g., Git)



Mapping from flawed metadata (e.g., docker)



Identifying these gaps can help w/ vuln. research



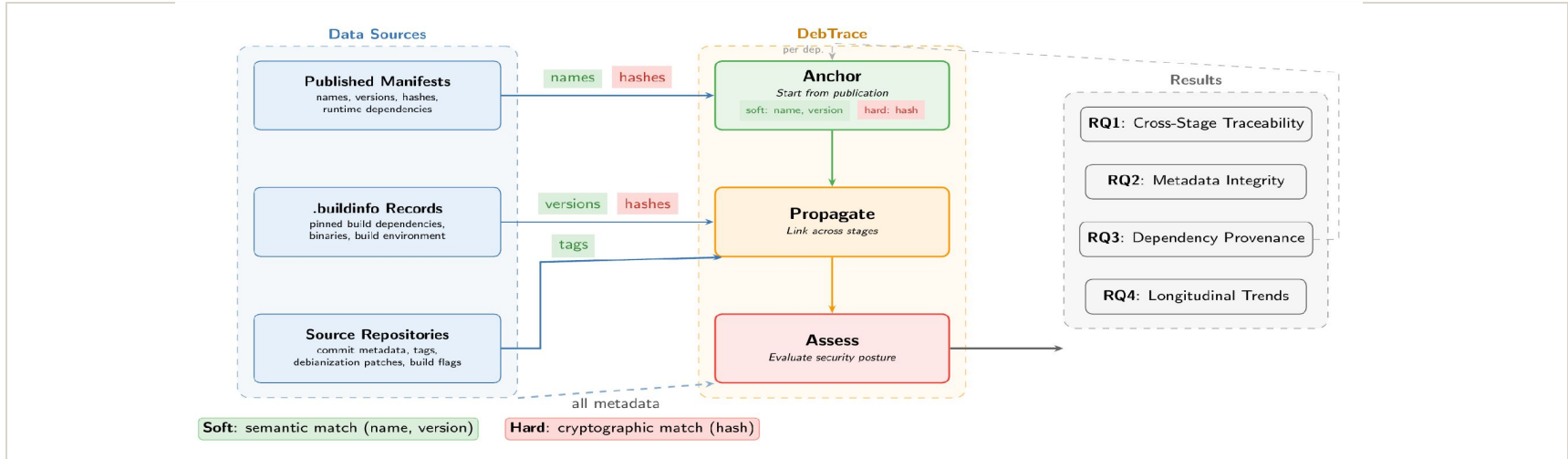
What do graph queries enable?

Risk Indicator	Graph Operation	Analyst Interpretation
Critical dependency	Out-degree / downstream reachability	Artifact affects many downstream elements
Blast radius	Reachability from a node	Compromise or failure may propagate broadly
Concentrated authority	Principal out-degree	One principal is associated with many steps or artifacts
Central resource	Resource in-degree / betweenness	Resource is reused across many paths or mediates many operations
Deep transitive risk	Longest path / path depth	Long dependency chains increase exposure and analysis complexity
Broker node	Betweenness centrality	Node mediates many supply chain paths
Single point of failure	Cut-vertex or articulation analysis	Removing a node disconnects parts of the graph
Missing provenance	Completeness query	Node lacks direct supporting evidence
Disjoint release path	Path query over commits, tags, and publish steps	Published artifact is not connected to an expected release tag



The DebTrace framework

Anchor on what users install, propagate identifiers, assess risk.



1 · Anchor on publication data: only installable packages carry consumer risk

2 · Propagate via soft identifiers (name, version: semantic) and hard identifiers (hashes: bit-for-bit)

3 · Assess Together these form **compositional risk maps**, impossible when stages are studied in isolation.

APPROACH

The dataset

A comprehensive cross-stage dataset of any Debian package.

900,000+

published (package, version, arch) tuples

80,000+

packages

2M+

.buildinfo records · Dec 2016 to Jan 2026 · amd64, arm64, all

4 TB

source repositories

~6 h

archive snapshot cadence

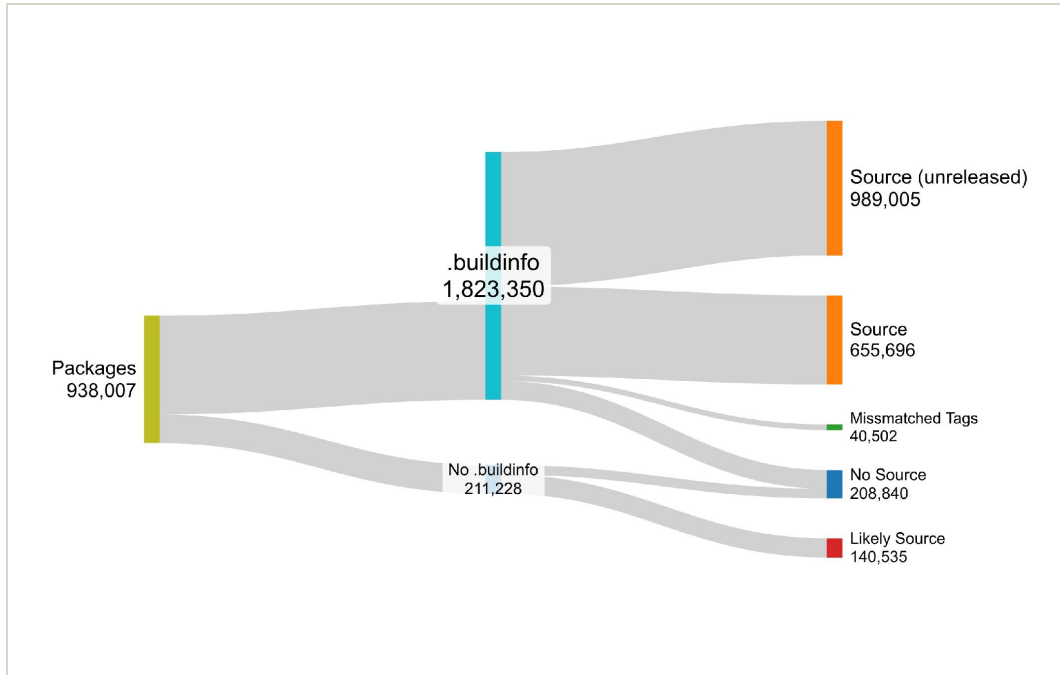
~30 GB

structured database

github.com/wboulton/debtools

Only ~70% of published packages are fully traceable

Compared to 97% step-local reproducibility



89%

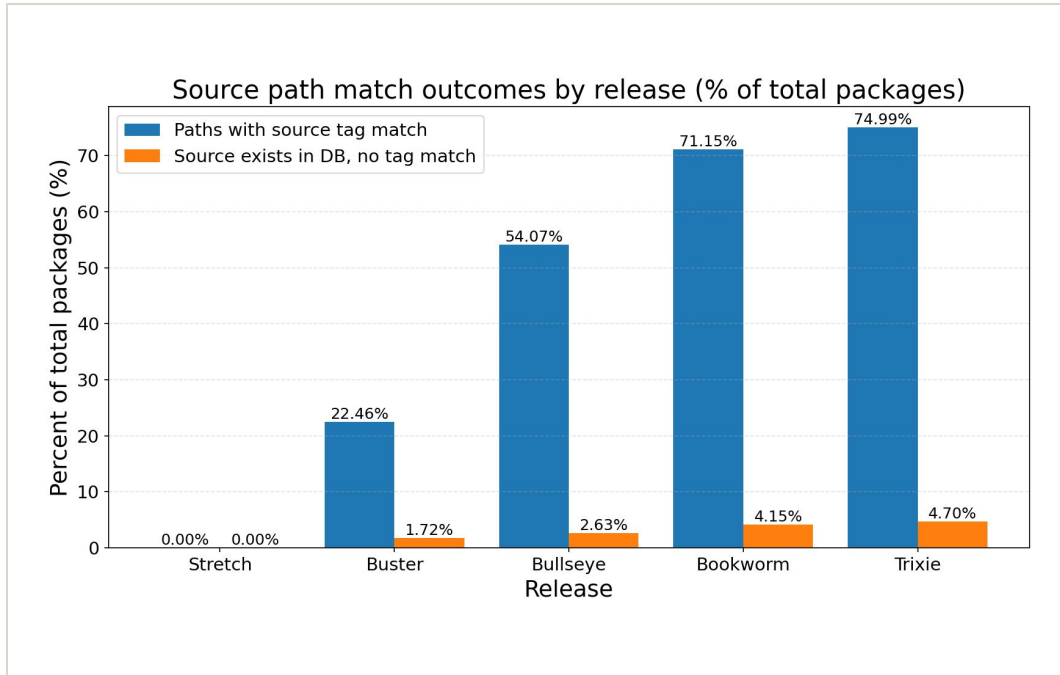
have a full publication → build → source repo

70%

also have a source tag matching the publication version: **fully traceable**

A decade of progress

Traceability rose from near zero to ~75%, driven by `.buildinfo` adoption.



~0% → 74.99%

Stretch, 2017

Trixie, 2025

The tag-match gap has been stable at **3 to 5%** since Buster. Overall repo availability improved but tagging discipline did not.

The 70% headline on earlier slide refers to the latest release

*The security pipeline doesn't publish
.buildinfo through the standard channel*

The packages you most need to verify are the least traceable.

8,128 unmatched records carry security-release markers (+debNum).

The debian build-system has (had) a race condition

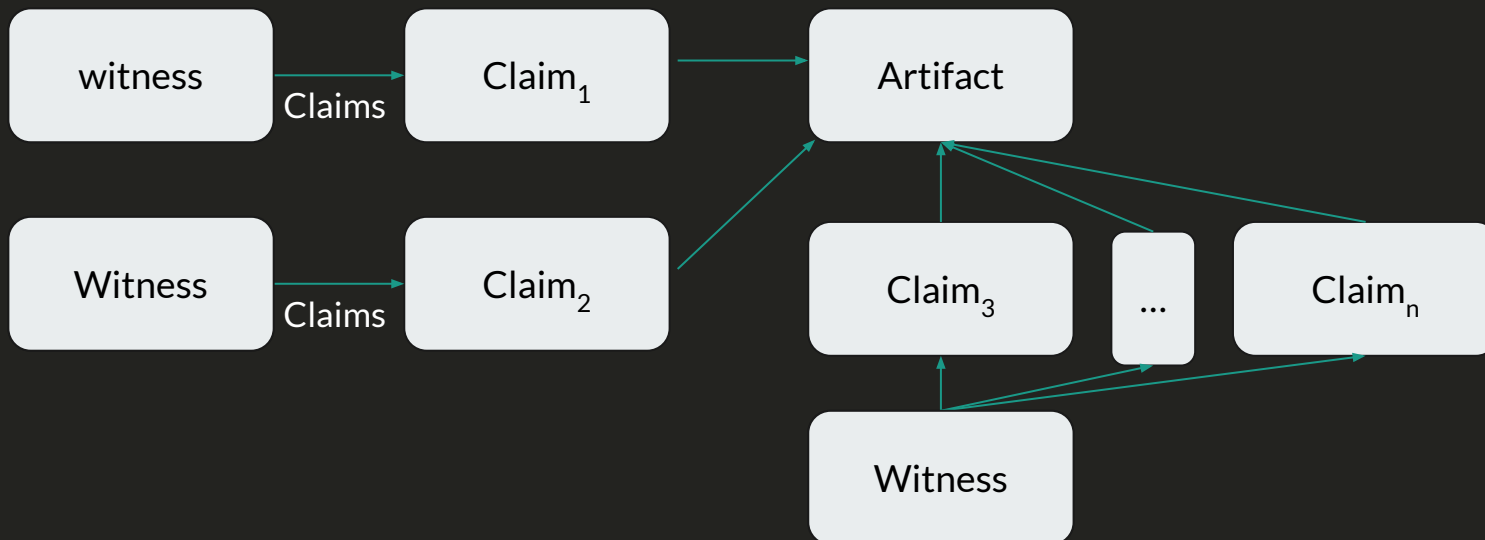
*3.7% of packages were built, used as dependencies of **other** builds, but never released*

These packages are also impossible to audit



Idea: we can decompose any supply chain data into auditable, causal, intermediate representations

As operations are carried out, we learn more about provenance, pedigree, vulnerabilities, etc.

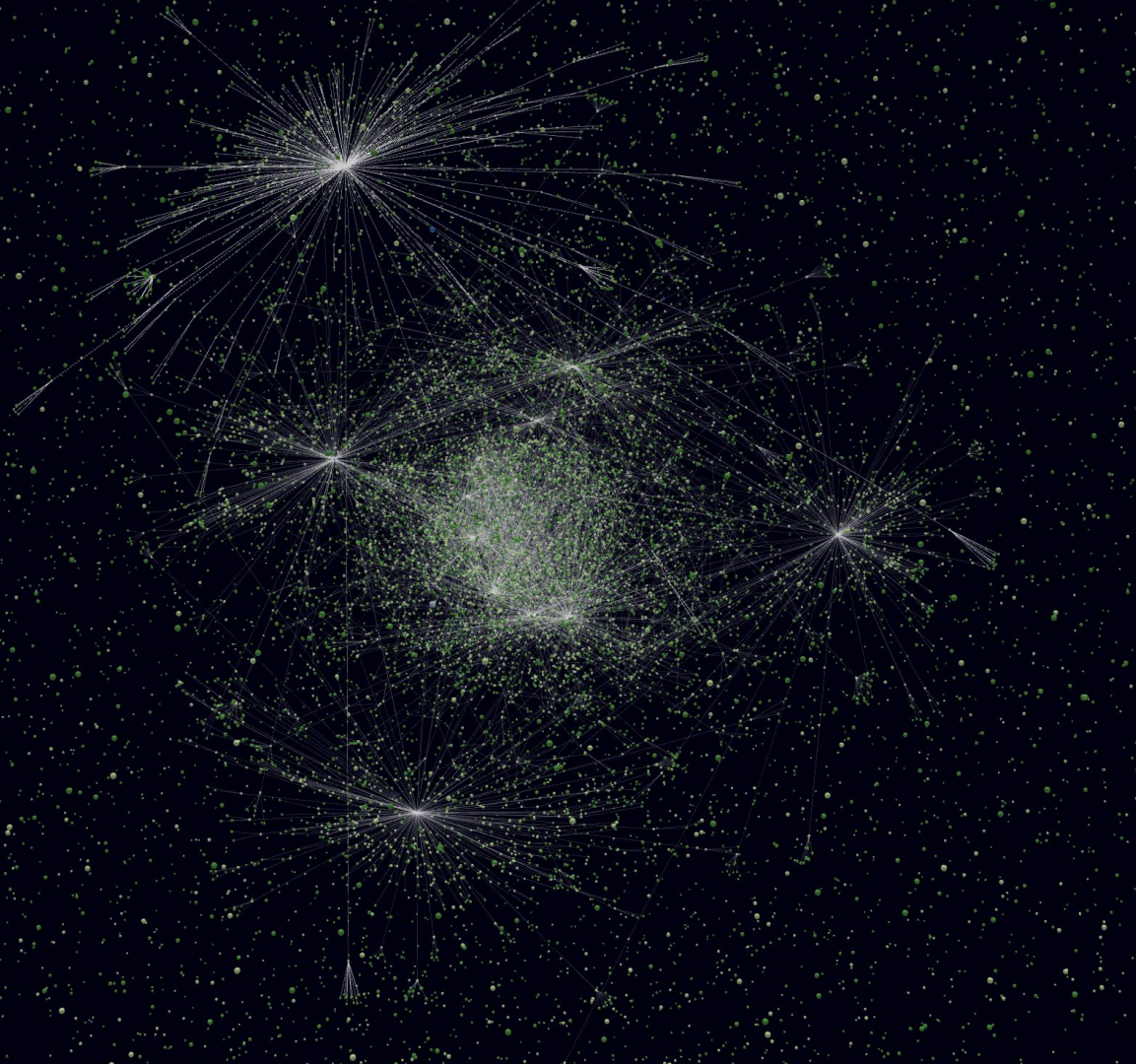


Idea: w
in

As o



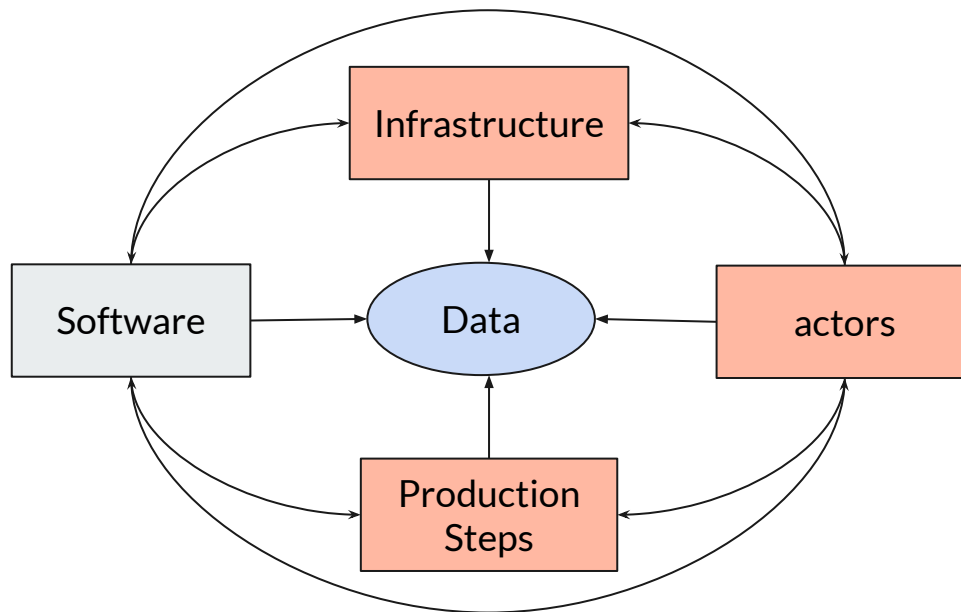
TRUSTWORTHY SOFTWARE
ECOSYSTEMS LAB



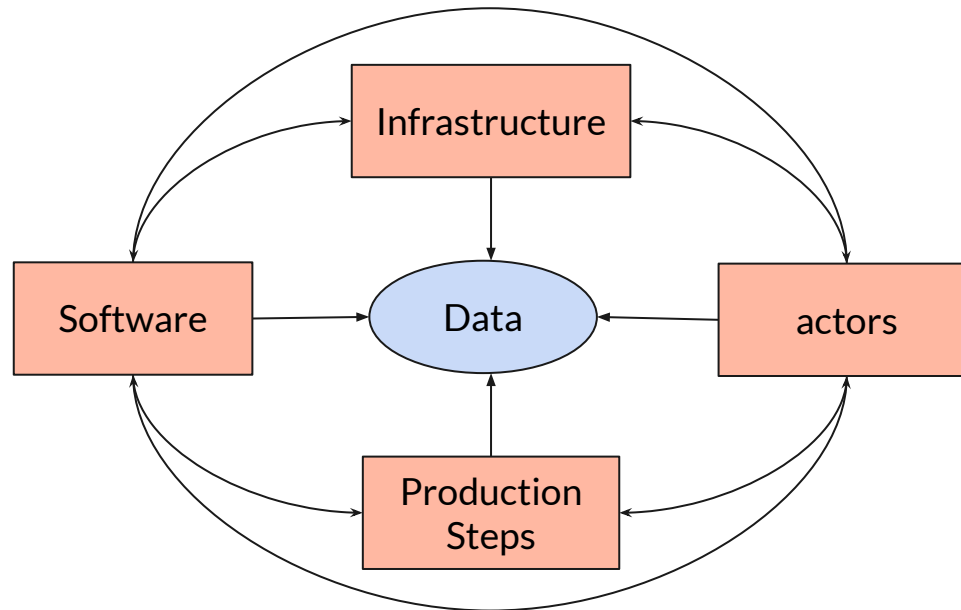
s, etc.

Claim_n

What do we need to defend SSCs?



What do we need to defend SSCs?

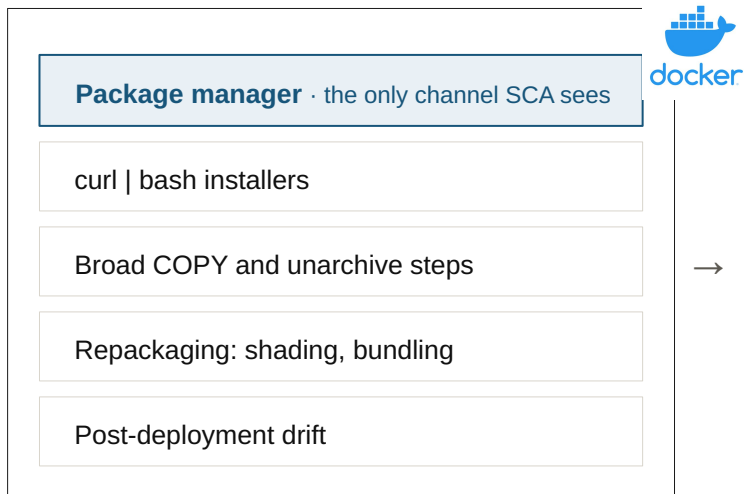


Darkfiles

"Software Dark Matter: Gazing at Uncharted Files to Navigate SBOM Integrations",
Supply Chain Offensive Research and Ecosystem Defenses (SCORED) '26

Why SBOMs miss files

Artifacts are assembled through channels that no manifest describes.



17 to 332

vulnerable dependencies reported for one application across nine SCA tools (Imitaiz et al.). Tools infer presence from package databases and heuristics.

$$SDM = Files_{actual} \setminus Files_{asserted}$$

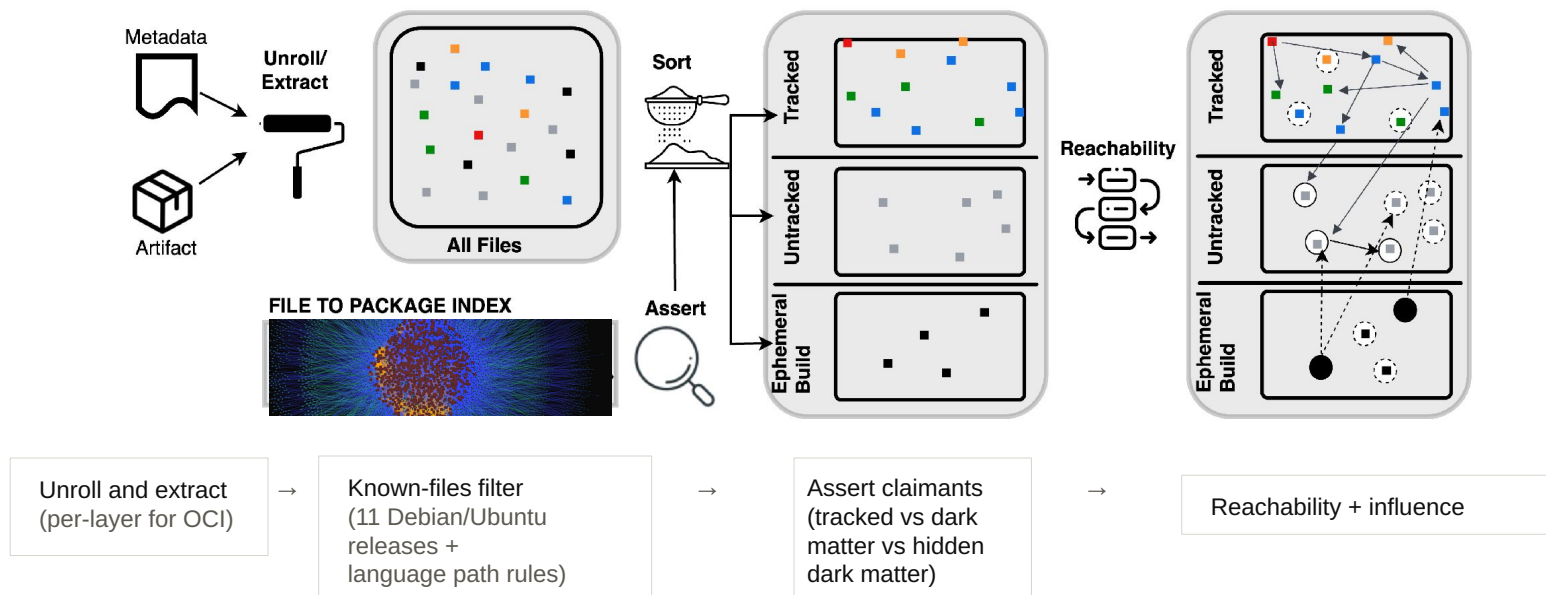
SDM is the set of files physically present that no metadata source claims.

Four ecosystems: Questions

	PREVALENCE	SECURITY IMPACT	CAUSAL ORIGINS	TEMPORAL ORIGINS
Containers image-based				
Maven Central library-based				
Plugins & extensions host-mediated				
Enterprise Linux config-driven				

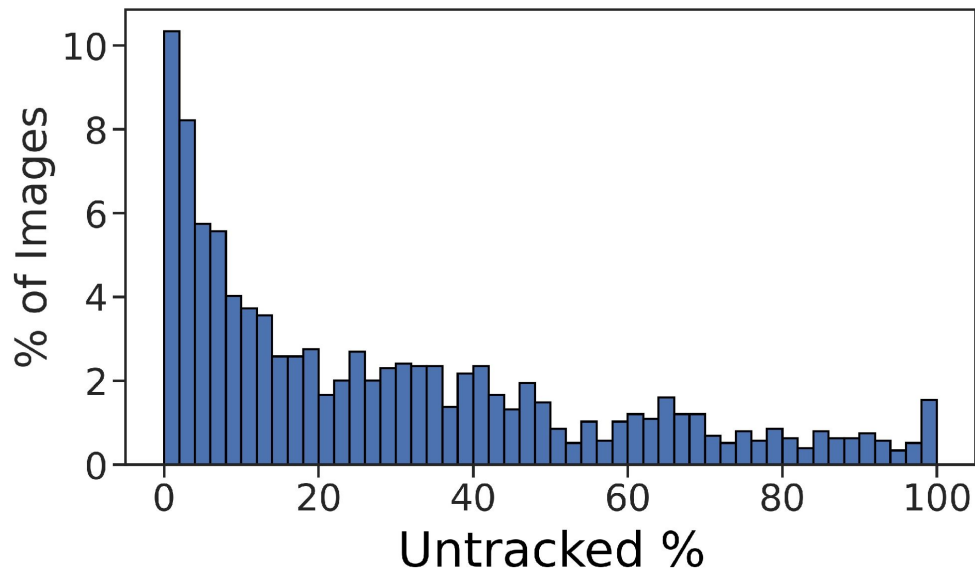
The Darkfiles pipeline

One **simple** workflow generalizes across disjoint ecosystems with policy-level configuration only.



Reported prevalence is a conservative, policy-driven lower bound.

Prevalence - Containers



30%

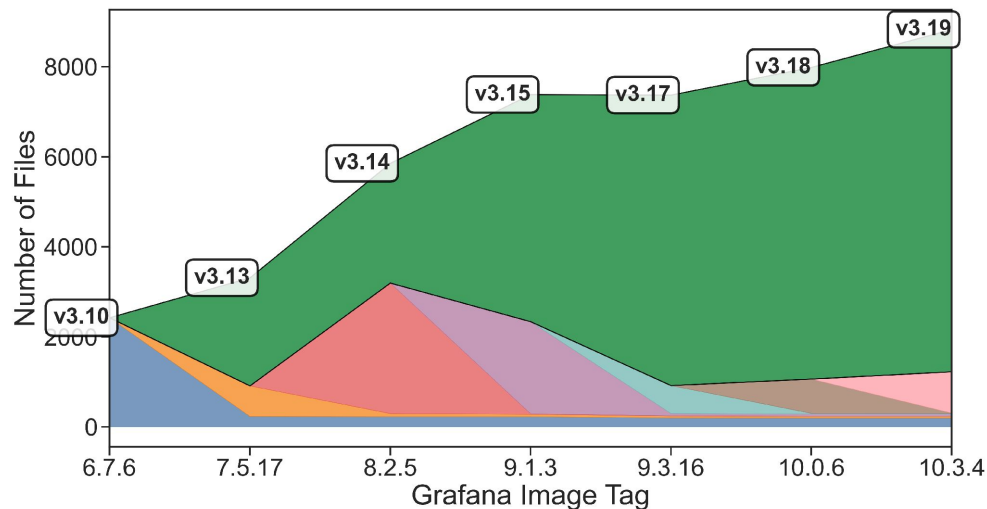
mean SDM per image

~22%

of images under 5%

Temporal Origins - Containers

Metadata never converges with the artifact.



Security Impact: Containers

Reachability narrows triage set to 1% of files on average

RECURRING DARK FILE	IMAGES	SECURITY IMPACT
/opt/bitnami/scripts/liblog.sh	63	Tampering can suppress logs or inject startup commands
/usr/bin/with-contenv	30	s6 env injector; can alter PATH and redirect exec before the main service
/etc/supervisord.conf	28	Can start extra daemons or change arguments and logging
/usr/sbin/enable_insecure_key	13	A reachable path to downgrade trust and authentication
/etc/ssh/ssh_host_rsa_key	7	SSH host private keys baked into images: impersonation and MITM

Case 1: Jenkins ssh-agent

Untracked files were an exploitable vulnerability in core CI infrastructure.

INTENDED

Host keys generated **at first run** , unique to every container instance.

VS

SHIPPED

`apt install openssh-server` generated keys **at build time** . Identical keys reused across every instance:

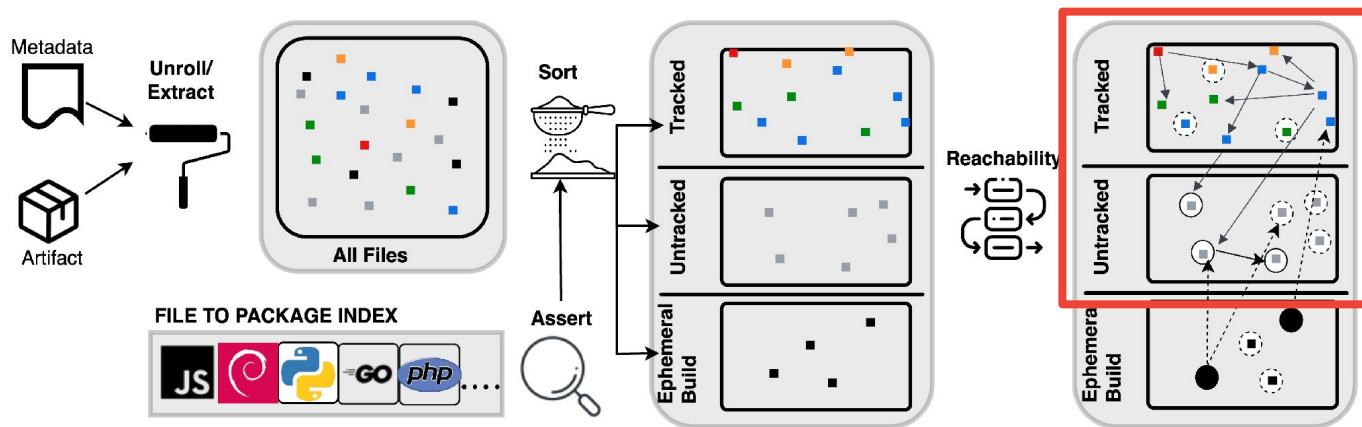
CVE-2025-3275

CVE-2025-3275

Coordinated disclosure · patched · official Jenkins advisory

Case 1: Jenkins ssh-agent

Untracked files were an exploitable vulnerability in core CI infrastructure.



CVE-2025-3275

CVE-2025-3275

Coordinated disclosure · patched · official Jenkins advisory

Case 2: The acme.sh token

A GitHub push token persisted in intermediate layers.

Layer N: final image · token absent (whiteout)

Layer N-1: token still present · recovered by deep-darkfiles

`COPY ./ . #swept .git/config` (with token) into the image

CVE-2025-3211

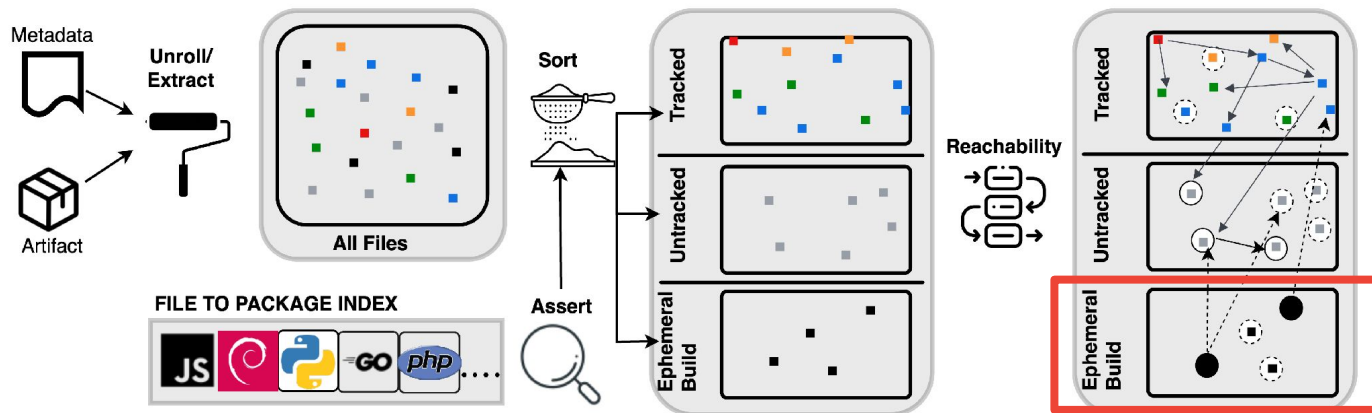
CVSS 8.7 · High

acme.sh: a project with **47,000+ GitHub stars**.

The token allowed CI triggering, source tampering, and release modification, which allows **decrypting a large portion of internet traffic**.

Case 2: The acme.sh token

A GitHub push token persisted in intermediate layers.



CVE-2025-3211

CVSS 8.7 · High

Security Impact: Maven Central

4,889

bundling-prone packages

~80% hide under 10% of their dependencies; **10%** hide over 90%.

5000+

untracked JAR instances at versions with
known CVEs

HOW THEY GET THERE

maven-shade-plugin

maven-assembly-plugin

Spring Boot nesting with spring-boot-maven-plugin

Manually copied vendored JARs

Security Impact: Jenkins Plugins

2000+ plugins analyzed, 1,891 with nested JARs.

Mean untracked 10.0%

648

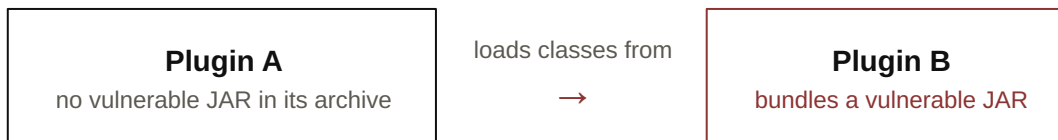
vulnerable JAR instances in 381
plugins $\approx$ 200,000 installations

57

plugins declare a safe version
while shipping a vulnerable JAR

**54.6% of vulnerable instances are invisible or
misleading to SBOM scanners.**

Security Impact: Jenkins Plugins



+80

additional vulnerable JAR instances across 48 plugins

33 of 84

CRITICAL jars are through plugins labelled
“deprecated”, “adopt-this-plugin” that remain
depended upon, with no maintainer to remediate

Security Impact: OpenVSX extensions

Bundling destroys attribution, and what can be see is already bad.

- Under 2% populate bundledDependencies
- VSIX manifest records no dependency info at all.
- We could only analyze small % of VSX extensions

1,259

CVE-matching packages in 187 extensions · 132 CRITICAL, 619 HIGH

BUILD-CONTEXT BLEED-IN FOUND IN PACKAGES

.claude/ and .copilot/ session data
.env · .ssh/ · .git/ histories

All figures are conservative lower bounds: minified bundles cannot be attributed.

Software Dark Matter shows up everywhere

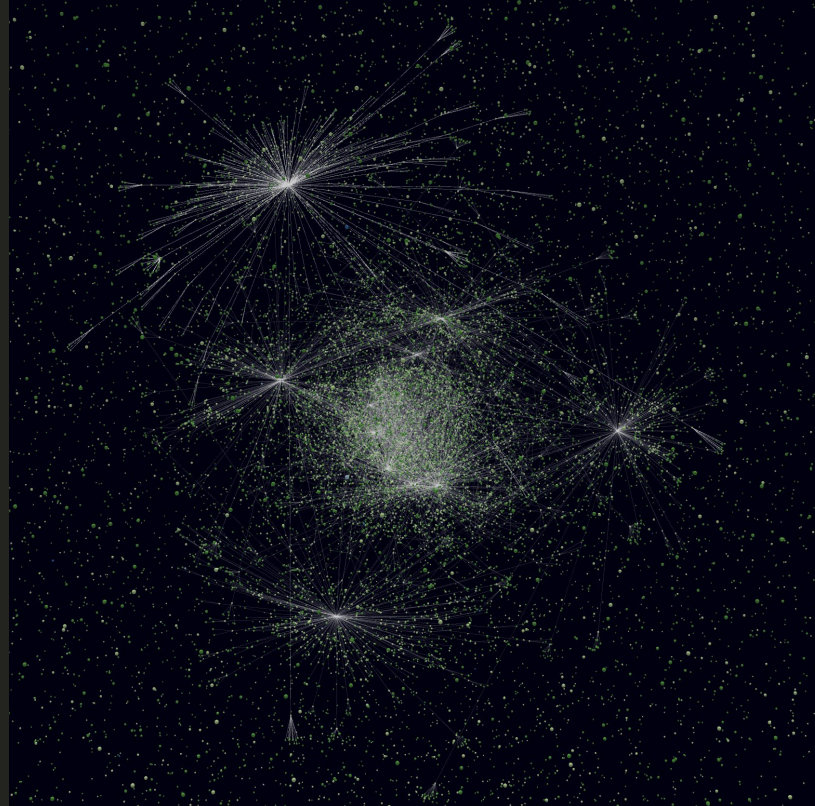
	PREVALENCE	SECURITY IMPACT	CAUSAL ORIGINS	TEMPORAL ORIGINS
Containers image-based	Most images hide some, a few hide a lot	3 high-severity vulnerabilities, previously unknown	ADD and COPY layers	Metadata never catches up
Maven Central library-based	Bimodal: none, or nearly all	Hundreds of CVEs missed by SCA tools, including widely used projects	Nesting and shading	Stable over time
Plugins & extensions host-mediated	Obfuscation hides it; source maps recover it	~1,000 CVE matches invisible to SBOMs, plus leaked credentials	Bundling and vendoring	Not representative
Enterprise Linux config-driven	—	Unused but executable scripts	IT automation and config management	—

Software Dark Matter shows up everywhere; *it is systemic!*

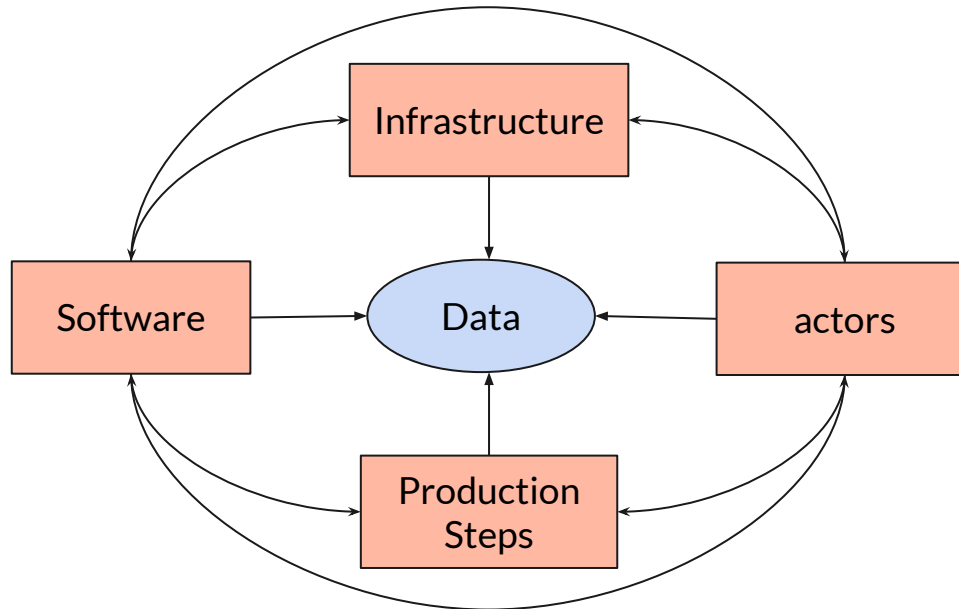
	PREVALENCE	SECURITY IMPACT	CAUSAL ORIGINS	TEMPORAL ORIGINS
Containers image-based	Most images hide some, a few hide a lot	3 high-severity vulnerabilities, previously unknown	ADD and COPY layers	Metadata never catches up
Maven Central library-based	Bimodal: none, or nearly all	Hundreds of CVEs missed by SCA tools, including widely used projects	Nesting and shading	Stable over time
Plugins & extensions host-mediated	Obfuscation hides it; source maps recover it	~1,000 CVE matches invisible to SBOMs, plus leaked credentials	Bundling and vendoring	Not representative
Enterprise Linux config-driven	—	Unused but executable scripts	IT automation and config management	—



Idea: A global artifact inventory can be used to surface SSC attacks & vulnerabilities on the fly



What do we need to defend SSCs?



Thank You! Questions?



<https://badhomb.re/>

<https://tsel.sh>

santiagotorres@purdue.edu

sangy@chaos.social

10:40
CEST

Ghost In The Codebase - Why Your LLM Needs a SCA/SAST Babysitter - Luca Galli, Open Systems AG

South Hall 3A
Luca Galli

10:55
CEST

Preparing for the Vulnpocalypse: Using OSS-CRS To Find and Fix Bugs Before They Find You - Jeff Diecks, OpenSSF & Laura Guazzelli, Linux Foundation

South Hall 3B
Jeff Diecks • Laura Guazzelli

11:02
CEST

Trusting-Trust Attack against an Entire Linux Distribution through Binary Manipulation - Aman Sharma, KTH Royal Institute of Technology

South Hall 3A
Aman Sharma

11:20
CEST

Defending Bare-Metal: Lessons Learnt From AI Security Analysis of Metal3 and OpenStack Ironi - Dmitry Tantsur, Red Hat & Tuomo Tanskanen, Ericsson Software Technology

South Hall 3B
Dmitry Tantsur • Tuomo Tanskanen

11:22
CEST

SandScope: A Behavioral Audit Layer for MCP Tools in LLM Agent Supply Chains - Zhuoran Tan, University of Glasgow

South Hall 3A
Zhuoran Tan

11:42
CEST

Librarian Catches Thief: Surfacing Supply Chain Attack Campaigns via Document Similarity in an AI Agent Skill Registry - Gale Fagan, Brightflag

South Hall 3A
Gale Fagan