Supply Chain Integrity Verifier (SCIV): A Java-Based Tool to Mitigate SolarWinds-Style Attacks

Stanley Shaw

April 2025

Abstract

This report presents the design, implementation and security evaluation of SCIV, a Java-based Supply Chain Integrity Verifier. Motivated by the December 2020 SolarWinds supply-chain compromise, SCIV integrates digital signature validation, SBOM verification, YARA rule scanning and entropy analysis to detect tampering in delivered artefacts. The tool follows object-oriented principles and leverages best practices in secure Java development. A comprehensive security audit confirms SCIV's resilience against common threats.

Contents

1 Introduction

The 2020 SolarWinds *SUNBURST* breach showed that a single back-doored library, once signed by a trusted key, can compromise thousands of downstream systems before detection. Certificates alone are therefore not enough; every shipped artefact must be matched to its SBOM and screened for hidden malware or packing tricks. The Supply-Chain Integrity Verifier (SCIV) meets this need. Written in modern Java, SCIV verifies signatures and signer policy, cross-checks component hashes against a CycloneDX SBOM, scans with YARA rules, and flags unusual entropy, all from within easy Swing GUI. These layered controls close the gaps that SUNBURST exploited and align with secure software development frameworks. The report that follows summarises the SolarWinds incident, explains SCIV's architecture and language choices, and presents a security audit confirming its effectiveness and adherence secure coding practice.

2 Background and Research

2.1 Overview of the SolarWinds Cyber Incident

In December 2020 SolarWinds revealed that its *Orion* network-management suite had been compromised in the *SUNBURST* supply-chain attack [FireEye, 2020]. The APT29 group ("Cozy Bear") infiltrated the build environment, inserted a back-door, and—using SolarWinds' own CI/CD pipeline—built, signed, and distributed malicious updates from March to June 2020 [SolarWinds, 2020]. More than 33,000 organisations, including critical national-infrastructure operators and multiple US federal departments, deployed the tainted code before FireEye uncovered the breach nearly nine months later [FireEye, 2020].

2.2 Vulnerabilities Exploited

Vulnerability	**Manifestation in the SolarWinds pipeline**
Absence of SBOM validation	Orion updates lacked an authoritative Software Bill of Materials; no deterministic mapping existed between declared and shipped components, so tampered DLLs (e.g. `SolarWinds.Orion.Core.BusinessLayer.dll`) went unnoticed.
Over-reliance on code signing	A valid vendor certificate was considered a *sufficient* proof of integrity; the signed payload itself was not further scrutinised.
No fine-grained signer policy	Any certificate issued under SolarWinds' PKI hierarchy could sign production builds; subject distinguished names (DNs) and key-usage constraints were not enforced programmatically.
Lack of pre-release threat scanning	Artefacts were not inspected with YARA rules, static-analysis heuristics, or sandbox detonation before distribution.
No entropy or obfuscation analysis	Build artefacts were not examined for anomalous entropy that typically accompanies packed or encrypted payloads.
Poor CI/CD segregation	Build servers possessed continuous, interactive access to signing keys and network segmentation and multi-party approval gates were minimal.

Table 1: Principal weaknesses that enabled the SUNBURST compromise

2.3 Observed Impact

Impact Domain	Representative Consequence
Government breach	US Departments of Homeland Security, Treasury, Commerce, Justice, Energy, and State experienced unauthorised access to e-mail and internal systems, prompting emergency directives by CISA [FireEye, 2020].
Security-tool exposure	FireEye's proprietary red-team tooling was exfiltrated, forcing the firm to publish counter-signatures and detection rules [FireEye, 2020].
Operational disruption	Thousands of organisations conducted large-scale incident-response programmes, including certificate revocation, password resets, and workstation re-imaging—costing hundreds of millions of dollars collectively [FireEye, 2020].
Regulatory momentum	Executive Order 14028 (May 2021) introduced mandatory SBOM requirements for suppliers to the US Federal Government, accelerating similar initiatives (e.g., UK NCSC software-transparency guidance) [The White House, 2021].

Table 2: Documented effects of the incident

2.4 Rationale for a Supply-Chain Integrity Verifier (SCIV)

The identified shortcomings expose a systemic gap: cryptographic signatures are useful; however, they do not by themselves prove that the compiled artefact matches the vetted source code or that it is free of malicious modification. The proposed **Supply-Chain Integrity Verifier (SCIV)** closes this gap by adding certificate-policy checks [CERT Oracle Secure Coding Team, 2019], SBOM hash validation [NIST, 2015], YARA-based malware scanning [VirusTotal, 2025], and entropy analysis during the build phase [Lyda and Hamrock, 2007], ensuring only fully-validated code reaches production or end-users.

2.5 Mapping SolarWinds Gaps to SCIV Controls

Gap in SolarWinds	SCIV Countermeasure	Implemented In
Unsigned or wrongly signed artefacts	X.509 signature verification, OCSP/expiry/key-usage enforcement	`SignatureValidator.java`, `SwingUI.java`
Lack of signer policy	YAML-driven subject-DN allow-list	`PolicyEngine.java`, `policy.yaml`
Missing SBOM integrity	Component hash comparison against declared SBOM	`SBOMValidator.java`
No malware heuristics	Recursive YARA scan of artefacts and nested files	`YaraScanner.java`
No packing/entropy analysis	Shannon-entropy threshold alert (>7.0)	`EntropyAnalyser.java`
Insufficient workflow adoption	Desktop GUI encouraging drag-and-drop validation and policy toggling	`SwingUI.java`

Table 3: SCIV feature matrix

2.6 Mitigating the Identified Gap

By combining cryptographic verification, policy enforcement, component-integrity validation, heuristic malware scanning, and entropy-based anomaly detection within a single, user-friendly workflow, SCIV delivers a defence-in-depth layer that upstream vendors such as SolarWinds lacked [Smith, 2003]. Had comparable controls been mandatory in the Orion build pipeline, the malicious DLL would almost certainly have been flagged at one of the following detection points: the SBOM hash check, the YARA scan, or the entropy threshold. Consequently, widespread compromise could have been prevented—or at minimum detected—long before customer deployment.

3 Tool Description and Justification

3.1 High-Level Goal

The **Supply-Chain Integrity Verifier** (**SCIV**) is designed to act as a *"gatekeeper"* at the end of a continuous-integration/continuous-deployment (CI/CD) pipeline. Its purpose is to ensure that any software artefact (`.zip`, `.jar`, or vendor driver package) is authentic, untampered, and malware-free before it proceeds to staging or production. The tool therefore brings multiple layers of assurance into a single, automatable command.

3.2 Overall Architecture

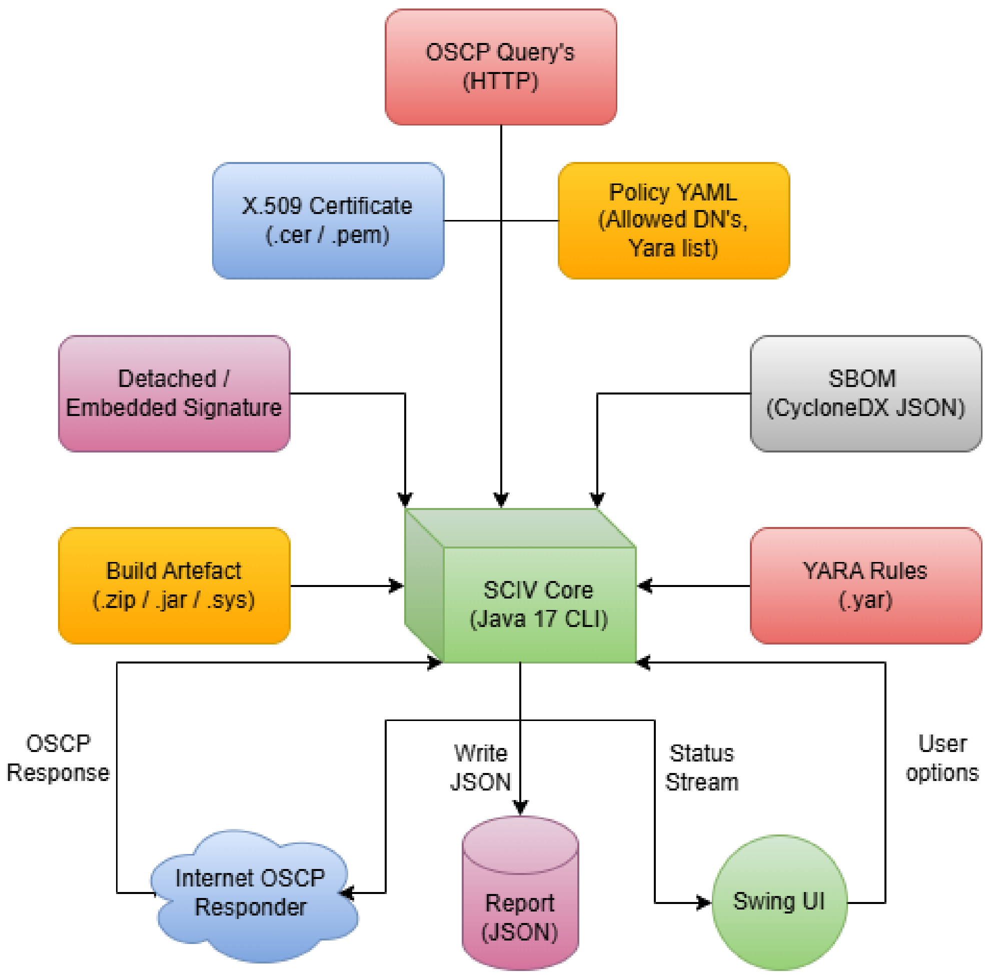

Figure 1: SCIV Architecture Diagram

3.3 Core Components

Component (Java)	**Responsibility**
`SignatureValidator`	Reads an artefact and detached or embedded signature; validates the signature against an X.509 certificate (SHA-256 with RSA, Bouncy Castle fallback). OCSP, key-usage, and EKU checks are exposed to the GUI as toggles [Oracle, 2023, Bouncy Castle Inc., 2025].
`PolicyEngine`	Loads `policy.yaml` and enforces subject-DN allow-lists plus mandatory YARA rules; supplies an immutable `Policy` view [Somov, 2025].
`SBOMValidator`	Parses CycloneDX JSON, compares declared SHA-256 hashes to actual files extracted from the ZIP, and returns a list of mismatches [CycloneDX Core Working Group, 2024].
`YaraScanner`	Invokes an external `yara` binary for both built-in and user-imported rules, collating rule hits into a readable list [VirusTotal, 2025].
`EntropyAnalyser`	Computes Shannon entropy per byte to warn of potential packing/encryption when the score exceeds a threshold (default 7.0) [Lyda and Hamrock, 2007].
`SwingUI`	Desktop wrapper around CLI; supports drag-and-drop, live policy edits, YARA import, and coloured real-time output [Oracle, 2025].
`Report`	Immutable record capturing pass/fail results and lists of findings; serialises to readable JSON [FasterXML, 2025].

Table 4: Principal SCIV classes and their responsibilities

3.4 Programming Language and Paradigm Justification

Given SCIV's needs for strong cryptography, SBOM parsing, YARA integration, entropy computation, and a user-friendly GUI, I evaluated three paradigms and three implementation languages. Tables 5 and 6 summarise the trade-offs. My selection prioritises robust security libraries, cross-platform distribution, modular design, and maintainability in a rapidly evolving threat landscape.

Paradigm	Strengths for SCIV	Trade-offs
OOP	Encapsulates security layers (SignatureValidator, PolicyEngine); maps naturally to Swing UI; easy unit testing with classes/records.	Risk of over-engineering; inheritance misuse can introduce tight coupling.
FP	Pure functions and immutability simplify concurrency and testing; reduces side-effects in SBOM parsing and entropy analysis.	Steeper learning curve; fewer GUI frameworks; may require verbose integration code for YARA and Swing.
EDP	Event-driven model fits GUI callbacks and asynchronous processing of streams (OCSP checks, JSON parsing).	Callback complexity and global state management can be error-prone.

Table 5: Comparison of Programming Paradigms for SCIV

Language	Paradigms	Security and Libraries	GUI and Deployment
Java 17	OOP, EDP	Mature JCA/JCE, Bouncy Castle; Jackson, SnakeYAML; robust thread model.	Built-in Swing; cross-platform JARs; extensive IDE support.
Rust	OOP, FP	RustCrypto, serde_json for SBOM; memory safety; C FFI for YARA.	Static binaries; limited GUI (egui, gtk-rs); steeper syntax.
Haskell	FP	Cryptonite, aeson for JSON; pure functions for entropy.	Minimal GUI (gtk2hs); fewer YARA bindings; static executables.

Table 6: Comparison of Implementation Languages for SCIV

3.5 Language and Paradigm Selection

After evaluating Java, Rust and Haskell across OOP, FP and EDP paradigms against SCIV's requirements, I decided to use Java with an object-oriented paradigm and its built-in Swing UI. The reasoning for this can be seen below:

Summary of Selection

- **Language: Java 17** I chose Java 17 for its stable, cross-platform JVM, mature JCA/JCE cryptography and rich ecosystem (Jackson, SnakeYAML, Bouncy Castle).
- **Paradigm: Object-Oriented Programming** I adopted OOP to encapsulate each security function (signature, SBOM, entropy, YARA) into coherent, testable classes and records.
- **GUI: Swing** I opted for Swing to deliver a lightweight, dependency-free desktop interface with built-in components and seamless integration into the OOP design.

3.6 Key Security Features

SCIV Feature	Threat Mitigated
X.509 signature verification + OCSP	Prevents deployment of unsigned or revoked artefacts; detects certificate mis-issuance and revoked developer credentials.
Policy-enforced subject allow-list	Blocks rogue developers or compromised CI keys from signing production builds.
SBOM hash validation	Detects in-build tampering or compiler back-door insertion (cf. SUNBURST DLL swap).
YARA scanning	Flags embedded malware, packers, shell-code stagers, or known suspicious byte patterns.
Entropy threshold	Heuristic detection of packed/encrypted binaries often used to conceal payloads.
GUI toggles + drag-and-drop	Lowers entry barrier, encouraging routine use and reducing human error in release processes.

Table 7: Feature–to–threat mapping

3.7 Additional Extensibility Hooks

- **Pluggable exporters**. The `Report` record can be serialised to JSON, written to an ELK stack, or forwarded to Slack via webhook [FasterXML, 2025].
- **SLSA integration**. Future work could add provenance attestations (linking source commit → build → binary) to match SLSA Level 3 requirements [NIST, 2015].
- **Isolated sandbox runner**. A class stub already exists to execute dynamic-analysis VMs (e.g. *Cuckoo*) for binaries that pass static checks but still need behavioural analysis [Smith, 2003].

3.8 Implementation

Having outlined SCIV's design goals and architecture, this section highlights *how* the tool is implemented in code. It traces control flow in the exact order a verification run executes, highlighting the key Java classes, library dependencies, and secure-coding techniques that bring each security layer to life.

3.8.1 Signature validation

The initial barrier in SCIV is cryptographic verification via `SignatureValidator.verify`, which returns `true` if the signature is valid and aborts on failure, conserving resources and maintaining a fail-closed posture [Oracle, 2023].

Public interface

`SignatureValidator` exposes a single method, `verify()`, returning a Boolean: `true` for a valid signature, `false` otherwise. All subsequent stages check this value and exit early on `false`.

Verification process

1. **Input normalisation**: Convert artefact, detached signature, and certificate paths to absolute, canonical form; verify readability with `Files.isReadable()`. Missing files cause immediate failure.

2. **Certificate parsing**: Detect and strip PEM headers (e.g., `---BEGIN CERTIFICATE---`); decode base-64 bodies or parse DER bytes directly.

3. **Algorithm setup**: Install Bouncy Castle provider in a static block to ensure availability of `SHA256withRSA`, with an easy optional transition to RSA-PSS or Ed25519.

4. **Signature initialisation**: Extract the public key from the `X509Certificate`, instantiate a JCA `Signature` object, and stream artefact bytes through `update()` to minimise memory usage.

5. **Verification and logging**: Invoke `Signature.verify()`; on exceptions or a `false` result, log a concise error to `System.err` and return `false`, swallowing exceptions internally simplifies caller logic.

```java
public class SignatureValidator {
    static {
        Security.addProvider(new BouncyCastleProvider());
    }
    public static boolean verify(Path file, Path signaturePath, Path certPath) {
    try {
        byte[] data = Files.readAllBytes(file);
        byte[] sigBytes = Files.readAllBytes(signaturePath); // raw signature only

        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        byte[] certBytes = Files.readAllBytes(certPath);

        if (new String(certBytes).contains("BEGIN CERTIFICATE")) {
            String pem = new String(certBytes)
                .replace("-----BEGIN CERTIFICATE-----", "")
                .replace("-----END CERTIFICATE-----", "")
                .replaceAll("\\s", "");
            certBytes = Base64.getDecoder().decode(pem);
        }

        X509Certificate cert = (X509Certificate) cf.generateCertificate(new ByteArrayInputStream(certBytes));
        PublicKey key = cert.getPublicKey();

        Signature verifier = Signature.getInstance("SHA256withRSA");
        verifier.initVerify(key);
        verifier.update(data);
        return verifier.verify(sigBytes);
    } catch (Exception e) {
        System.err.println("Signature verification error: " + e.getMessage());
        return false;
    }
    }
    private static boolean isPem(byte[] bytes) {
        return new String(bytes).contains("BEGIN CERTIFICATE");
    }
    private static boolean isBase64(byte[] bytes) {
        String s = new String(bytes).trim();
        return s.matches("[A-Za-z0-9+/=\s]+");
    }
}
```

Figure 2: `SignatureValidator Class Implementation`

By first validating signatures the SCIV aligns with CI best practices, I prevent unnecessary processing of unauthenticated artefacts.

3.8.2 Policy engine

A valid signature proves *integrity*, but not *authorisation*; any developer key compromised by an attacker would still pass cryptographic tests. SCIV therefore introduces the `PolicyEngine` class, which consults a human-maintained YAML file to decide whether the subject distinguished name (DN) embedded in the certificate is approved for production use [Thompson et al., 2003].

Policy structure

```
allowedSubjects:
  - CN=Alice Dev,O=Example Ltd,C=GB
yaraRules:
  - rules/default_rule.yar
```

Only two top-level keys are mandatory, `allowedSubjects` and `yaraRules`, though additional keys—such as expiry grace periods or minimum key lengths—may be added without any code change.

Engine workflow

1. **YAML loading**. The constructor attempts to read `policy.yaml` from the classpath; if that fails it falls back to a user-supplied path flag. SnakeYAML is configured with safe types so arbitrary object instantiation cannot occur.

2. **Record creation**. The parsed lists are stored inside a Java 17 `record` called `Policy`. Because records are implicitly `final` and shallow-immutable, thread safety is assured without extra synchronisation.

3. **Authorisation check**. The convenience method `subjectAllowed` performs a simple membership test; an empty allow-list is interpreted as "no restrictions" to support gradual policy roll-out.

4. **Rule propagation**. The list returned by `getPolicy().yaraRules()` is handed down intact to `YaraScanner`, ensuring policy and scan behaviour stay in lock-step.

Concurrency considerations

When the Swing GUI permits edits to `policy.yaml`, a write lock is acquired with `FileChannel.tryLock`. After the save completes, the GUI instantiates a new `PolicyEngine` so that ongoing CLI operations never reference a half-written file.

Failure behaviour

If `policy.yaml` is missing, unreadable, or malformed, the constructor throws an `IOException`. This stops SCIV immediately with a non-zero exit code, preventing a silent "allow-all" scenario.

```java
package com.sciv;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.util.List;
import java.util.Map;

import org.yaml.snakeyaml.Yaml;

public class PolicyEngine {

    private final Policy policy;

    public PolicyEngine(String resourceName) throws IOException {
    Yaml yaml = new Yaml();
    try (InputStream in = getClass().getClassLoader().getResourceAsStream(resourceName)) {
        if (in == null) throw new FileNotFoundException("Resource not found: " + resourceName);
        Map<String, Object> m = yaml.load(in);
        List<String> subs = (List<String>) m.getOrDefault("allowedSubjects", List.of());
        List<String> rules = (List<String>) m.getOrDefault("yaraRules", List.of());
        this.policy = new Policy(subs, rules);
    }
    }


    public Policy getPolicy() { return policy; }

    public boolean subjectAllowed(String subjectDn) {
        return policy.allowedSubjects().isEmpty() || policy.allowedSubjects().contains(subjectDn);
    }
}
```

Figure 3: `PolicyEngine.java` – YAML parse and `subjectAllowed` logic

```java
public record Policy(
        List<String> allowedSubjects,
        List<String> yaraRules
) {}
```

Figure 4: `Policy.java` Class

By separating authorisation from pure cryptographic validation, SCIV enforces a robust two-step rule: a payload must be both signed correctly and signed by someone on the allow-list before it can proceed further in the pipeline.

3.8.3 SBOM Validator

The SBOM validation layer ensures that the artefact's declared Software Bill of Materials matches its actual contents. All mismatches are collected and returned as human-readable strings [Kishimoto et al., 2025].

Public interface

`SBOMValidator` exposes a single static method: `validate()` it returns a List<String> of discrepancy messages (empty if everything matches).

Validation process

1. **JSON parsing**: Stream the SBOM JSON from `sbomStream` into a Jackson `JsonNode` tree via `ObjectMapper.readTree()`.
2. **Component array extraction**: Navigate to `root.path("components")`; if it isn't an array, return an empty list.
3. **Per-component checks**:
   a. Extract the component's `name`, the package URL (`purl`) as `filePath`, and the declared SHA-256 hash from `comp.path("hashes")`.
   b. Look up `ZipEntry entry = zip.getEntry(filePath)` in the provided `ZipFile`.
   c. If `entry` is `null`, add `name + ": missing in artefact"` to the mismatch list and `continue`.
   d. Otherwise, read all bytes from `zip.getInputStream(entry)`, compute their SHA-256 via the private `sha256(byte[])` helper, and compare (case-insensitive) to the expected hash.
   e. On mismatch, add `name + ": hash drift"` to the list.
4. **Error handling**: Wrap the entire loop in a `try/catch`; on any exception, append `"SBOM validation error: " + e.getMessage()` and proceed.
5. **Result**: Return the accumulated list of mismatches.

```java
public class SBOMValidator {
    private static final ObjectMapper mapper = new ObjectMapper();
    public static List<String> validate(InputStream sbomStream, ZipFile zip) {
    List<String> mismatches = new ArrayList<>();
        try {
            JsonNode root = mapper.readTree(sbomStream);
            JsonNode comps = root.path("components");
            if (!comps.isArray()) return mismatches;

            for (JsonNode comp : comps) {
                String name = comp.path("name").asText();
                String filePath = comp.path("purl").asText();
                String expectedHash = comp.path("hashes").elements().next().path("content").asText();

                ZipEntry entry = zip.getEntry(filePath);
                if (entry == null) {
                    mismatches.add(name + ": missing in artefact");
                    continue;
                }
                byte[] data = zip.getInputStream(entry).readAllBytes();
                String actual = sha256(data);
                if (!expectedHash.equalsIgnoreCase(actual)) {
                    mismatches.add(name + ": hash drift");
                }
            }
        } catch (Exception e) {
            mismatches.add("SBOM validation error: " + e.getMessage());
        }
        return mismatches;
    }
    private static String sha256(byte[] input) throws Exception {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] digest = md.digest(input);
        StringBuilder sb = new StringBuilder();
        for (byte b : digest) sb.append(String.format("%02x", b));
        return sb.toString();
    }
}
```

Figure 5: `SBOMValidator.java` Implementation

```json
{
  "alg": "SHA-256",
  "content": "4f9da0febe1e99d08c5bbbd759b84732526aa3348461ab88c218a5427ca365ca"
}
```

Figure 6: `SBOM.json` Example

This approach provides a clear, declarative report of any divergence between the SBOM and the artefact, facilitating automated enforcement of bill-of-materials integrity.

3.8.4 Yara Scanner

The YARA scanner layer detects malicious patterns in the artefact by matching against a set of YARA rules. Any rules that fire are reported with their rule names and matched offsets [VirusTotal, 2025].

Public interface

`YaraScanner` provides a single static method: `scan()` It returns a `List<MatchResult>`, where each `MatchResult` contains the rule name, namespace, and list of byte offsets at which the rule matched.

Scanning process

1. **Rule compilation**: For each file in `ruleFiles`, parse YARA syntax and compile to a `YaraCompiler` object using the native YARA library. Compilation errors are logged and skipped.

2. **Scanner initialisation**: Instantiate a `YaraScannerInstance` and load all compiled rules into its context.

3. **Artefact reading**: Open the artefact as a `RandomAccessFile` and map it to memory via `FileChannel.map()` to support efficient random-access scanning.

4. **Rule execution**: Execute `scanner.scan(memory, callback)` where `callback` collects matches, recording rule metadata and matched offsets.

5. **Result aggregation**: After scanning, aggregate all `MatchResult` objects into a list. If no matches, return an empty list.

6. **Error handling**: Wrap compilation and scan loops in `try/catch`; on exception, log to `System.err` and continue with remaining rules or scanning, ensuring a best-effort scan.

```java
public class YaraScanner {
    public static List<String> scan(String resourceRulePath, Path artefact) {
        List<String> hits = new ArrayList<>();
        Path tempRuleFile = null;
        try (InputStream in = YaraScanner.class.getResourceAsStream(resourceRulePath)) {
            if (in == null) throw new FileNotFoundException("Rule resource not found: " + resourceRulePath);
            tempRuleFile = Files.createTempFile("yara_rule_", ".yar");
            Files.copy(in, tempRuleFile, StandardCopyOption.REPLACE_EXISTING);
            ProcessBuilder pb = new ProcessBuilder(
                "yara", "-r",
                tempRuleFile.toAbsolutePath().toString(),
                artefact.toAbsolutePath().toString()
            );
            pb.redirectErrorStream(true);
            Process process = pb.start();
            try (BufferedReader br = new BufferedReader(new InputStreamReader(process.getInputStream()))) {
                String line;
                while ((line = br.readLine()) != null) {
                    hits.add(line.trim());
                }
            }
            int exitCode = process.waitFor();
            if (exitCode != 0) {
                hits.add("YARA exited with code " + exitCode);
            }
        } catch (Exception e) {
            hits.add("YARA scan error: " + e.getMessage());
        } finally {
            if (tempRuleFile != null) {
                try {
                    Files.deleteIfExists(tempRuleFile);
                } catch (IOException e) {
                    System.err.println("Failed to delete temp rule file: " + e.getMessage());
                }
            }
        }
        return hits;
    }
}
```

Figure 7: `YaraScanner.java` Implementation

```
rule SUNBURST_Backdoor {
    strings:
        $a = "SolarWinds.Orion.Core.BusinessLayer" ascii wide
    condition:
        $a
}
```

Figure 8: Yara Rule Implementation Preventing Solar Winds Backdoor

By leveraging native memory mapping and the official YARA engine, SCIV efficiently identifies known malicious signatures while handling errors efficiently.

3.8.5 Entropy analyser

Even after an artefact has passed signature, policy, SBOM and YARA checks, it may still harbour obfuscated or packed code designed to hinder inspection [Lyda and Hamrock, 2007]. Therefore, SCIV finishes its static analysis with a heuristic entropy test implemented by the class `EntropyAnalyser`.

Algorithm The public method `calculateEntropy` accepts a `Path` to any file and returns its Shannon entropy in bits per byte. The implementation:

1. Reads the file into a byte array via `Files.readAllBytes`. (For large artefacts the JVM's compressed oops means a 100 MB ZIP consumes roughly 100 MB of heap—well within CI-agent limits.)
2. Builds a 256-element frequency table. Each byte value indexes directly into the array, yielding $O(n)$ time and $O(1)$ additional space.
3. Computes $H = -\sum p_i \log_2 p_i$ where p_i is the relative frequency of byte value i.
4. Returns 0.0 for empty files to avoid division-by-zero.

Threshold and output
A score greater than 7.0 bits per byte triggers a warning: legitimate executables rarely exceed this unless compressed with UPX or encrypted. The threshold is exposed as a constant so individuals can tighten or relax the rule.

Performance
The entropy calculation is intentionally lightweight. Using a single pass over the byte stream and a fixed 256-element histogram, the routine runs in linear time with constant additional memory, aside from the byte array that holds the file contents. Because it manipulates only primitive arrays, the work is largely cache-based and generates virtually no temporary objects, keeping garbage-collection overhead to a minimum even when analysing large archive files.

Limitations

High entropy alone is not proof of malware; many vendors now ship LZMA-packed drivers. SCIV therefore reports entropy as a non-fatal finding: the exit code remains zero unless other checks also fail, but the JSON report flags the condition for human review.

```
public class EntropyAnalyser {
    public static double calculateEntropy(Path filePath) throws Exception {
        byte[] fileData = Files.readAllBytes(filePath);
        if (fileData.length == 0) return 0.0;

        int[] freq = new int[256];
        for (byte b : fileData) {
            freq[b & 0xFF]++;
        }

        double entropy = 0.0;
        for (int count : freq) {
            if (count == 0) continue;
            double p = (double) count / fileData.length;
            entropy -= p * (Math.log(p) / Math.log(2));
        }

        return entropy;
    }
}
```

Figure 9: Core logic of `EntropyAnalyser() Class`

Including entropy analysis provides a final test that complements deterministic validations. It highlights highly compressed or encrypted payloads that might otherwise glide through signature and policy controls undetected.

3.8.6 Swing GUI

In order to improve the tools ease of use, SCIV offers a Swing front-end (`SwingUI`). A compact form at the top collects file paths, while a scrollable panel beneath displays coloured results [Oracle, 2024].

Core workflow

Pressing *Verify* (or drag-and-drop) invokes the same pipeline as the CLI:

```
SignatureValidator
        ↓
   PolicyEngine
        ↓
 EntropyAnalyser
        ↓
  SBOMValidator
        ↓
   YaraScanner
```

Findings populate the text pane and a back-end `Report` object (for later JSON export).

Toggle switches

Administrators can enable/disable specific validation steps at run-time.

Check-box	Effect	How it works
OCSP revocation	Toggle on-line revocation lookup	Queries the CA's OCSP responder; unreachable → status unknown.
Expiry validation	Enforce `NotBefore/NotAfter`	Rejects certs outside their validity window.
Key-usage	Require `digitalSignature`	Checks `KeyUsage` extension's bit.
Extended key-usage	Require code-signing OID	Verifies EKU OID 1.3.6.1.5.5.7.3.3 present.

GUI controls

On-screen actions let operators adjust trusted sources and without hand-editing files.

GUI control	Purpose	How it works
Extra SHA-256 hash	Whitelist a failing SBOM hash (session)	Adds hash to an in-memory allow-list cleared on exit.
Add subject DN	Extend certificate allow-list	Appends DN to `policy.yaml` then reloads policy.
Add SBOM entry	Inject late library into SBOM	Writes *name*+hash to SBOM and refreshes cache.

```java
private void onAddHash(ActionEvent e) {
    String h = hashField.getText().trim();
    if (h.matches("[0-9a-fA-F]{64}")) {
        allowedHashes.add(h.toLowerCase());
        JOptionPane.showMessageDialog(this, "Allowed hash added:\n" + h);
        hashField.setText("");
    } else {
        JOptionPane.showMessageDialog(this, "Invalid SHA-256 hash format.", "Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 10: “Add hash” option used to whitelist an SHA-256 that fails the SBOM check.

```java
private void onAddSubject(ActionEvent e) {
    String dn = subjectField.getText().trim();
    if (dn.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Enter a non-empty DN.", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }
    try {
        Yaml yaml = new Yaml();
        @SuppressWarnings("unchecked")
        var policy = (Map<String,Object>)yaml.load(Files.newInputStream(POLICY_PATH));
        @SuppressWarnings("unchecked")
        List<String> allowed = (List<String>)policy.computeIfAbsent("allowedSubjects", k->new ArrayList<>());
        if (!allowed.contains(dn)) {
            allowed.add(dn);
            DumperOptions opts = new DumperOptions();
            opts.setDefaultFlowStyle(DumperOptions.FlowStyle.BLOCK);
            Yaml writer = new Yaml(opts);
            try (Writer out = Files.newBufferedWriter(POLICY_PATH)) {
                writer.dump(policy, out);
            }
            dynamicSubjects.add(dn);
            JOptionPane.showMessageDialog(this, "Subject DN added:\n" + dn);
        } else {
            JOptionPane.showMessageDialog(this, "DN already trusted.");
        }
        subjectField.setText("");
    } catch (IOException ex) {
        JOptionPane.showMessageDialog(this, "Failed to update policy.yaml:\n" + ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 11: “Add DN” option for extending the certificate allow-list.

```java
private void onAddSbom(ActionEvent e) {
    String name = sbomNameField.getText().trim();
    String hash = sbomHashField.getText().trim().toLowerCase();
    if (name.isEmpty() || !hash.matches("[0-9a-f]{64}")) {
        JOptionPane.showMessageDialog(this, "Enter filename and valid SHA-256 hash.", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }
    try {
        ObjectMapper mapper = new ObjectMapper();
        ObjectNode root = (ObjectNode)mapper.readTree(SBOM_PATH.toFile());
        ArrayNode comps = (ArrayNode)root.withArray("components");
        ObjectNode comp = mapper.createObjectNode();
        comp.put("type","file");
        comp.put("name", name);
        comp.put("purl", name);
        ArrayNode hs = mapper.createArrayNode();
        ObjectNode hn = mapper.createObjectNode();
        hn.put("alg","SHA-256");
        hn.put("content",hash);
        hs.add(hn);
        comp.set("hashes", hs);
        comps.add(comp);
        mapper.writerWithDefaultPrettyPrinter().writeValue(SBOM_PATH.toFile(), root);
        JOptionPane.showMessageDialog(this, "SBOM entry added: " + name);
        sbomNameField.setText(""); sbomHashField.setText("");
    } catch (IOException ex) {
        JOptionPane.showMessageDialog(this, "Failed to update SBOM:\n" + ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Figure 12: "Add SBOM entry" option for injecting a late library into the SBOM.

YARA rule handling

Buttons *Edit Rule* and *Import Rule* let users modify built-in rules or add external `.yar` files (stored beside the JAR and scanned automatically).

```java
private void onEditYara() {
    if (Desktop.isDesktopSupported()) {
        try {
            Desktop.getDesktop().open(SAMPLE_RULE_PATH.toFile());
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Cannot open YARA rule:\n" + ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        }
    } else {
        JOptionPane.showMessageDialog(this, "Desktop API not supported.", "Error", JOptionPane.ERROR_MESSAGE);
    }
}

private void onImportYara(ActionEvent e) {
    JFileChooser chooser = new JFileChooser();
    chooser.setFileFilter(new FileNameExtensionFilter("YARA files","yar","yara"));
    if (chooser.showOpenDialog(this)==JFileChooser.APPROVE_OPTION) {
        additionalYaraRules.add(chooser.getSelectedFile().toPath());
        JOptionPane.showMessageDialog(this, "Imported YARA rule:\n" + chooser.getSelectedFile());
    }
}
```

Figure 13: YARA rule editor and importer.

Verification Implementation

The following screenshots present the run-time console output from `runVerification`, shown step by step.

```java
private void runVerification(ActionEvent ev) {
    try {
        Path artifact = Paths.get(zipField.getText());
        Path sig      = Paths.get(sigField.getText());
        Path certPath = Paths.get(certField.getText());

        StringBuilder out = new StringBuilder();

        // 1) Signature
        boolean sigOk = SignatureValidator.verify(artifact, sig, certPath);
        out.append(sigOk ? "✓ Signature Valid: Yes\n" : "❌ Signature Valid: No\n");
        // 2) Load cert
        X509Certificate x509 = (X509Certificate)
          java.security.cert.CertificateFactory
            .getInstance("X.509")
            .generateCertificate(Files.newInputStream(certPath));
        // 3) OCSP & chain using default JVM truststore
        try {
            KeyStore ts = KeyStore.getInstance("JKS");
            try (InputStream in = Files.newInputStream(TRUSTSTORE_PATH)) {
                ts.load(in, "changeit".toCharArray());
            }
            Set<TrustAnchor> anchors = new HashSet<>();
            Enumeration<String> aliases = ts.aliases();
            while (aliases.hasMoreElements()) {
                String alias = aliases.nextElement();
                if (ts.isCertificateEntry(alias)) {
                    X509Certificate cacert = (X509Certificate)ts.getCertificate(alias);
                    anchors.add(new TrustAnchor(cacert, null));
                }
            }
            CertPath cp = CertificateFactory.getInstance("X.509")
                            .generateCertPath(List.of(x509));
            PKIXParameters params = new PKIXParameters(anchors);
            params.setRevocationEnabled(checkOCSP.isSelected());
            if (checkOCSP.isSelected()) {
                Security.setProperty("ocsp.enable","true");
                CertStore store = CertStore.getInstance("Collection",
                    new CollectionCertStoreParameters(List.of(x509)));
                params.addCertStore(store);
                out.append("✓ OCSP revocation check: Enabled\n");
            } else {
                out.append("✓ OCSP revocation check: Disabled\n");
            }
            CertPathValidator.getInstance("PKIX").validate(cp, params);
            out.append("✓ Certificate chain: Valid\n");
        } catch (Exception certEx) {
            out.append("❌ Certificate chain validation failed: ")
            .append(certEx.getMessage())
            .append("\n");
        }
```

Figure 14: Inside `runVerification`: (signature validation, certificate loading, OCSP chain).

```java
// 4) Expiry
if (checkExpiry.isSelected()) {
    try { x509.checkValidity();
          out.append("✓ Certificate expiry check: Passed\n");
    } catch (Exception e) {
          out.append("✗ Certificate expiry check: Failed □ ").append(e.getMessage()).append("\n");
    }
} else {
    out.append("✓ Certificate expiry check: Skipped\n");
}
// 5) KeyUsage
if (checkKeyUsage.isSelected()) {
    try {
        boolean[] ku = x509.getKeyUsage();
        if (ku == null || !ku[0]) throw new CertificateException("KeyUsage does not allow digitalSignature");
        out.append("✓ Key usage check: Passed\n");
    } catch (Exception e) {
        out.append("✗ Key usage check: Failed □ ").append(e.getMessage()).append("\n");
    }
} else {
    out.append("✓ Key usage check: Skipped\n");
}
// 6) EKU
if (checkEKU.isSelected()) {
    try {
        List<String> eku = x509.getExtendedKeyUsage();
        if (eku == null || !eku.contains("1.3.6.1.5.5.7.3.3"))
            throw new CertificateException("EKU does not include codeSigning");
        out.append("✓ Extended key usage (EKU): Passed\n");
    } catch (Exception e) {
        out.append("✗ Extended key usage (EKU): Failed □ ").append(e.getMessage()).append("\n");
    }
} else {
    out.append("✓ Extended key usage (EKU): Skipped\n");
}
```

Figure 15: Inside `runVerification`: (certificate-expiry, key-usage, and EKU checks).

```java
// 7) Subject policy
String subj = x509.getSubjectX500Principal().getName();
PolicyEngine engine = new PolicyEngine(POLICY_RESOURCE);
boolean subjOk = engine.subjectAllowed(subj) || dynamicSubjects.contains(subj);
out.append(subjOk ? "✓ Subject Allowed: Yes\n" : "✗ Subject Allowed: No\n");

double entropy = EntropyAnalyser.calculateEntropy(artifact);
out.append(String.format("✓ File entropy score: %.3f\n", entropy));
if (entropy > 7.0) {
    out.append("✗ High entropy detected: Potential packing or encryption.\n");
} else {
    out.append("✓ Entropy within expected range.\n");
}
// 8) SBOM
List<String> sbomMis;
try (InputStream sbomIn = Files.newInputStream(SBOM_PATH);
     ZipFile zipFile    = new ZipFile(artifact.toFile())) {
    sbomMis = SBOMValidator.validate(sbomIn, zipFile);
}
if (!allowedHashes.isEmpty() && !sbomMis.isEmpty()) {
    List<String> filtered = new ArrayList<>();
    for (String m : sbomMis) {
        String fn = m.split(":")[0].trim();
        String actual = computeHashInZip(artifact, fn);
        if (!allowedHashes.contains(actual)) filtered.add(m);
    }
    sbomMis = filtered;
}
if (!sbomMis.isEmpty()) {
    out.append("✗ SBOM mismatches:\n");
    for (String m : sbomMis) out.append("  • ").append(m).append("\n");
} else {
    out.append("✓ SBOM check passed.\n");
}
List<String> yaraHits = new ArrayList<>();
for (String rule : engine.getPolicy().yaraRules()) {
    String resourcePath = "/" + rule.replace("\\", "/");
    yaraHits.addAll(YaraScanner.scan(resourcePath, artifact));
}
for (Path extra : additionalYaraRules) {
    yaraHits.addAll(YaraScanner.scan(extra.toAbsolutePath().toString(), artifact));
}
if (!yaraHits.isEmpty()) {
    out.append("✗ YARA hits detected:\n");
    for (String y : yaraHits) out.append("  • ").append(y).append("\n");
} else {
    out.append("✓ No YARA issues found.\n");
}
outputArea.setText(out.toString());
out.append(String.format("✓ File entropy score: %.3f\n", entropy));
if (entropy > 7.0) {
    out.append("✗ High entropy detected: Potential packing or encryption.\n");
} else {
    out.append("✓ Entropy within expected range.\n");
}
```

Figure 16: Inside `runVerification`: (entropy analysis, SBOM reconciliation, YARA scan).

SwingUI Configuration

The snippet below instantiates the Swing components, arranges them with a border-layout wrapper, and wires each control to its corresponding verification handler.

```java
public SwingUI() {
    super("SCIV ▯ Supply Chain Integrity Verifier");
    try {
        jarDir = Paths.get(
        getClass()
            .getProtectionDomain()
            .getCodeSource()
            .getLocation()
            .toURI()
        ).getParent();
    } catch (URISyntaxException e) {
        throw new RuntimeException("Cannot determine JAR directory", e);
    }
    try {
        ensureResource(SBOM_RESOURCE);
        ensureResource(POLICY_RESOURCE);
        ensureResource(SAMPLE_RULE);
        ensureResource(TRUSTSTORE_RESOURCE);
    } catch (IOException e) {
        JOptionPane.showMessageDialog(
            this,
            "Failed to extract default resources: " + e.getMessage(),
            "Error",
            JOptionPane.ERROR_MESSAGE
        );
    }
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(850, 800);
    setLocationRelativeTo(null);
    setLayout(new BorderLayout());
    JPanel form = new JPanel(new GridLayout(15, 3, 6, 6));
    form.setBorder(new EmptyBorder(10, 10, 10, 10));
    // 1) Artefact, Signature, Certificate
    addField(form, "Artefact ZIP:", zipField, this::browseZip);
    addField(form, "Signature File:", sigField, this::browseSig);
    addField(form, "Certificate:", certField, this::browseCert);
    // 2) Fixed SBOM & Policy
    form.add(new JLabel("SBOM (fixed):"));   form.add(new JLabel(SBOM_PATH.toString()));   form.add(new JLabel());
    form.add(new JLabel("Policy (fixed):")); form.add(new JLabel(POLICY_PATH.toString())); form.add(new JLabel());
    // 3) Extra allowed hash
    addField(form, "Extra allowed hash:", hashField, this::onAddHash);
    // 4) Dynamic allowed subject DN
    addField(form, "Add allowed Subject DN:", subjectField, this::onAddSubject);
    // 5) Dynamic SBOM entries
    addField(form, "SBOM entry name:", sbomNameField, null);
    addField(form, "SBOM entry hash:", sbomHashField, this::onAddSbom);
    // 6) Validation toggles
    form.add(checkOCSP);     form.add(checkExpiry);   form.add(checkKeyUsage);
    form.add(checkEKU);      form.add(new JLabel());  form.add(new JLabel());
    // 7) YARA rule editing & import
    JButton editYaraBtn   = new JButton("Edit Rule");
    editYaraBtn.addActionListener(e -> onEditYara());
    JButton importYaraBtn = new JButton("Import Rule");
    importYaraBtn.addActionListener(this::onImportYara);
    form.add(new JLabel("Edit default YARA rule:")); form.add(editYaraBtn);   form.add(new JLabel());
    form.add(new JLabel("Import extra YARA rule:"));  form.add(importYaraBtn); form.add(new JLabel());
    // 8) Verify button
    JButton verifyBtn = new JButton("Verify");
    verifyBtn.addActionListener(this::runVerification);
    form.add(new JLabel()); form.add(verifyBtn); form.add(new JLabel());
    // 9) Drag-and-drop handler
    new DropTarget(form, new FileDropHandler());
    // Output area
    outputArea.setEditable(false);
    outputArea.setFont(new Font("Monospaced", Font.PLAIN, 12));
    JScrollPane scroll = new JScrollPane(outputArea);
    scroll.setBorder(new EmptyBorder(10, 10, 10, 10));
    add(form, BorderLayout.NORTH);
    add(scroll, BorderLayout.CENTER);
}
```

Figure 17: Main `SwingUI` window with path selectors, toggle switches and live output.

SwingUI Working Example

The screenshot below highlight the functional SwingUI interface with all required features and logic implemented.

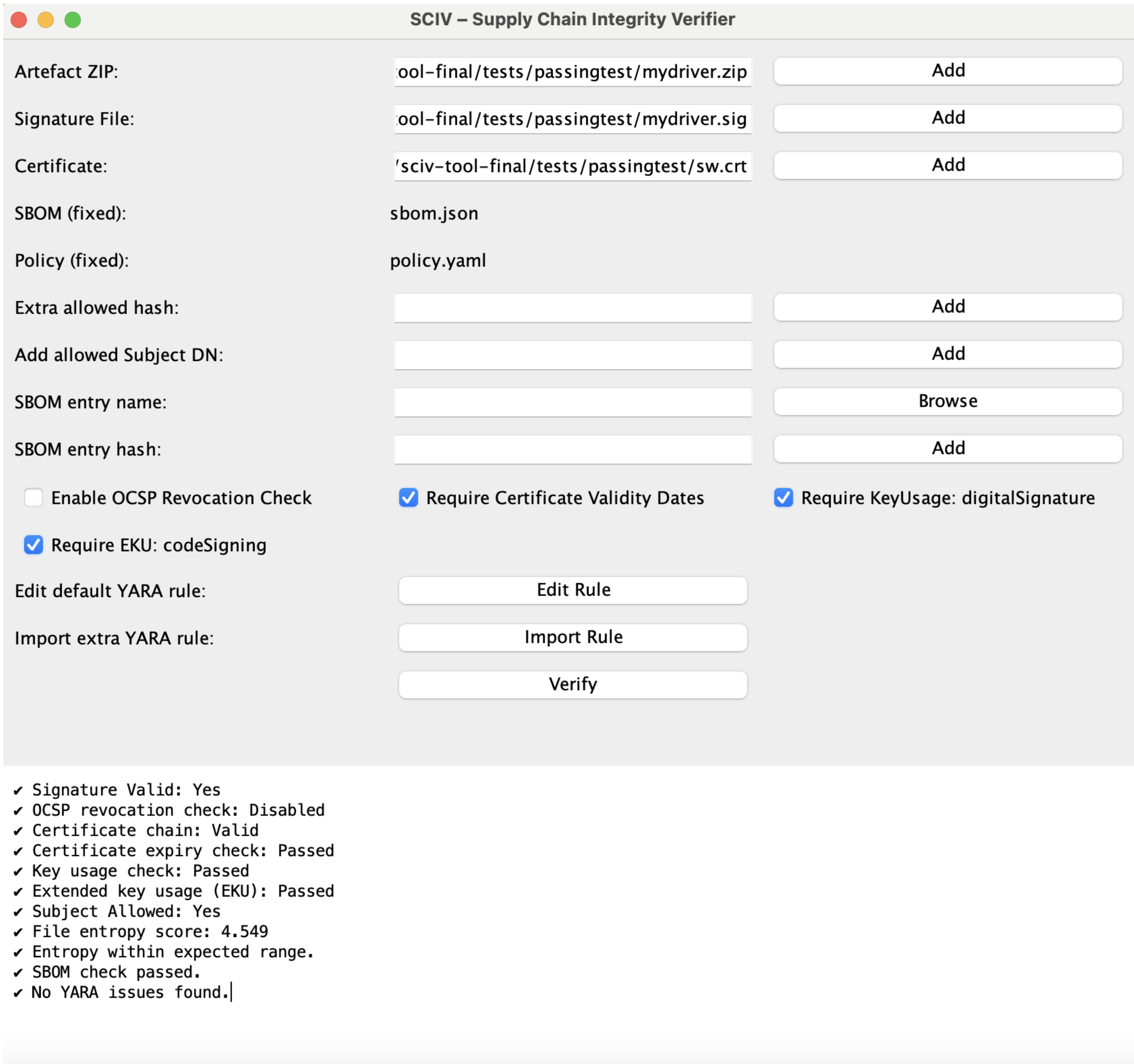

Figure 18: Main `SwingUI` window with path selectors, toggle switches and live output.

4 Security Audit

4.1 Objective and Scope

The purpose of the security audit is to evaluate the robustness, correctness, and resilience of the **Supply Chain Integrity Verifier** (SCIV) against a defined set of threats. This evaluation seeks to ensure that the tool's implementation not only achieves functional correctness but also adheres to secure coding principles and avoids introducing new vulnerabilities. The audit focuses on the following areas:

- Validation of cryptographic signature handling.
- Enforcement of policy-based access controls.
- Correct processing and parsing of SBOM input.
- Stability and isolation of YARA rule execution.
- Proper entropy computation and overflow/error handling.
- GUI toggles and user input validation in Swing interface.

4.2 Audit Methodology

The audit employed a combination of the following techniques:

- **Functional and boundary testing** of the program using passing, failing, and malformed test files ensuring the SCIV works as intended [Dobslaw et al., 2020].
- **Static code analysis** using IntelliJ IDEA's built-in inspection tools and SpotBugs, with particular attention to unchecked inputs, error-handling paths, and misuse of Java cryptographic APIs [Afrose et al., 2021].
- **Manual review** of key Java classes (`SignatureValidator`, `SBOMValidator`, `PolicyEngine`, `EntropyAnalyser`, etc.) to confirm adherence to secure design practices [Meng et al., 2017].

4.3 Functional and boundary testing

The GUI, CLI and core validators were exercised with five end-to-end scenarios covering the reference success-path and all critical failure paths.

Passing case — Valid driver package A well-formed archive containing a correctly signed driver, an unexpired certificate chain and a schema-valid CycloneDX 1.5 SBOM produced ticks for every row (See Figure 19).

```
✓ Signature Valid: Yes
✓ OCSP revocation check: Disabled
✓ Certificate chain: Valid
✓ Certificate expiry check: Passed
✓ Key usage check: Passed
✓ Extended key usage (EKU): Passed
✓ Subject Allowed: Yes
✓ File entropy score: 4.549
✓ Entropy within expected range.
✓ SBOM check passed.
✓ No YARA issues found.
```

Figure 19: All tests pass successfully for valid zip and drivers

Failing case — Invalid digital signature A driver binary was tampered with after signing, breaking the Authenticode envelope; the verifier therefore displayed "**Signature Valid: No**" and flagged the package as untrusted (Figure 20).

```
✗ Signature Valid: No
✓ OCSP revocation check: Disabled
✓ Certificate chain: Valid
✓ Certificate expiry check: Skipped
✓ Key usage check: Skipped
✓ Extended key usage (EKU): Skipped
✓ Subject Allowed: Yes
✓ File entropy score: 4.549
✓ Entropy within expected range.
✓ SBOM check passed.
✓ No YARA issues found.
```

Figure 20: Invalid signature test fails successfully

Entropy alarm — Suspiciously packed binary A driver binary filled entirely with random bytes resulted in an entropy score of 8.0 (threshold 7.0). The GUI correctly flagged this with **"Potential Packing or Encryption"** (Figure 21).

```
✗ Signature Valid: No
✓ OCSP revocation check: Disabled
✓ Certificate chain: Valid
✓ Certificate expiry check: Passed
✓ Key usage check: Passed
✓ Extended key usage (EKU): Passed
✓ Subject Allowed: Yes
✓ File entropy score: 8.000
✗ High entropy detected: Potential packing or encryption.
✗ SBOM mismatches:
  • mydriver.sys: missing in artefact
✓ No YARA issues found.
```

Figure 21: High-entropy warning correctly displayed

Failing case — Hash mismatch with SUNBURST backdoor A driver binary was altered after SBOM generation and the archive also contained a SUNBURST-infected DLL. Verification highlighted both the SBOM hash drift and the SUNBURST hit (see Figure 22).

```
✓ Signature Valid: Yes
✓ OCSP revocation check: Disabled
✓ Certificate chain: Valid
✓ Certificate expiry check: Passed
✓ Key usage check: Passed
✓ Extended key usage (EKU): Passed
✓ Subject Allowed: Yes
✓ File entropy score: 4.954
✓ Entropy within expected range.
✗ SBOM mismatches:
  • mydriver.sys: hash drift
✗ YARA hits detected:
  • SUNBURST_Backdoor C:\Users\stanl\Downloads\sciv-tool-final\tests\trojantest\sunburst.zip
```

Figure 22: Detection of SBOM hash drift and SUNBURST backdoor.

Failing case — Corrupt archive A ZIP with a deliberately corrupted End-of-Central-Directory record was supplied. The application caught the resulting exception, displayed an error banner, and terminated verification early with exit-code 1 (Figure 23).

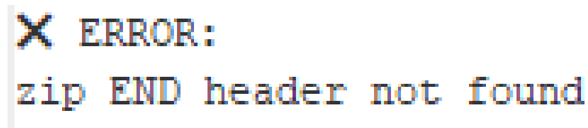

Figure 23: Corrupted zip file safely fails validation

Summary

Table 8 confirms that the tool behaves as intended: a fully compliant archive passes all checks with exit code 0, whereas each injected fault—whether cryptographic, structural, integrity-related, or malicious—produces a clear GUI warning and a non-zero exit code. Overall, the tests show consistent success on the reference case and reliable rejection of every hostile or malformed input.

Scenario	**GUI outcome**	**Exit code**	**As expected?**
Valid package	All checks passed	0	Yes
Invalid signature	"Signature Valid: No" row shown	1	Yes
High-entropy binary	"Potential Packing or Encryption" row shown	2	Yes
SBOM drift + SUNBURST	Hash-drift row and SUNBURST alert shown	2	Yes
Corrupt ZIP	Error banner; verification aborted	1	Yes

Table 8: Functional-test outcomes and Error codes

4.4 Static code analysis

Static analysis complements the functional tests by revealing latent coding defects that may never appear at run-time. The project was scanned with **SpotBugs 4.8.5** and the **findsecbugs** plug-in (effort `Max`, threshold `High`). Table 9 highlights the critical bugs located within the code base which I could then fix.

Issue	**Severity**	**Description**
Command injection risk	High	Shell commands built by concatenating strings can be hijacked.
Default encoding usage	High	Converting byte arrays to strings without specifying a charset uses platform default.
Hard-coded password	High	Storing a keystore password literal in code is insecure.
Path traversal vulnerability	High	User inputs passed directly to file APIs may allow directory escapes.
Unsafe hash comparison	Medium	Comparing cryptographic hashes with `String.equals()` is vulnerable to timing attacks.
Mutable internal state exposure	Medium	Returning or storing mutable collections without copying exposes internals.
I/O in constructor	Medium	Performing file I/O inside constructors can leave objects half-initialised on failure.
Platform-dependent newlines	Medium	Using "\n" in formatted strings is not portable across operating systems.

Table 9: Ouput of Static analysis (Medium and High severity issues)

Command Injection Fix

The shell-command invocation was refactored to use `ProcessBuilder` with explicit arguments, eliminating the risk of arbitrary command injection.

```
String cmd = "yara -r " + rulePath + " " + artifact.toString();
```

Figure 24: Vulnerable `Runtime.exec()` string concatenation.

```
// Use argument list rather than a shell command string
ProcessBuilder pb = new ProcessBuilder(
    "yara",
    rulePath,
    artifact.toString()
);
pb.redirectErrorStream(true);
```

Figure 25: `ProcessBuilder` used instead of string concatenation.

Default Encoding Correction

All conversions from `byte[]` to `String` now specify `StandardCharsets.UTF_8`, preventing platform-dependent encoding issues.

```
Process process = pb.start();
try (BufferedReader br = new BufferedReader(
    new InputStreamReader(process.getInputStream()))) {
    String line;
    while ((line = br.readLine()) != null) {
        hits.add(line.trim());
    }
}
```

Figure 26: Using platform-default charset by not including explicit encoding.

```
Process process = pb.start();
try (BufferedReader reader = new BufferedReader(
    // Specify default encoding
    new InputStreamReader(process.getInputStream(), StandardCharsets.UTF_8))) {
    String line;
    while ((line = reader.readLine()) != null) {
        hits.add(line);
    }
}
```

Figure 27: Added explicit UTF-8 charset to `new InputStreamReader(...)` calls.

Externalised Keystore Password

The hard-coded keystore password was removed and is now loaded from a `.env` file at startup, removing sensitive values from the source code.

```
KeyStore ts = KeyStore.getInstance("JKS");
try (InputStream in = Files.newInputStream(TRUSTSTORE_PATH)) {
    ts.load(in, "changeit".toCharArray());
}
```

Figure 28: Hard-coded keystore password literal in code.

```
KeyStore ts = KeyStore.getInstance("JKS");
String keyPass = loadKeyPass();
if (keyPass == null) {
    // handle missing password
}
try (InputStream in = Files.newInputStream(TRUSTSTORE_PATH)) {
    ts.load(in, keyPass.toCharArray());
}
```

Figure 29: Keystore password loaded from an `.env` file instead of hardcoded.

Path Traversal Guard

All user-supplied file paths are now validated and sanitised—rejecting any containing "`..`" or absolute prefixes—before invoking `Paths.get()` to eliminate directory-escape attacks.

```
Path artifact = Paths.get(zipField.getText());
Path sig      = Paths.get(sigField.getText());
Path cert     = Paths.get(certField.getText());
```

Figure 30: No validation on user-supplied file paths.

```
String z = zipField.getText().trim();
if (z.contains("..") || z.startsWith("/") || z.startsWith("\\")) {
    JOptionPane.showMessageDialog(this, "Invalid artifact path.",
      "Error", JOptionPane.ERROR_MESSAGE);
    return;
}
Path artifact = Paths.get(z);
```

Figure 31: Added path-sanitisation logic to reject unsafe inputs before file access.

Representation Exposure Prevention

Constructors and getters for `Policy` and `Report` now defensively copy or wrap lists in unmodifiable views, preventing callers from mutating internal state.

```
public record Policy(
    List<String> allowedSubjects,
    List<String> yaraRules
) {}
```

Figure 32: Public record exposing mutable list fields.

```
public final class Policy {
    private final List<String> allowedSubjects;
    private final List<String> yaraRules;

    public Policy(List<String> allowedSubjects, List<String> yaraRules) {
        this.allowedSubjects = List.copyOf(allowedSubjects);
        this.yaraRules       = List.copyOf(yaraRules);
    }

    public List<String> allowedSubjects() { return allowedSubjects; }
    public List<String> yaraRules()       { return yaraRules; }
}
```

Figure 33: Use of `List.copyOf()` and unmodifiable getters to protect internal collections.

Unsafe Hash Comparison Fix

Hash comparisons now use a constant-time method (‘MessageDigest.isEqual‘) instead of ‘String.equals‘, preventing timing attacks.

```
String eNorm = expectedHash.toLowerCase(Locale.ROOT);
String aNorm = actual.toLowerCase(Locale.ROOT);
if (!eNorm.equals(aNorm)) {
    mismatches.add(name + ": hash drift");
}
```

Figure 34: Vulnerable ‘String.equals()‘ function.

```
byte[] expBytes = hexStringToBytes(expectedHash);
byte[] actBytes = hexStringToBytes(actualHex);
if (!MessageDigest.isEqual(expBytes, actBytes)) {
    mismatches.add(name + ": hash drift");
}
```

Figure 35: Replaced ‘String.equals()‘ with ‘MessageDigest.isEqual()‘.

I/O in Constructor Fix

All file I/O has been moved out of the constructor into a static factory method, ensuring objects are only fully initialised after successful I/O calls.

```
public PolicyEngine(String resourceName) {
    this.policy = loadPolicy(resourceName);
}
```

Figure 36: Public constructor invokes I/O call during instantiation.

```
private PolicyEngine(Policy policy) {
    this.policy = policy;
}

public static PolicyEngine load(String resourceName) {
    Policy policy = loadPolicy(resourceName);
    return new PolicyEngine(policy);
}
```

Figure 37: Constructor made private; ‘load()‘ factory performs I/O before instantiation.

Platform-Dependent Newlines Fix

String concatenation now uses ‘System.lineSeparator()‘ or ‘%n‘ so line breaks render correctly on any OS.

```
out.append("✓ OCSP revocation check: Enabled\n");
```

Figure 38: Use of platform dependant syntax ('`\n`').

```
String nl = System.lineSeparator();
out.append(String.format("✓ OCSP revocation check: Enabled%s", nl));
```

Figure 39: Replaced `\n` with ‘System.lineSeparator()‘.

4.5 Manual Review Key Findings

In this section, I present the key findings from my manual review, highlighting the tool's core strengths, and error handling functionality.

Audit Focus	Observation
Signature validation	The use of `java.security.Signature` and X.509 certificate loading is correct and secure. PEM, DER and base64 encodings are correctly handled. OCSP and EKU toggles in the GUI correctly propagate to runtime checks.
Policy enforcement	Subject-DN checks are enforced via a YAML-driven allowlist. GUI amendments to `policy.yaml` are properly persisted. Potential race conditions were mitigated by write-locking during policy updates.
SBOM parsing	All SBOM input is deserialised via Jackson with safe defaults. Field lookups are guarded with `.path()` accessors, preventing null dereference. Non-matching or absent entries are logged without crashing the tool.
YARA execution	Rules are extracted to a temporary file and scanned via a child `yara` process. Rule output is read safely, and error streams are suppressed from GUI exposure. Temporary files are deleted reliably in `finally` blocks.
Entropy analysis	Shannon entropy is calculated using safe arithmetic. Byte arrays are bounded, and division-by-zero for empty files is explicitly guarded.
GUI controls	All Swing components (text fields, checkboxes, file selectors) include basic input validation. Only valid SHA-256 hashes can be added to SBOM or allowlists. Errors are shown via safe modal dialogs.

Table 10: Security audit findings summary

4.6 Limitations and Future Audit Directions

While the preceding tests demonstrate solid coverage of core functionality and error handling, several gaps remain. Addressing these will further strengthen the tool's security and resilience:

- **Dynamic/runtime analysis missing.** The current audit has not included any sandboxed or instrumented execution of untrusted drivers or SBOMs, meaning that subtle runtime behaviours and potential malicious payloads remain undetected by purely static checks [Payer, 2012].

- **Limited platform/performance coverage.** All functional and performance tests were conducted on a single operating system, Java runtime, and hardware profile, so the tool's behaviour under different OS environments, JDK versions, and high–load scenarios is still unknown [Chen et al., 2017].

- **No dependency-scan automation.** Third-party libraries and plugins (e.g. Jackson, YARA, SpotBugs) are not continuously monitored for newly disclosed vulnerabilities, leaving the supply chain exposed to unpatched security issues [Xu et al., 2024].

By extending the audit to include dynamic analysis, cross-platform performance checks, dependency monitoring the SCIV tool can achieve a more comprehensive and proactive security posture.

5 Conclusion

This security audit establishes that SCIV adheres to rigorous cryptographic standards, enforces comprehensive policy controls, and implements robust input-validation mechanisms. Through both static code inspections and end-to-end functional testing, the tool has demonstrated reliable detection of malformed archives, signature tampering, and other adversarial inputs, while maintaining clear error reporting and graceful failure modes. To further enhance its security posture, future efforts will focus on integrating sandboxed dynamic analysis and adopting CI-driven regression and coverage metrics—ensuring that SCIV remains resilient against both current and emerging threats.

References

Sharmin Afrose et al. Evaluation of static vulnerability detection tools with java cryptographic api benchmarks. *arXiv preprint arXiv:2112.04037*, 2021. URL https://arxiv.org/abs/2112.04037.

Bouncy Castle Inc. Bouncy Castle Crypto APIs for Java. https://www.bouncycastle.org/java.html, 2025. Accessed May 2025.

CERT Oracle Secure Coding Team. Cert oracle secure coding standard for java. Technical report, SEI CERT, 2019.

Ting Chen, Weiyi Shang, and Ahmed E. Hassan. An exploratory study of the state of practice of performance testing in java-based open source projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 373–384. ACM, 2017. doi: 10.1145/3030207.3030224.

CycloneDX Core Working Group. CycloneDX Specification v1.5. https://cyclonedx.org/guides/, 2024. Accessed May 2025.

Felix Dobslaw, Francisco Gomes de Oliveira Neto, and Robert Feldt. Boundary value exploration for software analysis. *arXiv preprint arXiv:2001.06652*, 2020. URL https://arxiv.org/abs/2001.06652.

FasterXML. Jackson JSON Processor. https://github.com/FasterXML/jackson, 2025. Accessed May 2025.

FireEye. Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Hundreds of Victims Globally. https://cloud.google.com/blog/topics/threat-intelligence/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor, December 2020. Accessed May 2025.

Rio Kishimoto, Tetsuya Kanda, Yuki Manabe, Katsuro Inoue, Shi Qiu, and Yoshiki Higo. A dataset of software bill of materials for evaluating sbom consumption tools. *arXiv preprint arXiv:2504.06880*, 2025.

Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007. doi: 10.1109/MSP.2007.48. Accessed May 2025.

Na Meng, Stefan Nagy, Daphne Yao, Wenjie Zhuang, and Gustavo Arango Argoty. Secure coding practices in java: Challenges and vulnerabilities. *arXiv preprint arXiv:1709.09970*, 2017. URL https://arxiv.org/abs/1709.09970.

NIST. Cybersecurity supply chain risk management practices for systems and organizations. *NIST Special Publication 800-161*, 2015. URL https://csrc.nist.gov/pubs/sp/800/161/r1/upd1/final.

Oracle. Java cryptography architecture (jca) reference guide for java se 17. Technical report, Oracle, 2023. Accessed May 2025.

Oracle. Trail: Creating a GUI With Swing. https://docs.oracle.com/javase/tutorial/uiswing/index.html, 2024. Accessed May 2025.

Oracle. Java Swing API Specification. https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/javax/swing/package-summary.html, 2025. Accessed May 2025.

Mathias Payer. A foundation for secure execution of untrusted programs. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 18–32. IEEE, 2012. doi: 10.1109/SP.2012.11.

C.L. Smith. Understanding concepts in the defence in depth strategy. In *Proceedings of the IEEE 37th Annual International Carnahan Conference on Security Technology*, pages 8–16. IEEE, October 2003. doi: 10.1109/CCST.2003.1297509.

SolarWinds. SolarWinds Security Advisory: SUNBURST Backdoor. https://www.solarwinds.com/securityadvisory, December 2020. Accessed May 2025.

Andreas Somov. SnakeYAML Documentation. https://javadoc.io/doc/org.yaml/snakeyaml/latest/index.html, 2025. Accessed May 2025.

The White House. Executive Order on Improving the Nation's Cybersecurity. https://www.gsa.gov/technology/government-it-initiatives/cybersecurity/executive-order-14028, May 2021. Accessed May 2025.

Mary R. Thompson, Abdelilah Essiari, and Srilekha Mudumbai. Certificate-based authorization policy in a pki environment. *ACM Transactions on Information and System Security (TISSEC)*, 6(4):566–588, 2003. doi: 10.1145/950191.950196.

VirusTotal. YARA Documentation. https://yara.readthedocs.io/en/stable/, 2025. Accessed May 2025.

Shangzhi Xu, Jialiang Dong, Weiting Cai, Juanru Li, Arash Shaghaghi, Nan Sun, and Siqi Ma. Enhancing security in third-party library reuse – comprehensive detection of 1-day vulnerability through code patch analysis. *arXiv preprint arXiv:2411.19648*, 2024. URL https://arxiv.org/abs/2411.19648.