Code Genome: Fingerprinting Code to Help Secure the Software Supply Chain

Jiyong Jang
IBM Team

Jeff Borek
Carlos Fonseca
Ted Habeck
Nigel Hinds
Jiyong Jang

Dhilung Kirat
Arnaud Le Hors
Ian Molloy
JR Rao
Matt Rutkowski
Doug Schales
Here is my application.
I signed it with this certificate.
It uses these libraries.
Please install it.
What do you do?
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.

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® 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 stunned 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 1

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 C1 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.

1UNIX is a trademark of AT&T Bell Laboratories.
2© 1984 805-0782/84/0605-0737 $1.50

—Ken Thompson
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
Supply Chain Security
Industry approach to protecting CI/CD pipelines
Supply Chain Security
Open security issues and residual risks

- Compromise dev (SolarWinds hack)
- How easy to replicate 10yr old dev environment? Security of toolchain? (XCodeGhost)
- Compromised/stolen certificate? revocation? (NVIDIA leak)
- SLSAv4 requires significant human resources
- How to inspect closed/legacy code?
- How to verify the completeness/correctness?
- Where else the code or library deployed? (broader dependency analysis, e.g., log4j)

Develop → Source code → Build → Compiled code → Release → Distributed code → Deploy

- Security (vulnerability)
- Integrity (reproducible CI)
- Provenance (list of components)
- Proprietary code
- Open source
- SBOM

Developer

User CISO

Code Genome © 2022 IBM Corporation
Software Fingerprints for Software Assurance

Stability across architectures
- 2.13.2 Ubuntu i386
  768:BkrtXbNQQGpDFbQE7PX7kvroyC8zIuNc8:ir1bNQVzFPLSrRN
- 2.13.2 Ubuntu amd64
  768:XXqchVhVlpgPCNzQSA8w5ie12QSn6CDyrs91c:nqilhsqSUtlcQEvriI

Stability across distributions
- 2.14.1 Ubuntu i386
  1536:BfaN57akumTpBnl/OyZl4Z170LiVEF2vrag//xFokTz0TGrw7
- 2.14.1 Ubuntu amd64
  768:XXqchVhVlpgPCNzQSA8w5ie12QSn6CDyrs91c:nqilhsqSUtlcQEvriI
- 2.14.1 RHEL7 amd64
  768:11qQHeMoGkp+Q3L6Vvb7+WwUT7VXZvFWDPYeoyN:1ISMeG0+gub77UT7ZZvFWDPJ

Robustness for legacy

Continuity across package repositories

Same Genome

Similar Genome

Same Genome
Timeline

2011  BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis
2013  Sigmal: A static signal processing-based malware triage
       Towards automatic software lineage inference
2015  Malgene: Automatic extraction of malware analysis evasion signature
       Experimental Study of Fuzzy Hashing in Malware Clustering Analysis
2017  Building a better vulnerability scanner for Docker
2020  Consolidating structured and unstructured security and threat
       intelligence with knowledge graphs
2021  Software intelligence as-a-service
       Find log4j buried on legacy systems
2022  Code Genome
Code Genome: Semantically meaningful fingerprint

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

Same Genome
Key Idea: Code Genome construction

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

- **Package** (e.g., foo.deb)
- **Archive** (e.g., data.tar.xz)
- **File** (e.g., /usr/bin/foo)

**Level Descriptions**
- **Data Level**
- **Segment Level**
- **Function Level**

**Examples**
- **Data**
- **Text**

**Labels**
- **P**
- **A**
- **F**

Code Genome © 2022 IBM Corporation
Code Genome Granularity

- **File (e.g., /usr/bin/foo)**
  - **data**
  - **text**

- **Package (e.g., foo.deb)**

- **Archive (e.g., data.tar.xz)**

**Levels:**
- **File Level**
- **Segment Level**
- **Function Level**
Demo Setup

Ubuntu Package Mirror | Seed the KG with Package Mirrors | IBM Code Genome | Compute Genes and Store in Database | Knowledge Base

Ubuntu Package Mirror | Seed the KG with Package Mirrors | IBM Code Genome | Compute Genes and Store in Database | Knowledge Base

Mystery Package | Identified Package and Metadata

Compile | Upload
Demo: Verifying `wget`

Compiling `wget` for multiple archs

Identifying (unknown) binary

Code Genome © 2022 IBM Corporation
Use Case 1: Finding Log4j
Legacy Software Discovery

- Software deployments are a turducken
  - zip, tar, container image, jar, etc.
  - Dependencies often wrapped up
- No good provenance or CMDB
- Can’t rely on standard directories, filenames, or hashes
- Code can be repackaged
- Applies across the board: CICD, DevSecOps and Legacy
Use Case 2: SBOM Verification

– 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)?

– Value
  • Given software, we can verify (generate) SBOM
  • Support closed-source and legacy software without requiring source code access
  • Help developers generate correct SBOM
  • Vet software before integrating/deploying into a product

Where else the code or library deployed? (broader dependency analysis, e.g., log4j)
“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.”


$ sbom generation tools

```
FROM ubuntu:focal
RUN apt-get update
RUN apt-get install -y wget
RUN apt-get update
```

```
FROM ubuntu:focal
RUN apt-get update
RUN apt-get install -y wget
RUN mv /var/lib/dpkg/status /var/lib/dpkg/status.bak
RUN touch /var/lib/dpkg/status
RUN apt-get update
```

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

Current Status
- Several techniques for computing genes
- Support for binaries, packages, bytecode
- Scalable, cloud native processing engine
- Currently performing large-scale evaluation

Release coming soon:
- A version of genome generation
- Service demonstrating technology
- Utilities for handling genomes and querying service

Requests:
- Welcome feedback, support and collaboration
- Insights on capabilities and use cases