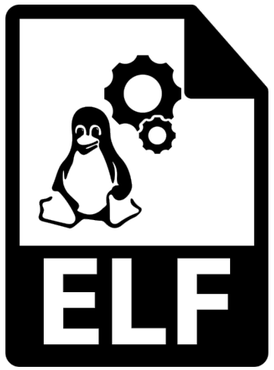
Uncovering Supply Chain Attack with Code Genome Framework

Dhilung Kirat, Jiyong Jang, Doug Schales, Ted Habeck, Ian Molloy, JR Rao

Dhilung Kirat

Jiyong Jang

AI Supply Chain Security Team
IBM **Research**




```
$ foo install bar
```

– Signed with a certificate.
– Lists dependencies.

– Do you trust it?

"You can't trust code that you did not totally create yourself."

—Ken Thompson

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

Figure 1 shows a self-reproducing program in the C[3] programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: 1) This program can be easily written by another program. 2) This program can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

[1] UNIX is a trademark of AT&T Bell Laboratories.

Supply Chain Attacks

SolarWinds (2019-2021) est. cost > $100B
- Malicious code (backdoor) pushed out through updates

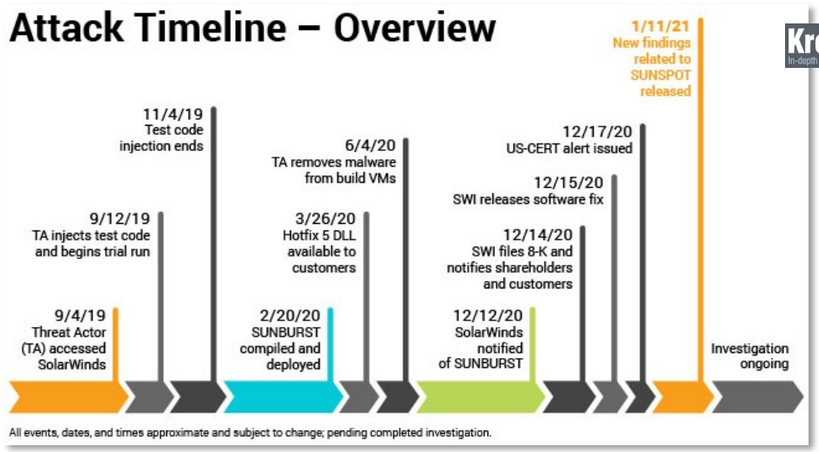
Dependency confusion (Feb 2021)
- Private vs public packages (npm, PyPi, RubyGems)

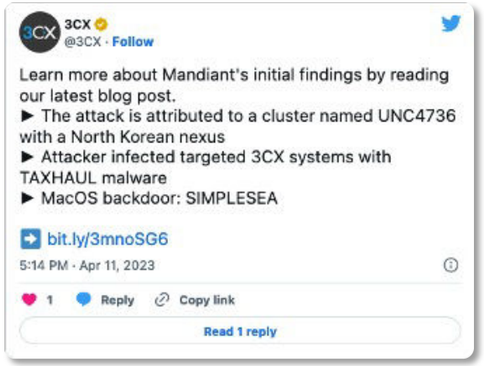
Codecov (Apr 2021)
- DevOps tool. Vulnerability in CI. Bash uploader modified

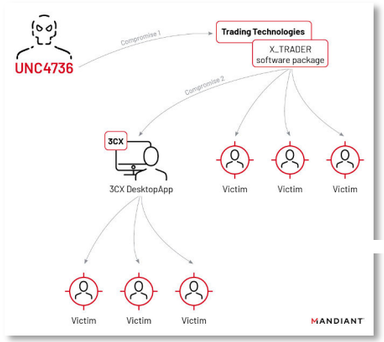
Kaseya (Jul 2021) ransom $70M
- IT solutions, including VSA (remote monitoring and management software) to deliver REvil ransomware

Protestware (Mar 2022)
- Popular NPM package wiped files in Russia and Belarus

3CX (Mar 2023)
- Backdoor implanted into Windows and macOS due to secondary supply chain attack







Codecov breach impacted 'hundreds' of customer networks: report

Updated: Reports suggest the initial hack may have led to a more extensive supply chain attack.

ZDNet

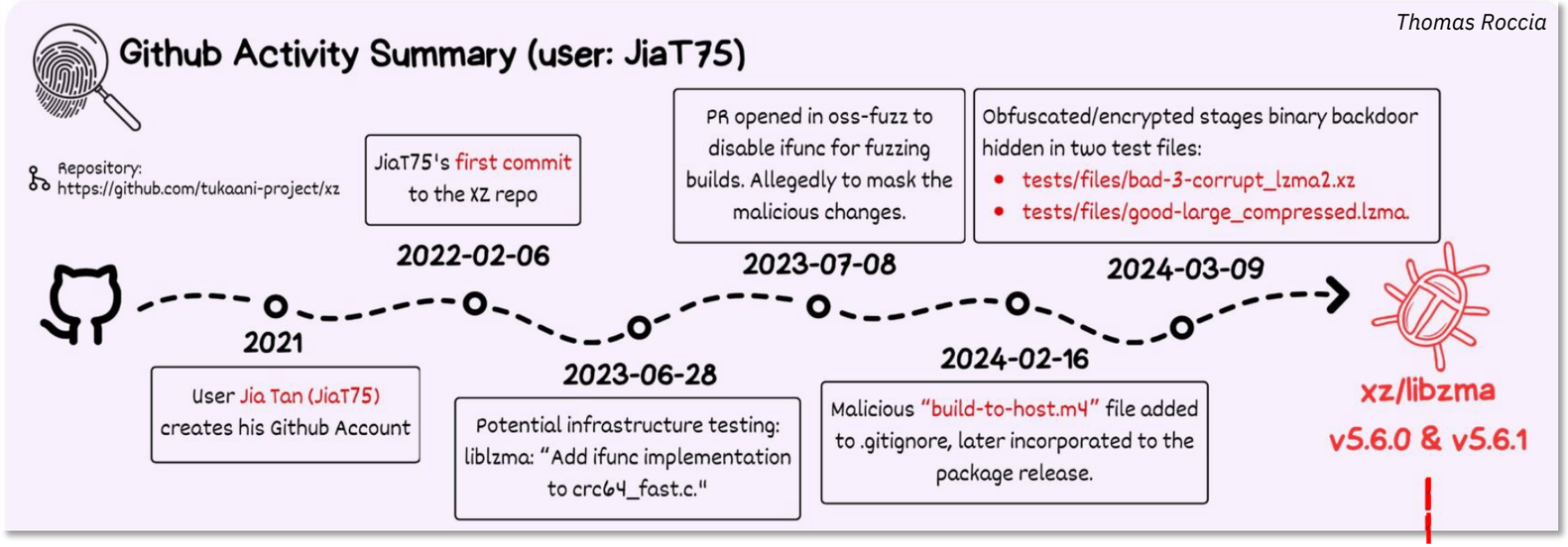
CISA-FBI Guidance for MSPs and their Customers Affected by the Kaseya VSA Supply-Chain Ransomware Attack

xz Backdoor

https://www.openwall.com/lists/oss-security/2024/03/29/4

```
Date: Fri, 29 Mar 2024 08:51:26 -0700
From: Andres Freund <andres@...razel.de>
To: oss-security@...ts.openwall.com
Subject: backdoor in upstream xz/liblzma leading to ssh server compromise

Hi,

After observing a few odd symptoms around liblzma (part of the xz package) on
Debian sid installations over the last weeks (logins with ssh taking a lot of
CPU, valgrind errors) I figured out the answer:

The upstream xz repository and the xz tarballs have been backdoored.

At first I thought this was a compromise of debian's package, but it turns out
to be upstream.


== Compromised Release Tarball ==

One portion of the backdoor is *solely in the distributed tarballs*. For
easier reference, here's a link to debian's import of the tarball, but it is
also present in the tarballs for 5.6.0 and 5.6.1:

https://salsa.debian.org/debian/xz-utils/-/blob/debian/unstable/m4/build-to-
host.m4?ref_type=heads#L63

That line is *not* in the upstream source of build-to-host, nor is
build-to-host used by xz in git.  However, it is present in the tarballs
released upstream, except for the "source code" links, which I think github
generates directly from the repository contents:

https://github.com/tukaani-project/xz/releases/tag/v5.6.0
https://github.com/tukaani-project/xz/releases/tag/v5.6.1


This injects an obfuscated script to be executed at the end of configure. This
script is fairly obfuscated and data from "test" .xz files in the repository.
```

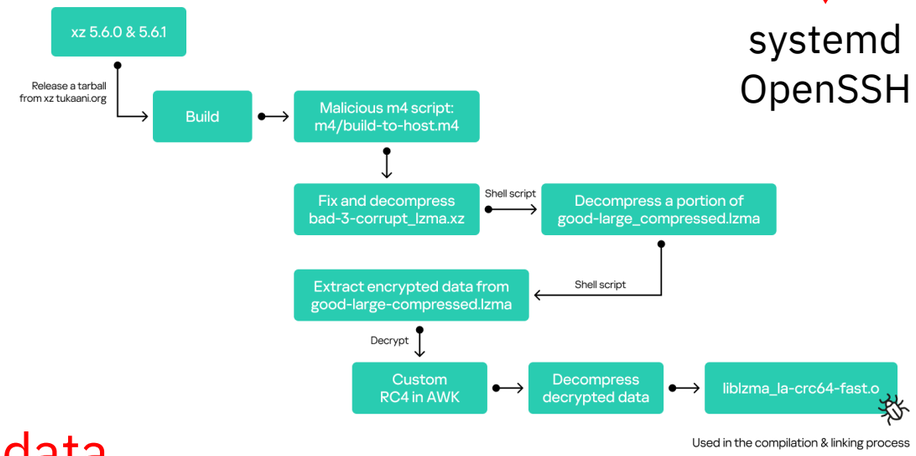
Thomas Roccia



Re: [xz-devel] XZ for Java

Jigar Kumar | Tue, 07 Jun 2022 09:00:18 -0700

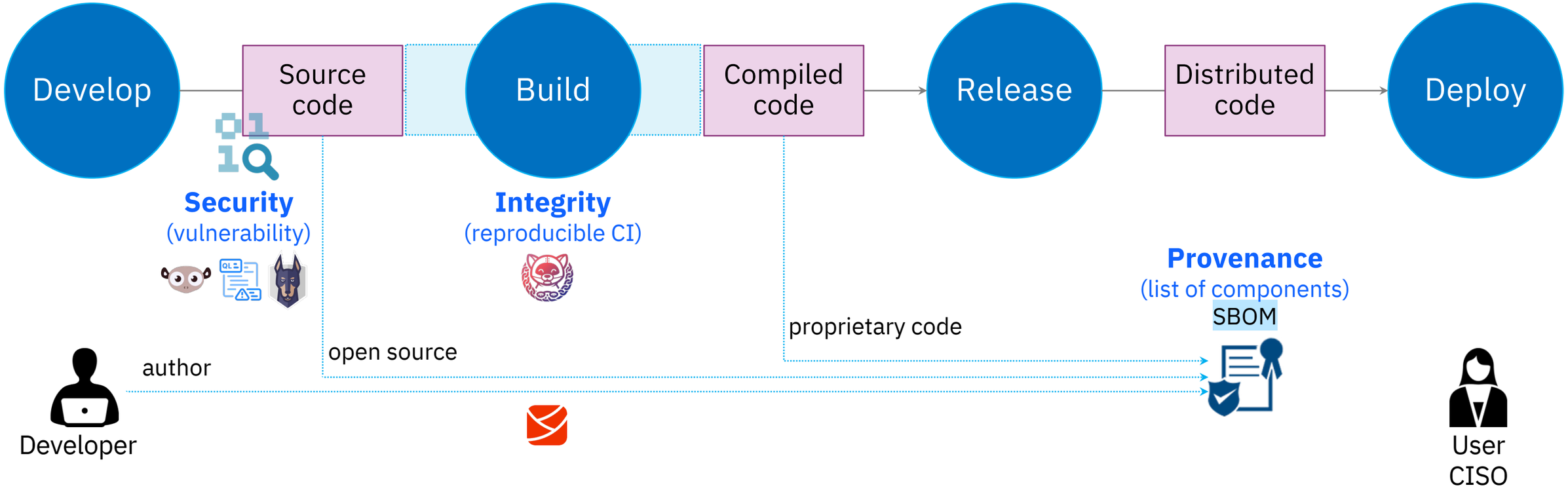
```
Progress will not happen until there is new
maintainer. XZ for C has sparse
commit log too. Dennis you are better off waiting
until new maintainer happens
or fork yourself. Submitting patches here has no
purpose these days. The
current maintainer lost interest or doesn't care to
maintain anymore. It is sad
to see for a repo like this.
```



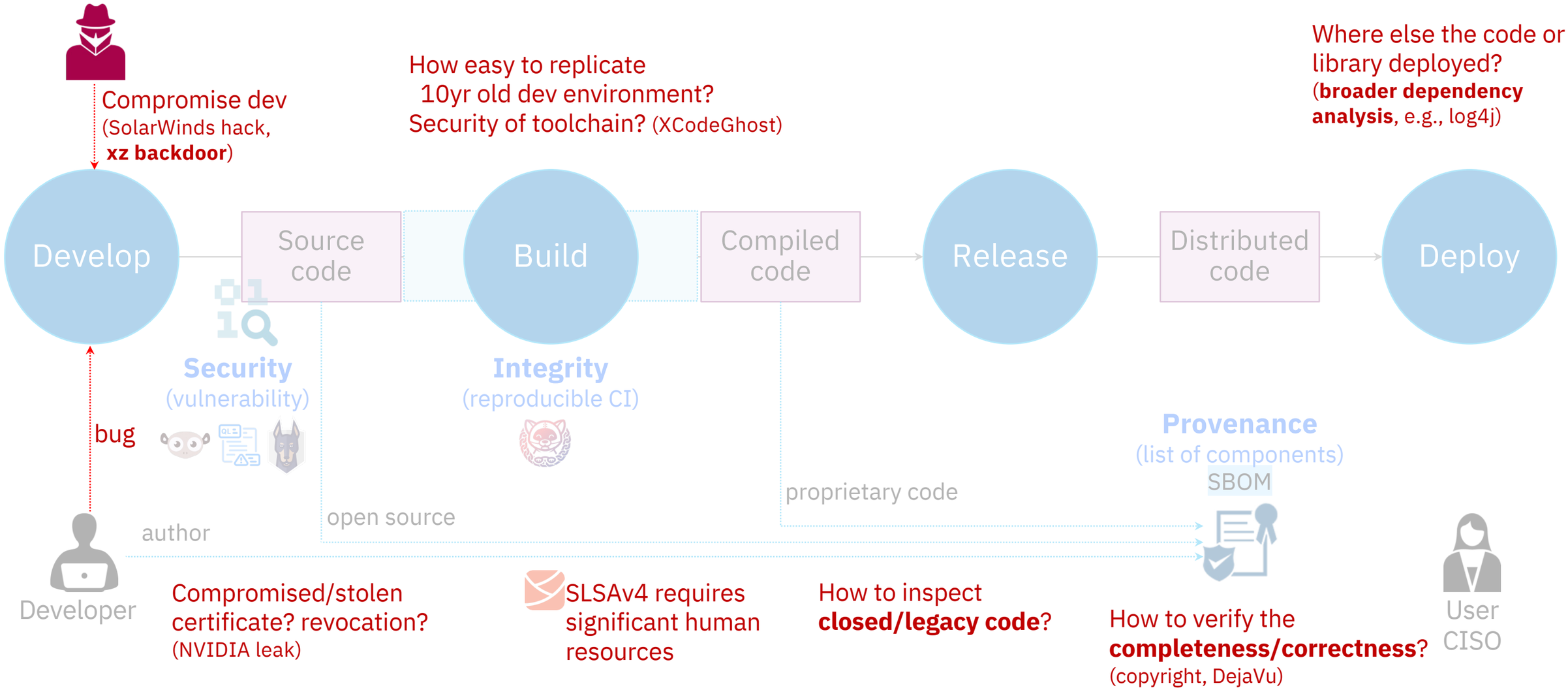
Semantic gap between compiled code behavior and its metadata

https://securelist.com/xz-backdoor-story-part-1/112354/

Supply Chain Security: Industry approach to protecting CI/CD pipelines

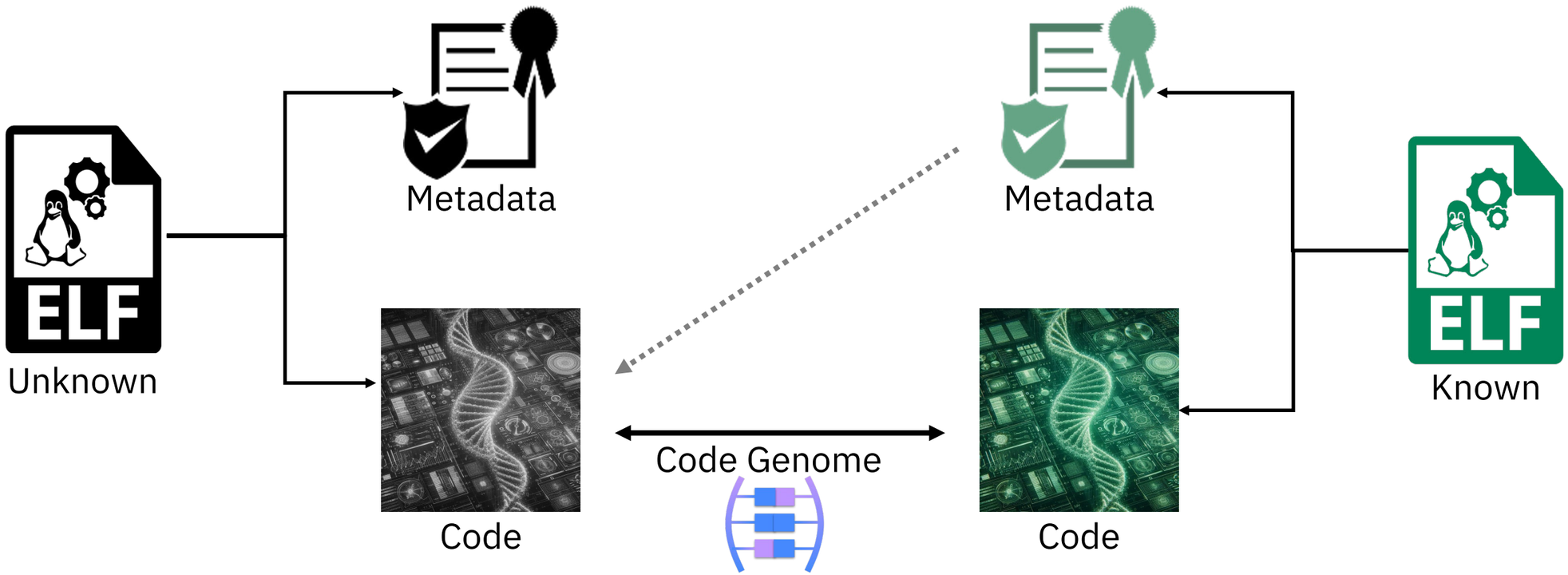


Supply Chain Security: Open security issues and residual risks



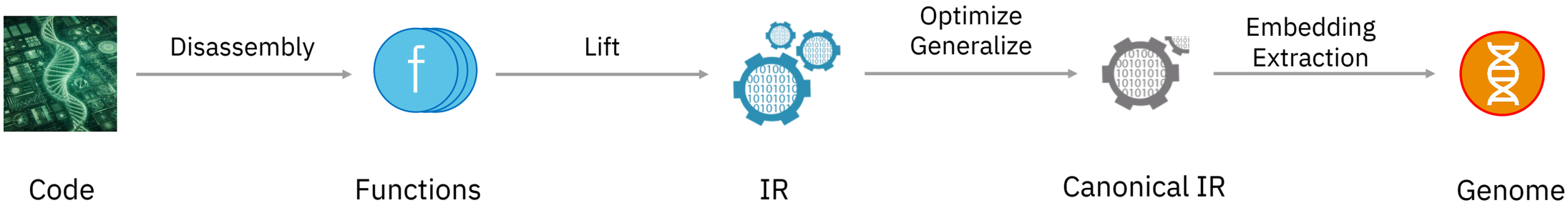
Code **Genome**

The Semantic Gap

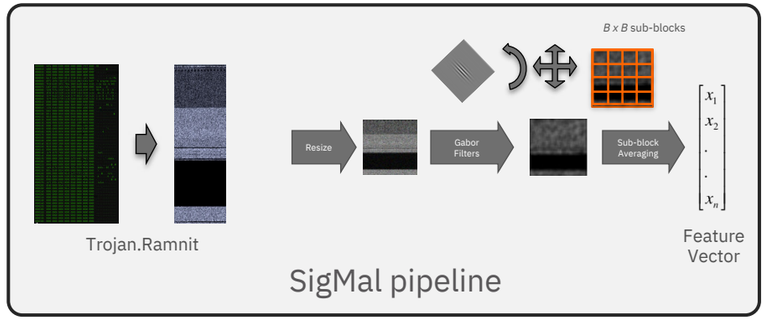


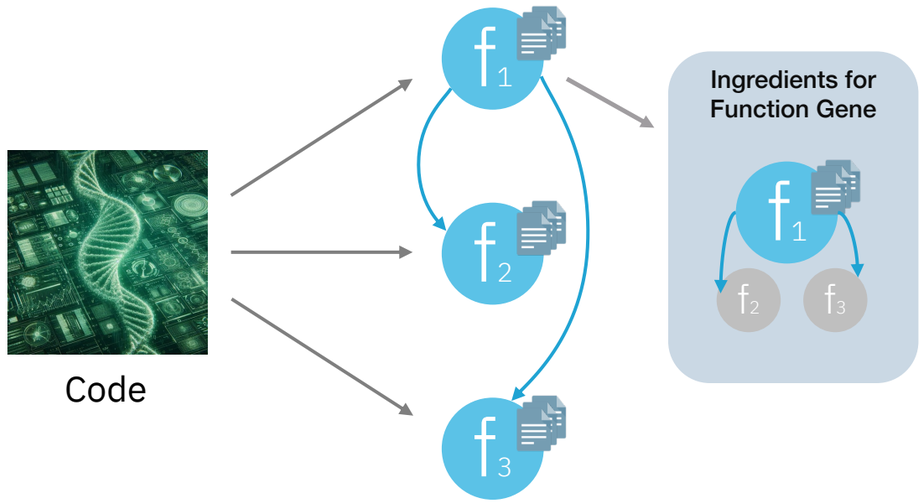
Build chain of trust
by following code equivalency

Code Genome Pipeline



Code Genome Pipeline





Ingredients for Function Gene

RetDec

LLVM IR

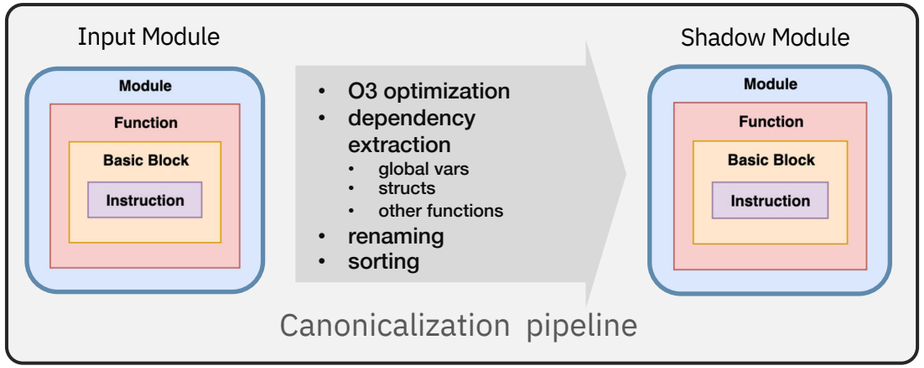
Optimize

Canonicalization LLVM Pass

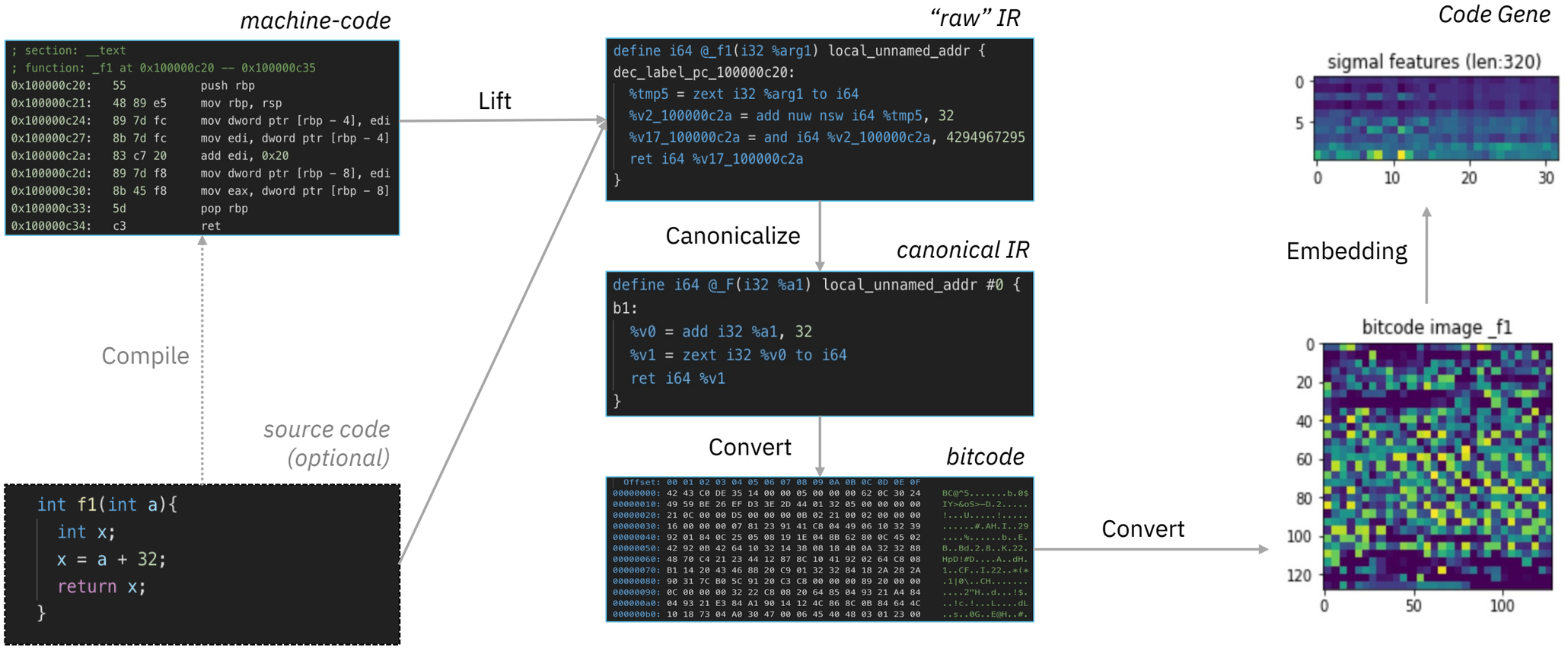
Canonical IR

SigMal

Genome

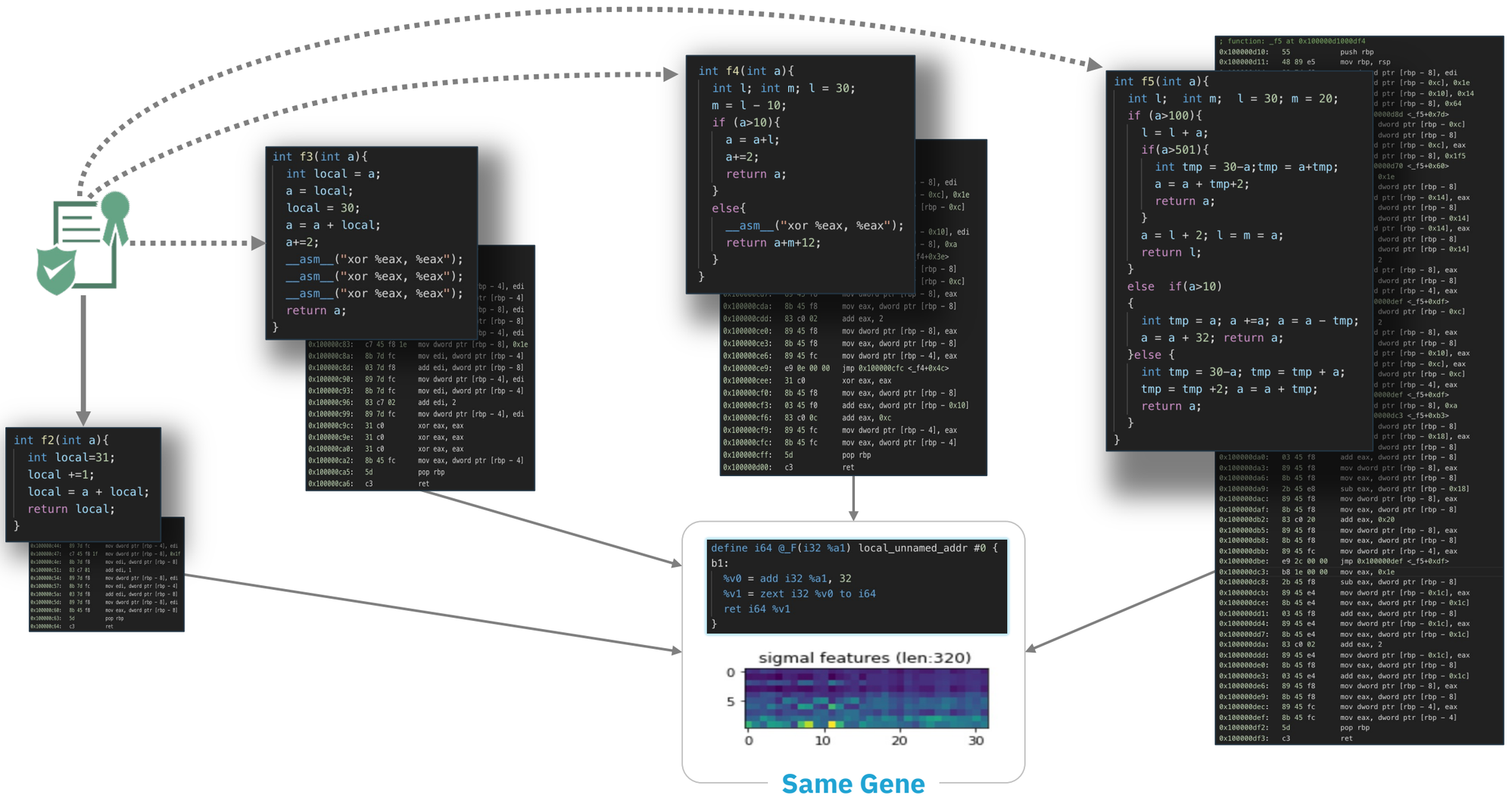


Code Genome Pipeline



Gene can be constructed from closed-source/legacy code where source code is not easily available.

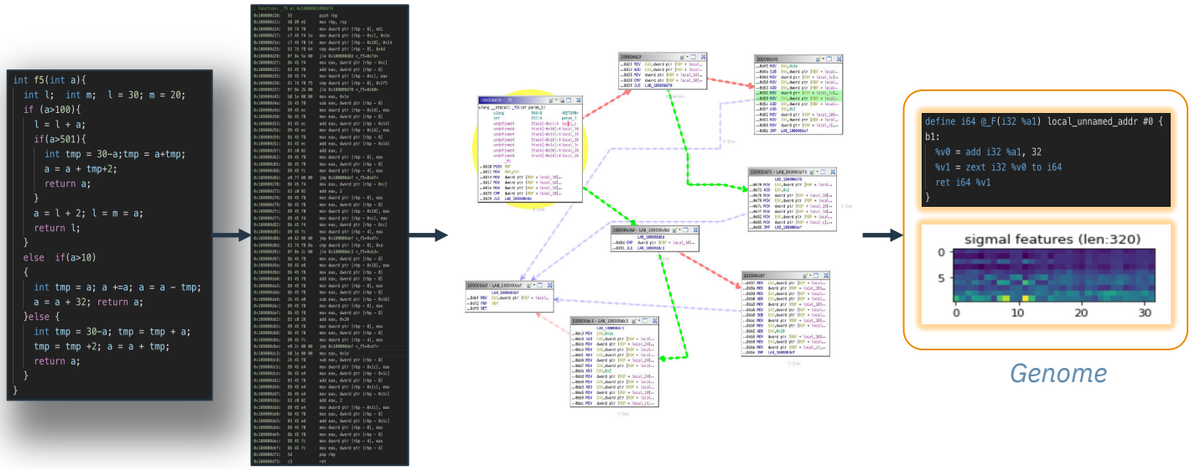
Code Genome: Semantically meaningful fingerprint



Advantages and Challenges

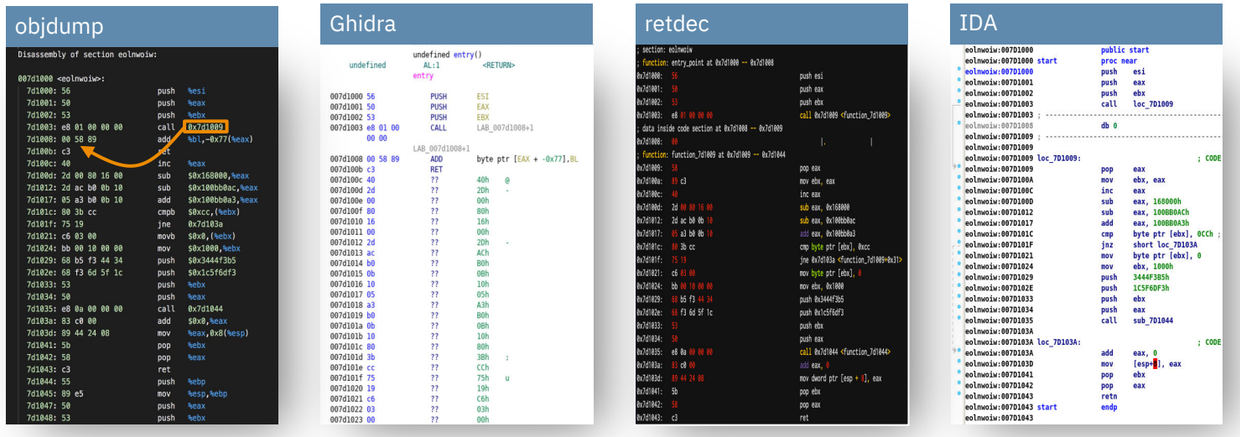
Advantages

- Across multiple architectures (x86, ARM, …)
- Across multiple compilers (gcc, clang, …)
- Across multiple optimization levels
- Handling obfuscation



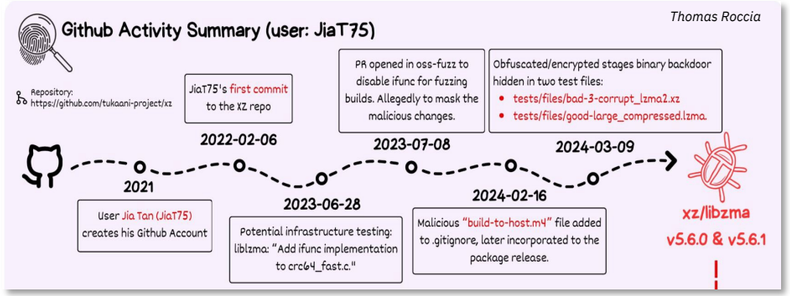
Challenges

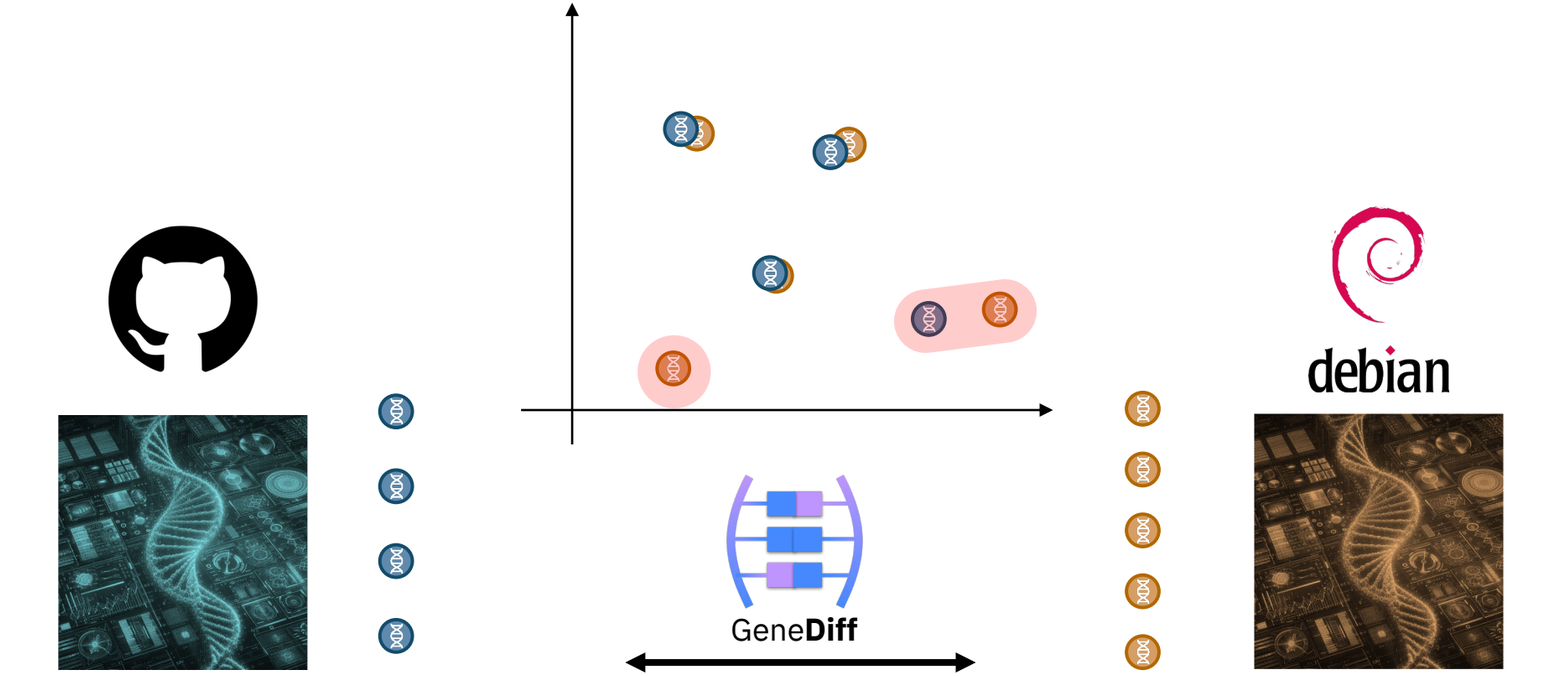
- Disassembly is undecidable
- Function boundary identification
- Loss of architecture specific nuances
- Canonicalization cannot completely recover high-level abstraction



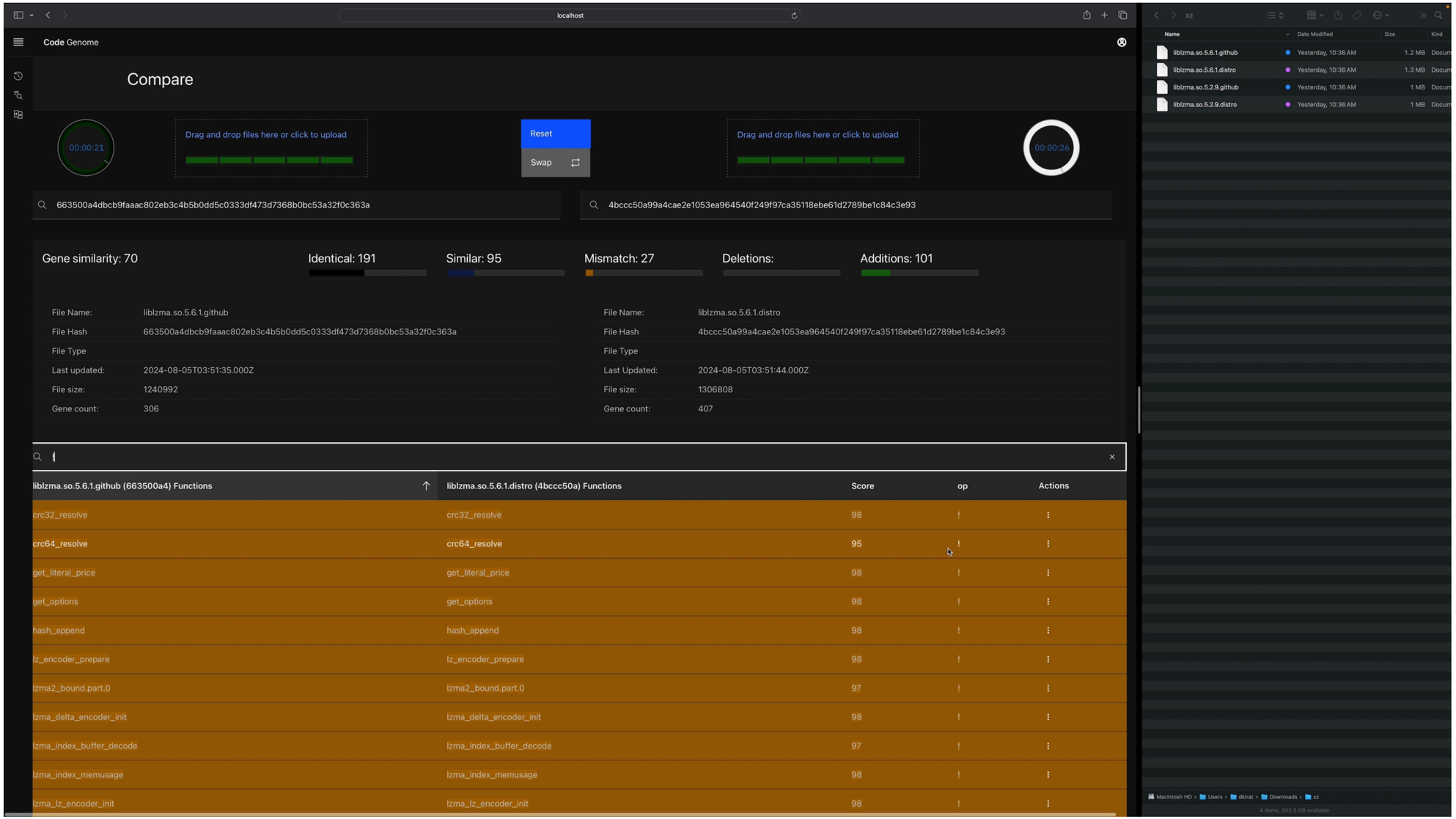
Uncovering **Supply Chain Attack**

Demo 1: xz backdoor analysis using Code Genome

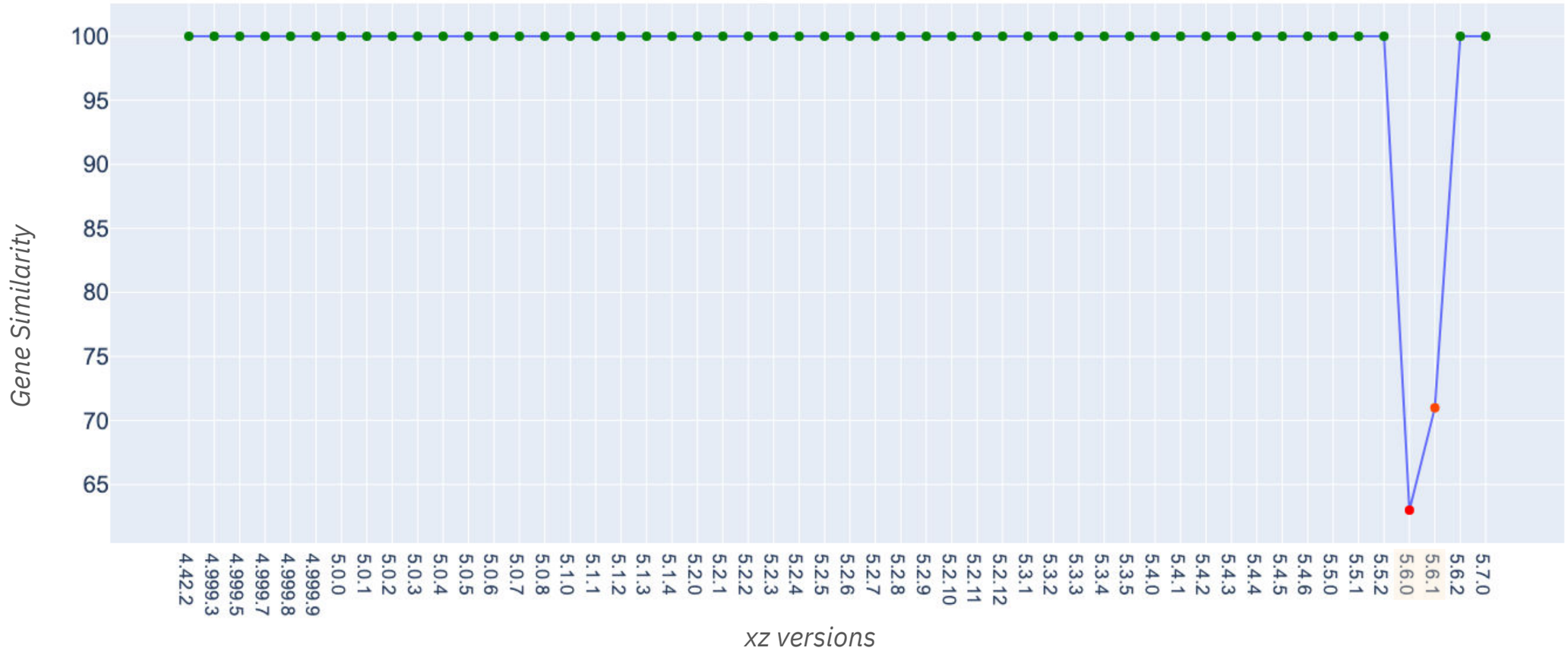




Demo 1: xz backdoor analysis using Code Genome

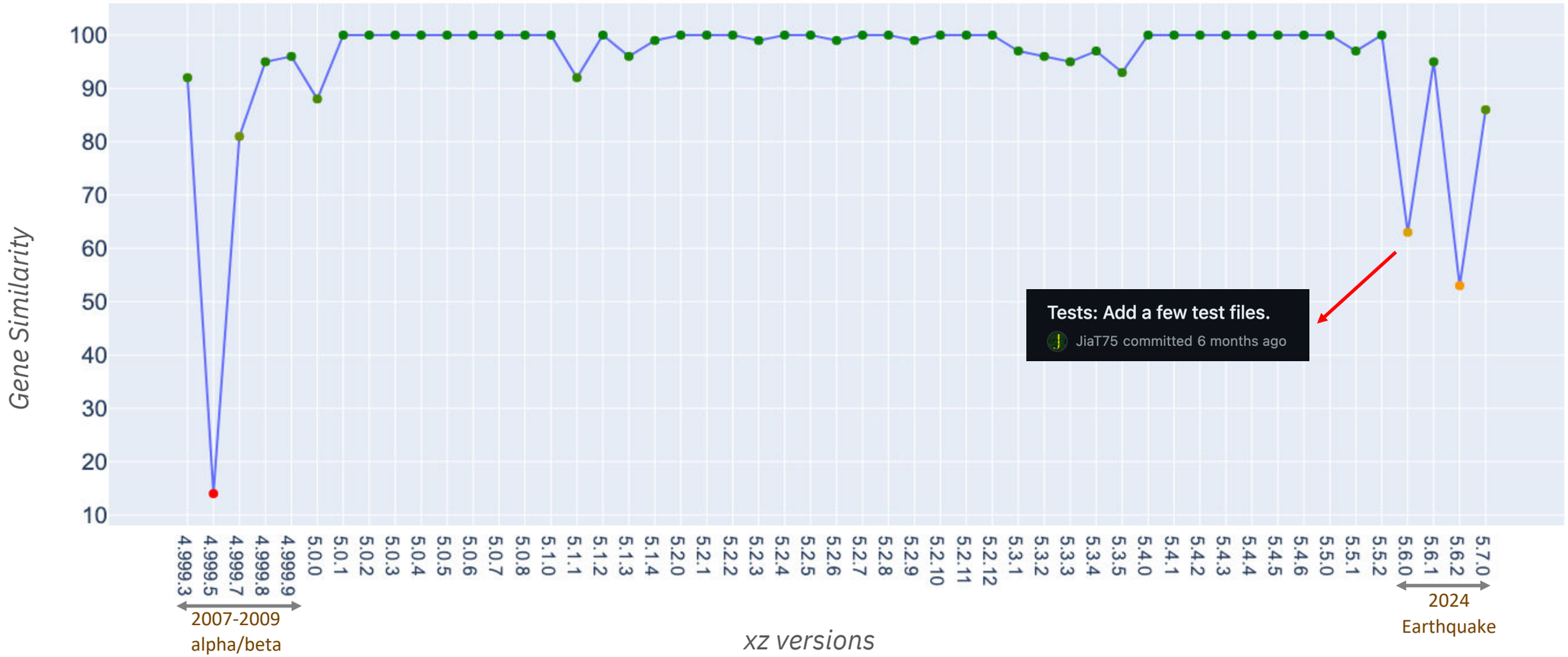


xz backdoor Gene Similarity Analysis using GeneDiff



Local vs distribution builds of same version

xz backdoor Gene Similarity Analysis using GeneDiff



Incremental version similarity in distribution builds

Improving **Supply Chain Security**

Trust but Verify SBOM: Metadata vs. Code

Problem

- Each vendor creates SBOM of their own software including open-source and closed-source components.
- How can we verify its *correctness* (containing incorrect library mistakenly/maliciously) and *completeness* (missing library)?

$ sbom generation tools

Dockerfile > ...

```
1   FROM ubuntu:focal
2
3   RUN apt-get update
4   RUN apt-get install -y wget
5
6   RUN apt-get update
7
```

```
"bom-ref": "pkg:wget@1.20.3-1ubuntu2",
"type": "library",
"name": "wget",
"version": "1.20.3-1ubuntu2",
"licenses": [
  {
    "license": {
      "name": "UNKNOWN"
    }
  }
],
```

Dockerfile > ...

```
1   FROM ubuntu:focal
2
3   RUN apt-get update
4   RUN apt-get install -y wget
5
6   RUN mv /var/lib/dpkg/status /var/lib/dpkg/status.bak
7   RUN touch /var/lib/dpkg/status
8
9   RUN apt-get update
10
```

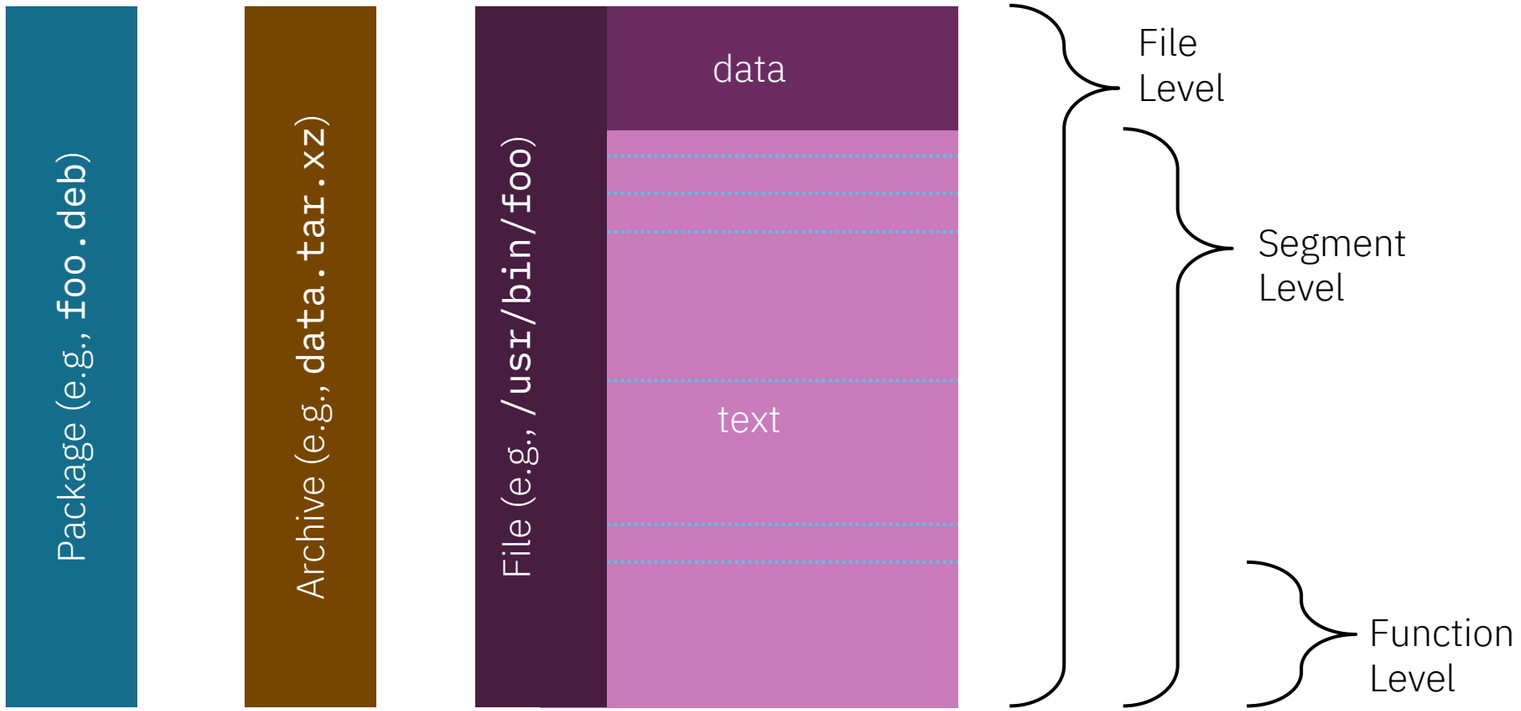
delete dpkg DB

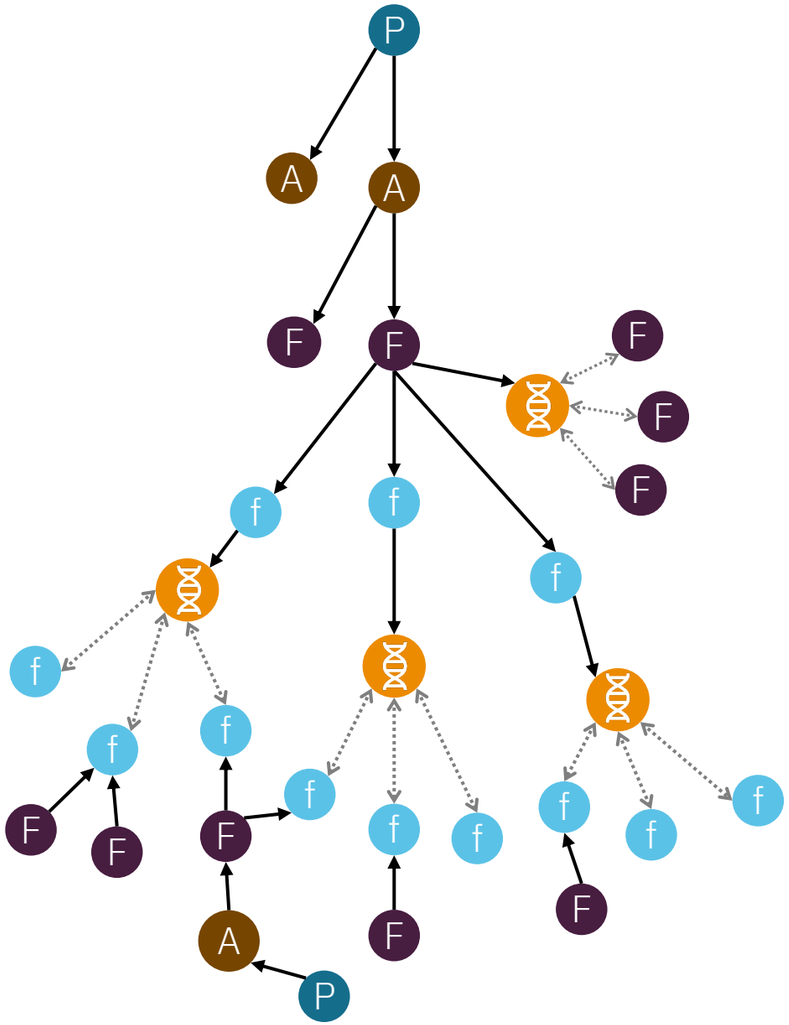
```
sbom/docker ❯ grep wget sbom.dpkg.json
sbom/docker ❯
```

"Unfortunately, some images – such as the official node image on Docker Hub – incorrectly report the version of OpenSSL that's used by the Node.js runtime."

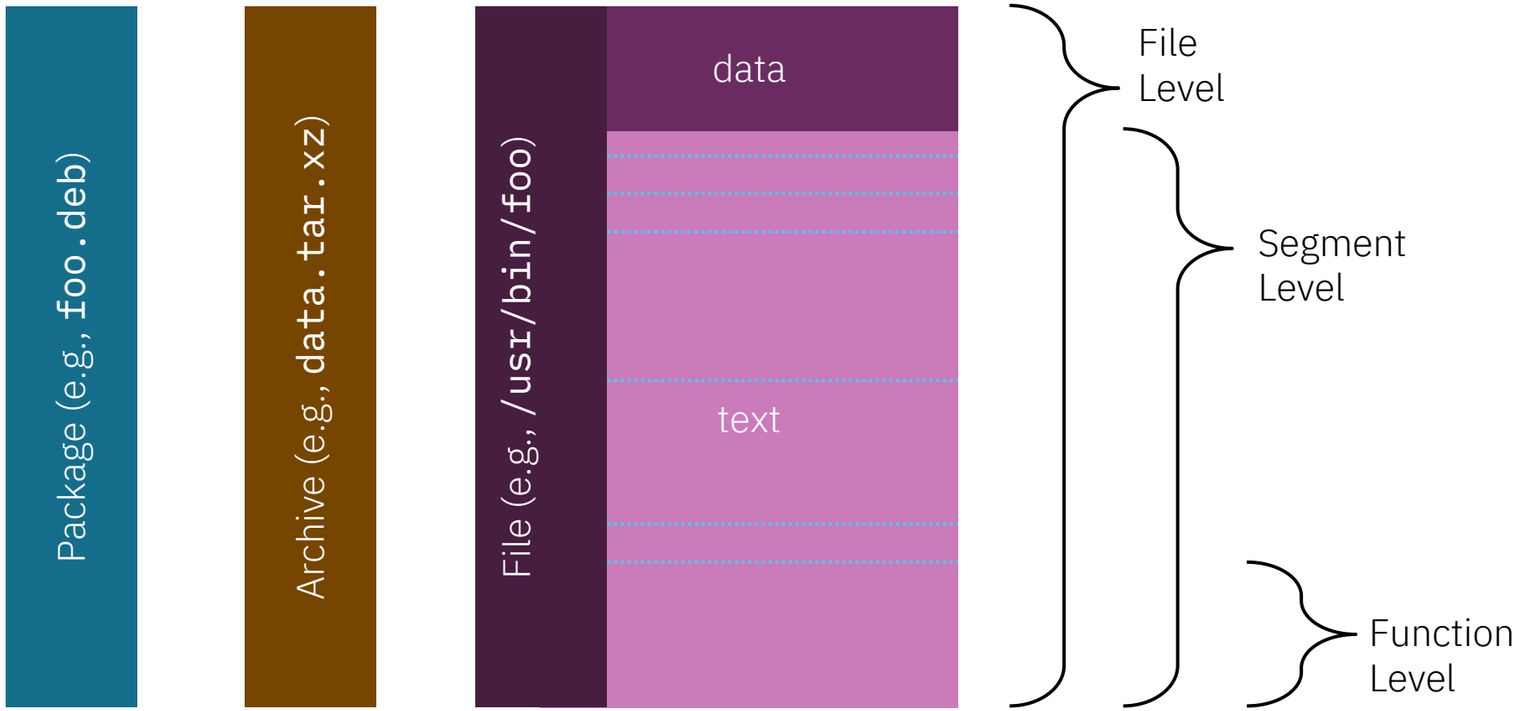
https://www.chainguard.dev/unchained/mitigating-critical-openssl-vulnerability-with-chainguard

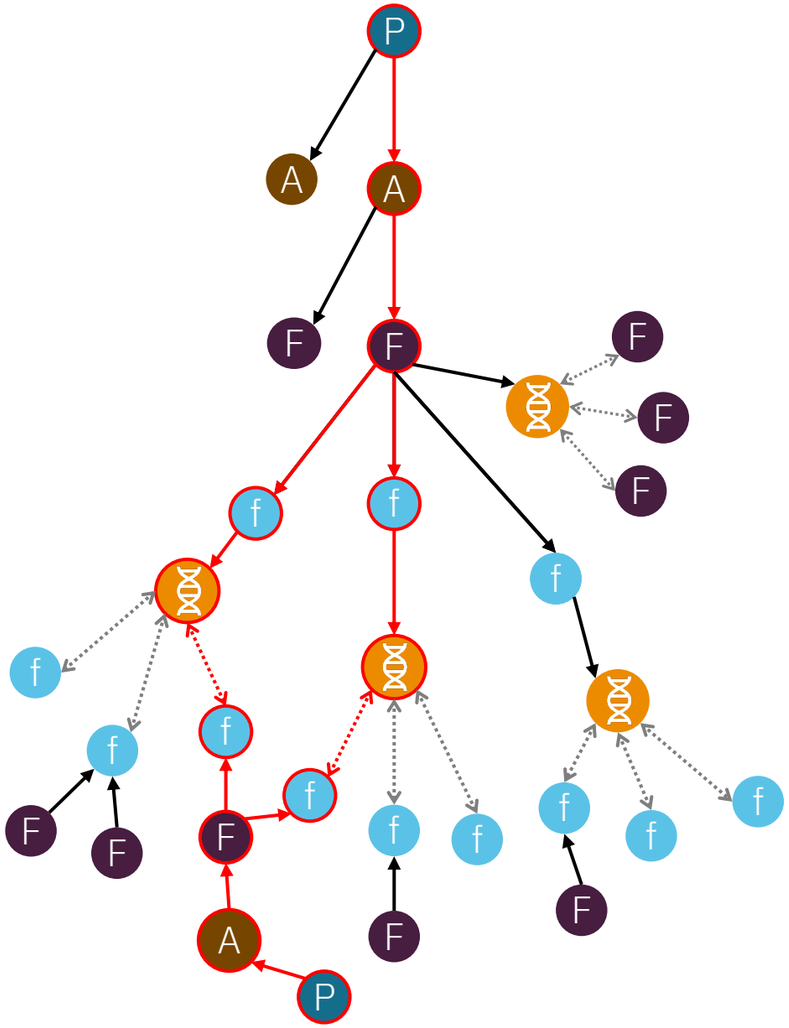
Knowledge Graph: Gene Granularity





Knowledge Graph: Gene Granularity





Demo 2: SBOM generation for an unknown `rpm` package

Custom `rpm` package

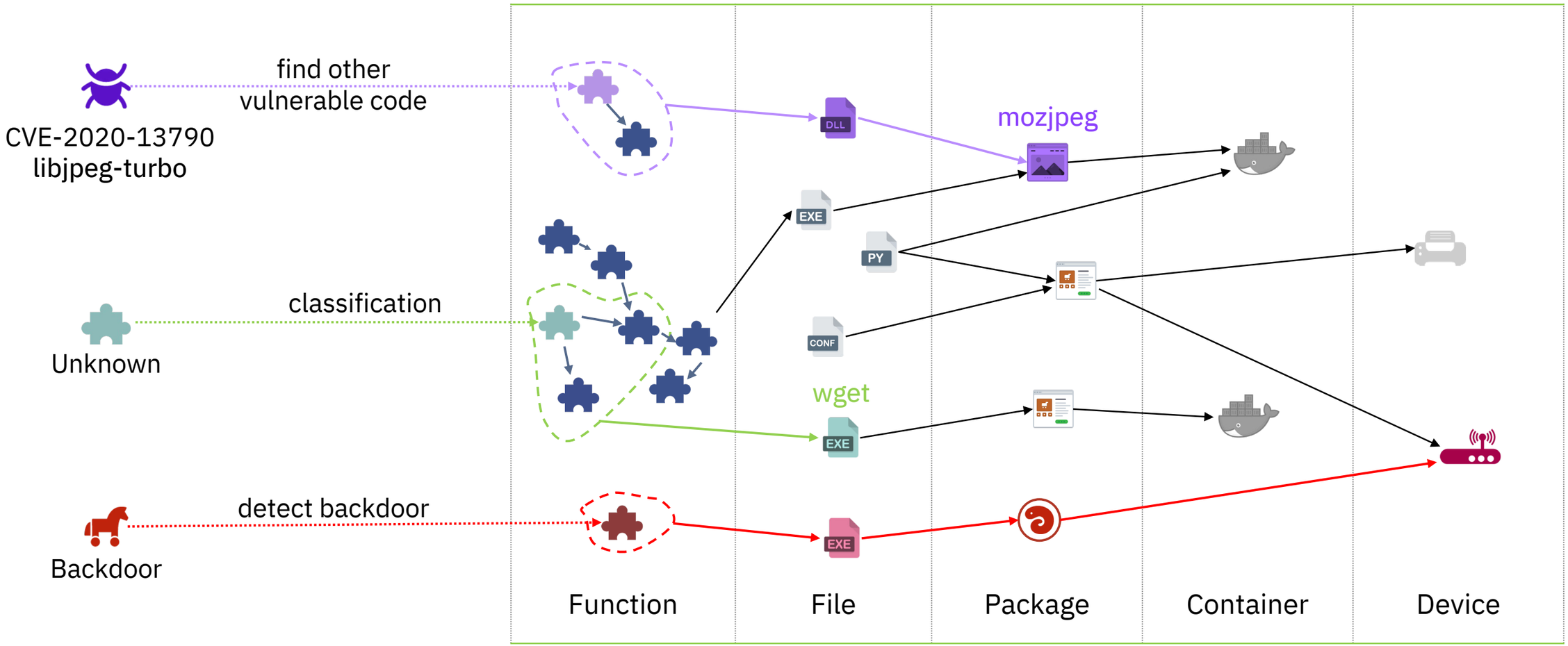
```
unknown2a
├── etc
│   └── sysconfig
│       └── rdisc
└── usr
    ├── bin
    │   ├── ping
    │   ├── ping6 -> ping
    │   ├── tracepath
    │   └── tracepath6
    ├── lib
    │   └── systemd
    │       └── system
    │           └── rdisc.service
    ├── sbin
    │   ├── arping
    │   ├── clockdiff
    │   ├── ifenslave
    │   ├── ping6 -> ../bin/ping
    │   ├── rdisc
    │   ├── tracepath -> ../bin/tracepath
    │   └── tracepath6 -> ../bin/tracepath6
    └── share
        ├── doc
        │   └── iputils-20160308
        │       ├── README.bonding
        │       └── RELNOTES
        └── man
            └── man8
                ├── arping.8.gz
                ├── clockdiff.8.gz
                ├── ifenslave.8.gz
                ├── ping.8.gz
                ├── ping6.8.gz -> ping.8.gz
                ├── rdisc.8.gz
                ├── tracepath.8.gz
                └── tracepath6.8.gz -> tracepath.8.gz
```

SBOM generated by Code **Genome**

Component	Version	License
arping	20160308	BSD and GPLv2+
clockdiff	20160308	BSD and GPLv2+
ifenslave	20160308	BSD and GPLv2+
iputils	20160308	BSD and GPLv2+
ping	20160308	BSD and GPLv2+
rdisc	20160308	BSD and GPLv2+
tracepath	20160308	BSD and GPLv2+
tracepath6	20160308	BSD and GPLv2+

Integrating with other SBOM analysis platforms

Knowledge Graph: Code Genome and Use Cases



Code Genome KG

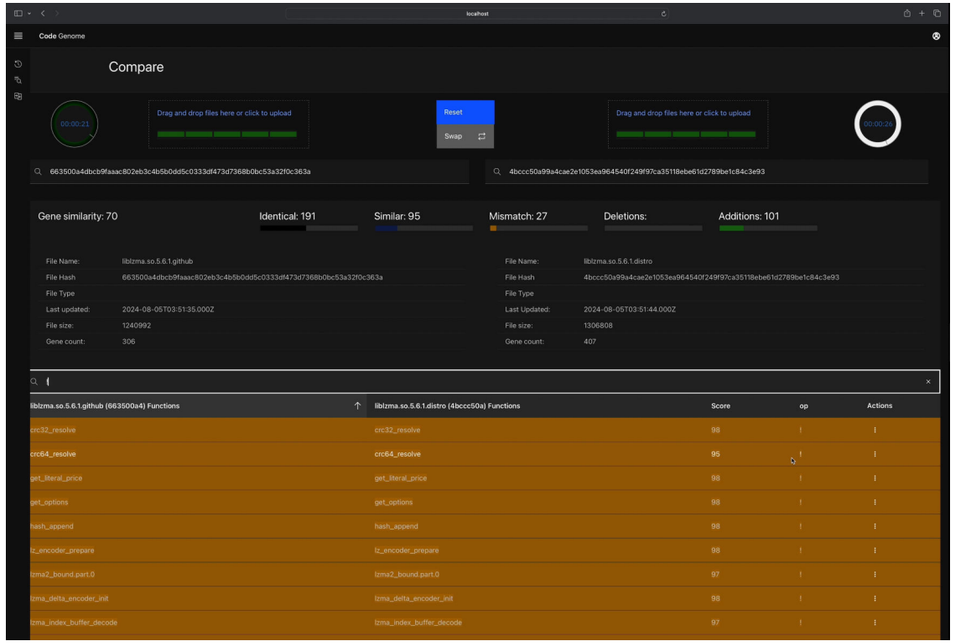
Open Sourcing **Code Genome**

Status and Roadmap

Open-source tools
- **`Code Genome Framework`**
  - GeneDiff, Basic KG, CLI tools, and GUI
  - Currently supported
    - Binaries: `ELF, PE, Mach-O`
    - Architectures: `x86, x86_64, arm, aarch64, mips, ppc`
  - Optimized canonicalization
- **`Jaudit`**
  - JAR file support
  - JAR version identification
  - CVE annotation

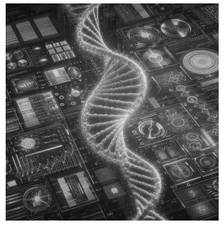
Next steps
- Support
  - Packages: `deb, rpm, ipa`
  - Archives: `ar, cpio, tar, bzip2, gzip, zstd, xz, rar, 7zip`


```
git clone https://github.com/code-genome/codegenome.git
cd codegenome

make start
```



Takeaways

Semantic Gap

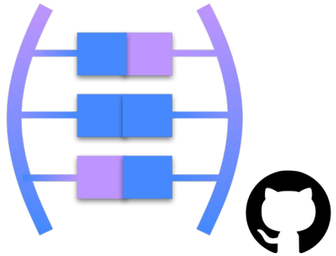


Code

Metadata

Inherent sematic gap breaks the transfer of trust from metadata to code

Code Genome



Now open-sourced Code Genome Framework can help bridge that gap

Supply Chain Security

Detection of XZ-backdoor demonstrates framework's capability in improving supply chain security

Dhilung Kirat ✉ *dkirat@us.ibm.com*

Jiyong Jang ✉ *jjang@us.ibm.com*

IBM **Research**

github.com/code-genome