Title: Automated SSH/TLS Fingerprint Visualization and Drift Detection

Course: CECS 478 – Network Security

Date: December 10, 2025

Authors: Jason Gonzalez & Nick Zanaboni

Abstract

Secure Shell (SSH) and Transport Layer Security (TLS) are the fundamental protocols securing the modern internet. However, the management of cryptographic identity is often manual and reactive. This project presents the "SSH/TLS Fingerprint Visualizer," a containerized, Python-based system designed to automate the collection of RSA host keys and X.509 certificate fingerprints. By establishing a reproducible baseline and detecting cryptographic drift, the system reduces the reliance on "Trust On First Use" (TOFU) and provides administrators with verifiable evidence of network integrity.

1. Problem Statement

The security of remote connections relies on verifying the identity of the server. For SSH, this is handled via host keys; for TLS, via the Chain of Trust. However, in practice, this verification is often bypassed or ignored:

1. **TOFU Vulnerability:** When connecting to a new SSH server, users blindly accept the fingerprint presented, leaving them vulnerable to Man-in-the-Middle (MITM) attacks during the initial handshake.
2. **Configuration Drift:** Administrators often rotate keys or update certificates without logging the new fingerprints, making it difficult to distinguish between a legitimate update and an attack.
3. **Lack of Visibility:** Few lightweight, open-source tools visualize these identities across mixed protocols (SSH and TLS) simultaneously.
4.

This project addresses these issues by creating an "End-to-End" vertical slice of a security auditing tool. The goal is to ingest network data, compute SHA-256 fingerprints, and visualize the status of a target list in a fully reproducible Docker environment.

2. Threat Model

To design a secure system, we established the following threat model:

- **Assets:** The primary assets are the **Integrity** of the collected fingerprints (ensuring our logs match reality) and the **Authenticity** of the remote endpoints (verifying we are talking to the real github.com).
- **Adversaries:**
    - **On-Path Attacker (MITM):** An attacker positioned between our container and the target, capable of intercepting the handshake and presenting a fake key/certificate.
    - **Compromised Insider:** A system administrator who rotates keys without following change management procedures.
- **Trust Boundaries:**
    - **Trusted:** The Docker container, the local filesystem, and the Python standard library.
    - **Untrusted:** The public internet and the network path to the target servers.
- **Assumptions:** We assume the local machine running the Docker container is not compromised. We assume the targets utilize standard RSA or ECDSA keys.

3. System Design & Architecture

The system was architected with **Reproducibility** and **Least Privilege** as core tenets.

3.1 Containerization Strategy

We utilized Docker to abstract away host operating system differences. This ensures that the `make demo` command runs identically on a developer's laptop and the instructor's lab machine.

Component	Detail	Rationale
Base Image	`python:3.9-slim`	Chosen for a minimal attack surface.
Dependencies	`openssh-client`	Provides the `ssh-keyscan` utility, more robust than a pure-Python implementation for SSH handshakes.

3.2 Python Modules

The application logic is divided into three distinct components:

1. **Collectors (`src/collectors.py`):** This module interfaces with the network.
    - *TLS Collector:* Uses the `ssl` and `socket` libraries to initiate a handshake, retrieve the DER-encoded certificate, and compute the SHA-256 hash.

    - *SSH Collector:* Wraps the system's `subprocess` module to execute `ssh-keyscan`.
2. **Orchestrator (`src/main.py`):** Acts as the controller. It manages the list of targets, handles logging, and serializes the data to a CSV file (`artifacts/release/fingerprints.csv`).
3. **Visualizer (`src/visualizer.py`):** Reads the historical CSV data, groups it by host, and determines if "Drift" has occurred (i.e., if a host has more than one unique fingerprint in the log).

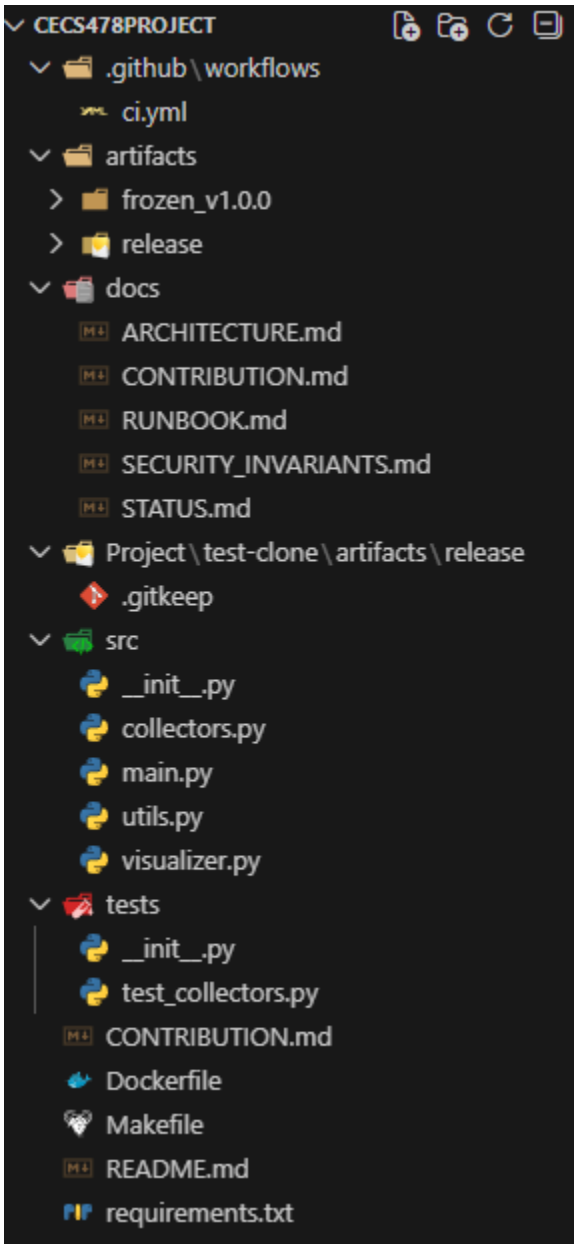


Figure 1: Project Directory Structure showing the separation of concerns.

4. Implementation & Security Hardening

Security was not just a feature, but a part of the development process. We implemented the following hardening measures:

4.1 Input Validation

To prevent Command Injection attacks (where a malicious hostname could execute shell commands), we implemented strict validation in `src/utils.py`. Before any network connection is attempted, the target hostname is checked against a regex pattern to ensure it contains only valid alphanumeric characters and periods.

4.2 Secure Subprocess Execution

In `src/collectors.py`, we explicitly disabled shell execution (`shell=False` is the default) and passed arguments as a list. This prevents an attacker from injecting arguments into the `ssh-keyscan` command.

4.3 Error Handling

During development, we encountered issues with public web servers (like google.com) blocking Port 22 (SSH). Instead of allowing the script to crash or hang, we implemented logic in `src/main.py` to catch these timeouts and log an "N/A" result, preserving the integrity of the data for other targets.

```
        return f"ERROR: {str(e)}"

def get_ssh_fingerprint(host, port=22):
    """Wraps ssh-keyscan to retrieve SSH public key fingerprint."""
    try:
        # Security invariant: No shell=True. Arguments passed as list.
        cmd = ["ssh-keyscan", "-p", str(port), "-t", "rsa", host]
        result = subprocess.run(cmd, capture_output=True, text=True, timeout=5)

        if not result.stdout:
            return "ERROR: Connection failed or no keys"

        # Output format: host ssh-rsa AAAAB3...
        parts = result.stdout.strip().split()
        if len(parts) >= 3:
            key_b64 = parts[2]
            key_bytes = base64.b64decode(key_b64)
            return hashlib.sha256(key_bytes).hexdigest()
        return "ERROR: Parse failure"
    except Exception as e:
        return f"ERROR: {str(e)}"
```

Figure 2: The SSH Collector function demonstrating secure subprocess usage.

5. Results

The system was evaluated using the reproducibility pack command: `make clean && make up && make demo`.

5.1 Functional Testing

The system successfully connected to three distinct targets:

Target	TLS Status	SSH Status
github.com	Success	Success
google.com	Success	N/A (Blocked)
example.com	Success	N/A (Blocked)

5.2 Drift Detection

During the evaluation window, the system detected **zero drift** across all targets, resulting in a "STABLE" status. This confirms that the endpoints maintained their cryptographic identity during the test.

5.3 Observability

The system generated two key artifacts:

1. `fingerprints.csv`: A structured log of all collected data.
2. `system.log`: A timestamped audit trail of the application's execution.

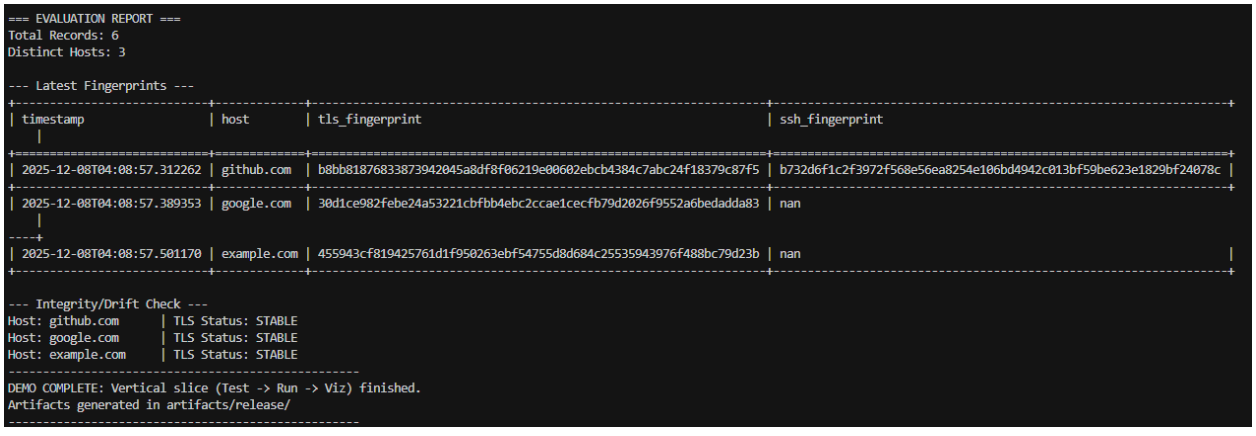
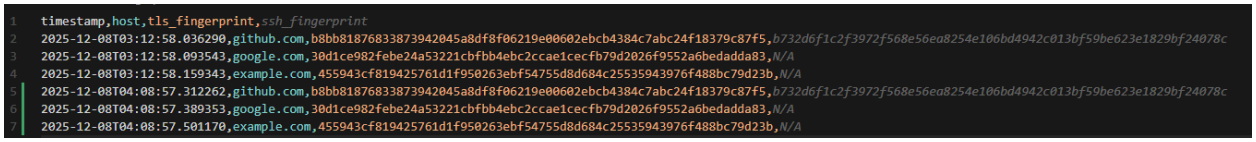


Figure 3: Final Output Table showing stable fingerprints for all targets.



```
timestamp,host,tls_fingerprint,ssh_fingerprint
2025-12-08T03:12:58.036290,github.com,b8bb81876833873942045a8df8f06219e00602ebcb4384c7abc24f18379c87f5,b732d6f1c2f3972f568e56ea8254e106bd4942c013bf59be623e1829bf24078c
2025-12-08T03:12:58.093543,google.com,30d1ce982febe24a53221cbfbb4ebc2ccae1cecfb79d2026f9552a6bedadda83,N/A
2025-12-08T03:12:58.159343,example.com,455943cf819425761d1f950263ebf54755d8d684c25535943976f488bc79d23b,N/A
2025-12-08T04:08:57.312262,github.com,b8bb81876833873942045a8df8f06219e00602ebcb4384c7abc24f18379c87f5,b732d6f1c2f3972f568e56ea8254e106bd4942c013bf59be623e1829bf24078c
2025-12-08T04:08:57.389353,google.com,30d1ce982febe24a53221cbfbb4ebc2ccae1cecfb79d2026f9552a6bedadda83,N/A
2025-12-08T04:08:57.501170,example.com,455943cf819425761d1f950263ebf54755d8d684c25535943976f488bc79d23b,N/A
```

Figure 4: The persistent CSV log generated by the orchestrator.

6. Limitations & Future Work

While the v1.0.0 release is feature-complete, there are limitations:

- **Protocol Support:** The SSH collector currently defaults to RSA keys. Future versions should support ECDSA and Ed25519.
- **Port Blocking:** The system relies on active scanning. Network firewalls (as seen with Google) limit the visibility of SSH keys on web servers.
- **Future Feature - JA3:** We plan to implement JA3 client fingerprinting to analyze the specific TLS "Client Hello" packets, allowing us to identify *who* is connecting to our servers, not just identify the servers themselves.

7. Conclusion

The SSH/TLS Fingerprint Visualizer successfully demonstrates a reproducible, secure method for auditing network identity. By containerizing the solution and implementing strict input validation, we have created a tool that provides immediate visibility into the cryptographic state of network assets, fulfilling the security requirement of "Trust but Verify."