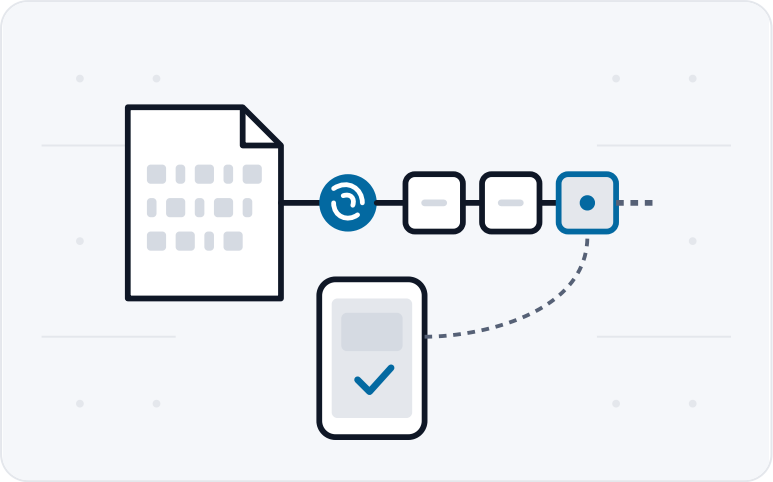
A-SIT Technology

E-Government Innovation Center

24 June 2025 Concept · Transparency · Digital Identity · 32 min read

Wallet Binary Transparency

Edona Fasllija



Authors Edona Fasllija

Source https://technology2.a-sit.at/en/publications/wallet-binary-transparency/

Key findings

- A public, append-only log records the hash, version and metadata of every wallet release, complementing the developers' signature.
- Independent rebuilders confirm reproducibility when their hash matches the log entry, and report deviations as a Mismatched Build Alert.
- At installation, a verification module checks the signature, Signed Entry Timestamp and inclusion proof, and blocks the installation or warns on failure.
- The design comprises five phases, from submission to the log through to continuous monitoring by monitors and auditors.
- For up to 400 million users, the Rekor-based log needs to scale through sharding, replication and static tile APIs.

Binary transparency is a security practice that ensures software binaries are verifiable and have not been tampered with. It aims to increase trust in software by providing mechanisms for third parties, like users, security researchers, or auditors, to verify the integrity and authenticity of a software binary. This approach makes it harder for attackers to distribute malicious code under the guise of a legitimate application.

Binary transparency is especially useful in maintaining trust in critical software systems, such Digital Identity Wallets, where ensuring the integrity of code is paramount for security. It helps mitigate risks in the software supply chain, making it harder for attackers to slip malicious updates or alter software unnoticed. When applying binary transparency to digital identity wallets, a public, append-only log stores cryptographic hashes of the binaries and metadata like version numbers, release dates, and publisher information. Logs are publicly auditable and ensure historical accountability: any modifications to a binary can be detected by comparing it to the hash recorded in the log. Users can verify that updates come from the legitimate developer and match a specific, public version recorded in the transparency log.

1. Introduction

Mobile app distribution systems have long relied on the foundational assumption that developers maintain exclusive control over their signing keys. These cryptographic keys are used to sign application binaries and offer a form of authenticity verification that enables app stores and users to trust that the software originates from the declared source. While this model is widely used, it carries inherent and significant vulnerabilities. Signing keys, once compromised through phishing attacks, malware infections, inadequate key management, or insider leaks, can be used to produce and distribute malicious software that appears indistinguishable from legitimate releases. Insider threats pose an additional danger, as any authorized developer with access to the signing infrastructure can intentionally release unauthorized or backdoored software. This risk is heightened in scenarios where distribution platforms assume responsibility for key management, shifting the trust model toward centralized infrastructure. For example, platforms like Google Play now require developers to upload their signing keys to the platform's systems. While this streamlines app signing and update processes, it also introduces a trust dependency on the platform's security practices and operational integrity.

To mitigate these risks, the software security community has developed the concept of Binary Transparency (BT). At its core, binary transparency introduces a verifiable record-keeping layer to software distribution, creating an append-only, cryptographically secured log of every released binary. Unlike traditional systems that depend solely on the security of signing keys, binary transparency allows tampered or unauthorized software to be detected through public audit logs. Every signing event and associated metadata, such as binary hashes, version numbers, and timestamps, is immutably recorded in a transparency log. This log, based on Merkle tree structures, ensures that past entries cannot be altered or removed without detection and forms a consistent and auditable history of software releases.

By publishing every version of a distributed binary to this log, developers can make it possible for external auditors, independent rebuilders, and even end users to verify the integrity and provenance of their software. This makes targeted attacks, where malicious versions are served only to specific users, much more difficult to carry out undetected. It also reduces the damage of a signing key compromise, as

malicious builds that are not properly logged or cannot be reproduced from source are quickly flagged as suspicious. Moreover, the log itself can be monitored continuously by third parties, who can alert the public if they observe inconsistencies, unexpected releases, or attempts to rewrite the historical record of published binaries.

This project explores how binary transparency can be integrated into the software development and distribution lifecycle of the European Digital Identity Wallet (EUDI Wallet). Developed under the EU's eIDAS 2.0 framework, the EUDI Wallet is intended to allow every EU citizen to securely store and present official digital credentials, such as national ID cards, driver's licenses, health certificates, and academic records. Given the sensitivity and legal significance of the data managed by the wallet, ensuring that its software is authentic and verifiably secure is essential.

Our work focuses on applying binary transparency to make every wallet release publicly auditable. In our design, developers sign binaries and submit them to a tamper-evident log, creating a permanent record of each release. Independent rebuilders can reproduce these binaries from source and publish reproducibility confirmations, ensuring that the released binary corresponds to the publicly reviewed code. Client devices or installer modules can then query the transparency log at install time to verify the binary's presence, authenticity, and integrity. If the binary is missing from the log, does not match reproducible builds, or shows signs of anomalous signing activity, the installation can be blocked or flagged for review.

By integrating transparency mechanisms, our project replaces a trust model based solely on central authorities with one that allows anyone to independently verify software authenticity. This enables regulators, developers, and end users to independently confirm that each release is genuine and unmodified. It also enables real-time monitoring and post-incident analysis, which are essential in a complex, high-assurance system like the EUDI Wallet. In this way, the project aligns with the objectives of eIDAS 2.0 by promoting a digital identity framework that is not only secure and interoperable but also transparent and verifiable by design.

2. Background

2.1. Software Supply Chain Attacks

Signing Key Compromises

Developer-controlled signing is common, but it comes with risks that can be hard to detect and even harder to recover from. Signing keys, which serve as the core trust anchors in this system, are sensitive assets. If compromised, they can be used to generate malicious binaries that appear fully legitimate to end users, update systems, and app stores.

There are several well-documented real-world examples illustrating how signing key compromise can lead to large-scale attacks[1], [2]. Not only are such attacks impactful, but they are also increasing in frequency and sophistication. According to the 2025 report by ReversingLabs [3], software supply chain attacks have continued to rise, with incidents increasing more than 700% over the past four years. The report highlights that attackers are increasingly targeting trusted development infrastructures to inject malicious code into otherwise legitimate software. Similarly, the European Union Agency for Cybersecurity (ENISA) identified

supply chain attacks as one of the top seven cybersecurity threats in its Threat Landscape 2024 report [4], alongside ransomware, malware, and social engineering. The report emphasizes that trust in signing keys alone is no longer sufficient to ensure software integrity.

Build Pipeline Threats

While signing key compromise is a serious risk, a growing class of attacks targets the build pipeline itself. These attacks exploit the gap between the reviewed source code and the final compiled binary. Even when developers securely manage their signing keys and release process, the build environment can be compromised to inject malware or unauthorized behavior into production software. Rather than compromising source code or stealing signing credentials, adversaries now focus on the software development and delivery process, manipulating what gets built and distributed. These build-time attacks introduce malicious behavior during compilation or packaging, often without leaving any trace in the source code or version control systems.

Moreover, modern build systems often rely on third-party dependencies fetched from public repositories (e.g., Maven Central, npm, PyPI). If these dependencies are not cryptographically pinned and verified, attackers can exploit "dependency confusion" [5], [6] by uploading malicious packages to public registries with the same names as internal dependencies. The build system may mistakenly resolve these public packages, leading to silent code injection.

Another category of build-time risk involves tampering with the tools used to compile source code. If a compiler or build tool is compromised, it can inject malicious code into the final binary, regardless of whether the source appears clean. This form of attack was described in Ken Thompson's seminal 1984 lecture, "Reflections on Trusting Trust" [7], where he demonstrated that a backdoored compiler could propagate itself across generations of software undetected. Although originally presented as a thought experiment, such toolchain attacks are now considered plausible, especially in closed-source ecosystems or unmonitored CI/CD environments.

2.2. Reproducible Builds

Reproducible builds have emerged as a fundamental strategy for improving software supply chain security. They ensure that, given the same source code, configuration, and environment, independent systems can produce identical binaries. While this might seem like a low-level technical detail, it has significant implications: it allows anyone to verify that a binary truly corresponds to its source, helping to detect tampering, build-time attacks, and other forms of unauthorized modification.

Historically, software compilation has been treated as a deterministic process, but in practice, most builds include a range of non-deterministic elements: timestamps, randomized file ordering, differing toolchain versions, or external dependency resolution. These inconsistencies mean that even when developers publish their source code, users or auditors cannot verify whether the binary they receive was faithfully built from that source. This creates a blind spot between "what was reviewed" and "what was actually shipped." As shown in the previous section, this gap has been exploited in real-world attacks. Malware authors have targeted the build process itself to inject backdoors, either by compromising build servers, tampering with compilers, or manipulating dependency resolution. Reproducible builds aim to close that gap. By enforcing build determinism, they allow independent parties such as auditors, researchers, or watchdog organizations

to compile the software themselves and verify that the resulting binary is identical to the one distributed to users. This cryptographic match proves that the build process was not tampered with and that the binary reflects the reviewed source code.

Across the open-source ecosystem, reproducible builds have already gained significant traction. Projects such as Debian, Tor, and the Bitcoin Core wallet have adopted reproducible build infrastructure to ensure that what users run is exactly what was intended by the developers. These practices are now being formalized in initiatives like the Reproducible Builds Project [8] and are being promoted by standards bodies and government agencies.

While this concept has gained significant traction in desktop and server software, applying these principles to mobile apps is considerably more complex. Mobile platforms come with distinct technical constraints, fragmented ecosystems, and platform-specific behaviors that make achieving reproducibility significantly more difficult.

Challenges in Reproducing Mobile App Builds

Reproducing mobile app builds poses distinct challenges due to the complexity and variability of mobile development environments. Toolchains such as Android Studio or Xcode evolve rapidly, and even minor differences in compiler versions or build configurations can result in divergent outputs. Mobile builds typically embed non-deterministic metadata, like timestamps, version codes, and build identifiers, into the final binary, making identical rebuilds impossible without sanitization.

Packaging tools introduce further variation through ZIP compression and file ordering, while dependency management systems (e.g., Maven, CocoaPods) may fetch unpinned or inconsistent libraries, leading to silent code changes. Android builds also use obfuscation tools like ProGuard and R8, which alter binaries based on the build environment. Together, these factors make reproducibility in mobile apps a technically demanding objective.

Despite the challenges, the Android ecosystem has made notable progress toward reproducible builds. The F-Droid platform enforces rebuild verification for open-source apps, supported by tools like Reproducible Builds for Android (RBA), which use Dockerized Gradle builds and strict dependency pinning. Gradle itself now supports reproducibility through configuration flags that control file timestamps and archive ordering, and developers can further stabilize builds by fixing locales, time zones, and build metadata. Reproducibility on iOS presents more challenges compared to Android, largely due to the platform's reliance on Apple's proprietary Xcode toolchain and tightly managed signing process. Additional factors, such as bitcode compilation and App Store re-signing, can introduce variability that complicates efforts to verify that a distributed binary exactly matches its original source.

2.3. Binary Transparency

Binary Transparency (BT) strengthens software supply chain security by making software releases publicly auditable [9]. At its core, BT relies on an append-only cryptographic log, often built on a Merkle tree structure, to record each published software binary alongside metadata such as version numbers, timestamps, and developer identity information. These logs are tamper-evident and publicly accessible and allow third parties to independently verify that a given binary matches what was officially released and reviewed.

When a developer produces and signs a new binary, they also submit a cryptographic hash of the binary and accompanying metadata to a transparency log. The log issues a Signed Entry Timestamp (SET), proving that the submission occurred and establishing the binary's place in the immutable log history. This record can then be distributed along with the binary itself, so that users, auditors, or client-side verifiers can confirm the binary's inclusion and authenticity. Independent monitors regularly check the log for anomalies, such as unauthorized entries, inconsistencies in release order, or attempts to suppress or roll back a version. Auditors may also rebuild binaries from source and compare them against those logged in the system, further ensuring that what was built matches what was signed and released.

Importantly, BT does not rely on trusting a single entity. Because logs are publicly verifiable, and anyone can run a monitor or perform audits, BT decentralizes oversight. It transforms software distribution from a system based on blind trust in signing keys, build servers, or app stores, into one grounded in cryptographic proof and transparency.

By extending traditional signing practices with publicly verifiable logs, BT introduces a new layer of defense against a range of software supply chain attacks. It ensures that even if a build system is compromised or a signing key is abused, the act of tampering cannot be hidden. Every released binary has a traceable, public history; any irregularity can be identified and investigated. BT also discourages coercion or censorship in centralized ecosystems. If an app store silently distributes a malicious or modified binary, the lack of a matching log entry or a mismatch between the log and the distributed version will expose the discrepancy.

Sigstore

Sigstore [10]is an open-source project that provides tools and infrastructure to make software signing and verification more secure, transparent, and accessible. It allows developers to sign software artifacts, such as binaries, container images, and source code, without managing long-term private keys. Instead, Sigstore uses short-lived certificates tied to trusted identity providers (like GitHub or Google) and logs all signatures in a public, tamper-evident transparency log called Rekor.

By combining keyless signing, identity-based authentication, and public logging, Sigstore helps ensure that software artifacts can be traced back to their origin and verified by anyone, making it harder for attackers to introduce untrusted or malicious code into the supply chain. It is widely adopted in the open-source community and forms a strong foundation for implementing binary transparency systems.

2.4. The EUDI Wallet

The European Digital Identity (EUDI) Wallet is a mobile application developed under the EU's eIDAS 2.0 regulation that aims to provide citizens with a secure, standardized way to store and present digital identity credentials (such as national ID cards, driver's licenses, and academic or health records). The wallet is intended to work across all EU Member States and brings together a range of actors (depicted in Figure 1):

Wallet Providers are public authorities or certified private vendors responsible for developing and maintaining the wallet applications. They ensure the wallet complies with eIDAS 2.0 technical and regulatory requirements and handle the software lifecycle, including updates and distribution. Credential Issuers are entities that issue verifiable credentials to wallet users. These credentials must be securely stored and presented through the wallet. Verifiers are the organizations (Relying Parties or Service Providers) that request and validate credentials presented via the EUDI Wallet. End Users are the citizens

who use the wallet to manage their digital identity and present credentials in both online and offline scenarios, such as opening a bank account, enrolling in school, or accessing cross-border public services. Further, Member State Authorities are national governments or designated institutions that oversee wallet issuance, certify wallet providers, and enforce compliance with EU regulations.

The wallet is intended to serve as a trusted interface for managing personal identity data across borders, institutions, and use cases. A compromise of the build process could allow malicious actors to insert backdoors into wallet binaries that enable credential theft, surveillance, or unauthorized data access without any visible change to the source code.

Given the critical nature of these interactions and the fact that wallet software may be developed by multiple providers across jurisdictions, traditional mechanisms like code signing and version control alone are insufficient to ensure trust, integrity, and transparency across the ecosystem.

3. Goals

This project aims to establish a verifiable distribution process for the EUDI Wallet by focusing on four key goals:

(i) Auditability of Software Releases: Wallet Binary Transparency (WBT) ensures that every wallet binary developed by either a national authority or a certified vendor leaves a verifiable record by logging each signing event in a public, append-only transparency log. This creates a cryptographic audit trail that allows anyone, including developers, auditors, regulators, and end users, to verify the authenticity of a given release and detect any unauthorized or tampered versions.

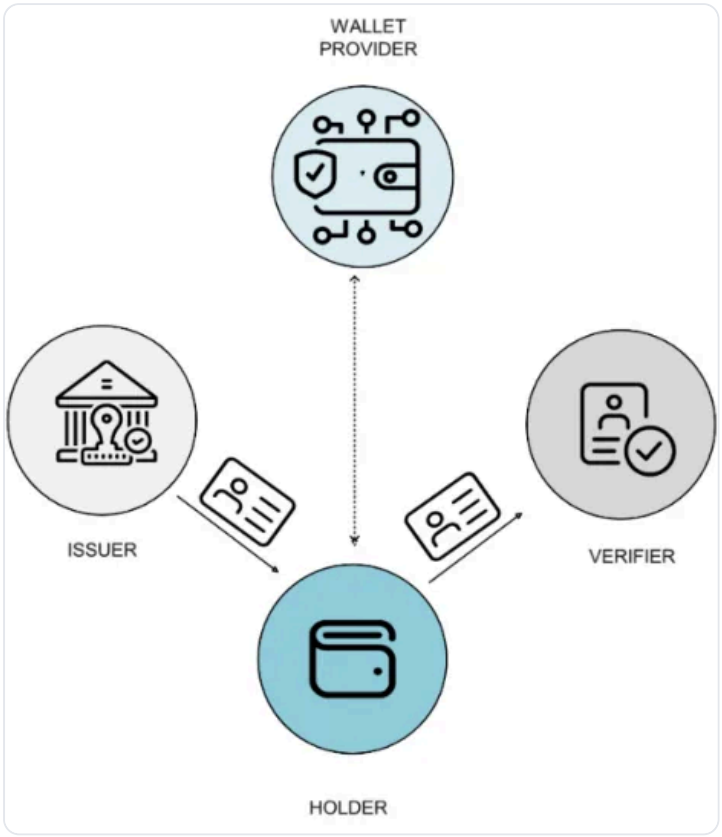


Figure 1: The Wallet Ecosystem

(ii) Integration with Automated Build Workflows: In WBT, transparency logging is embedded directly into the CI/CD pipeline. By automating the signing of binaries and submission to the transparency log, WBT aims to achieve a scalable and consistent process across releases. Each build is paired with signed metadata and a unique timestamped log entry, ensuring traceability throughout the software lifecycle.

(iii) End-User Protection through Verification at Installation: To safeguard users at the point of installation, a verification module checks whether the binary has been properly logged. It retrieves inclusion proofs from the transparency log and validates the binary's integrity and signature. If inconsistencies are found, such as a missing or altered log entry, the installation is blocked or flagged for review.

(iv) Alignment with Regulatory Requirements: By combining public transparency logs, reproducible build attestations, and independent verification, the system provides a strong foundation for demonstrating compliance with eIDAS 2.0 requirements. It not only secures the software supply chain but also supports regulatory oversight through verifiable evidence of software integrity and authenticity.

Together, these goals contribute to a distribution model that is not only secure by design, but also independently auditable and compliant with the high-assurance standards required for digital identity systems in the EU.

4. Wallet Binary Transparency

4.1. Ecosystem Actors

This section focuses on designing a binary transparency system for the EUDI Wallet. We start by defining the roles of the various actors of the wallet distribution ecosystem. Each actor performs specific functions that contribute to verifiability and transparency at different stages of the software lifecycle. The system is designed to distribute trust and responsibilities among these independent participants.

Wallet Providers: Once a new version is ready for release, they compile the source into a binary and digitally sign it using cryptographic keys. Importantly, developers are also required to submit metadata about the binary, including the hash, version, and signature, to the binary transparency log. Additionally, developers may also publish reproducible build instructions, enabling others to independently verify that the distributed binary corresponds to the intended source code. In addition, developers monitor the transparency log to detect unauthorized entries that may indicate key misuse or impersonation (or alternatively delegate this task to a Monitor).

App Distribution Platforms (App Stores) serve as the primary channels through which wallet applications are delivered to end users. These stores make the wallet software accessible while enforcing platformspecific policies on signing, packaging, and update delivery. In our system, app stores continue to serve as the delivery channel, but now the integrity of what they distribute can be audited independently. Monitors or clients can compare the app store's binary with the public log and verify its authenticity The Log operator maintains the core infrastructure of the system: the append-only, cryptographically verifiable log. This log records each submitted wallet binary and issues a Signed Entry Timestamp (SET) to confirm inclusion. The log is implemented using Merkle trees, ensuring that all entries are immutable and tamper-evident. The operator must guarantee availability, consistency, and correct operation of the log, serving inclusion and consistency proofs to clients and monitors on request. The log operators' actions are continuously audited by third parties.

Independent rebuilders obtain the wallet's source code and build configuration from public repositories and attempt to reproduce the distributed binary. If the resulting hash matches the one recorded in the transparency log, they can publish a "reproducibility confirmation". These confirmations serve as strong signals that the build process was honest and deterministic. If discrepancies are found, rebuilders raise alerts to signal potential tampering or non-reproducibility.

End users, or Holders, interact with the system at the final stage of the distribution chain. When a new wallet version is installed, a client verification module checks the signature, retrieves the corresponding entry from the transparency log, and validates the inclusion proof. In more advanced configurations, it can also verify whether the release has been independently confirmed by rebuilders. If the checks fail, the device may warn the user or block installation. This design empowers end users to verify software integrity.

Monitors are independent entities responsible for verifying the integrity of individual entries in the transparency log. Their primary role is to detect signs of malfeasance, such as missing, altered, or unauthorized entries, by inspecting specific submissions to the log. They regularly fetch entries, validate their inclusion proofs and Signed Entry Timestamps (SETs), and ensure that each logged artifact matches what was distributed or expected. If irregularities are found, such as a signed binary that was not properly logged or a mismatch between a log entry and a released binary, monitors can raise alerts, notify developers, or trigger regulatory action.

Auditors oversee the log provider to ensure it appends information properly and maintains a consistent global view. Auditors play a key role in detecting any dishonest behavior by the log provider. Any auditor can identify and irrefutably demonstrate if a log provider has cheated. Since anyone in the public can act as an auditor without needing special access, log providers must operate under the assumption that their published data is continuously monitored and audited. This setup, coupled with a secure consistency protocol for sharing log fingerprints, enforces honest behavior by the log provider.

Regulatory bodies, such as the European Commission or national cybersecurity agencies, oversee compliance with the eIDAS 2.0 framework. They can mandate adherence to binary transparency practices as part of software certification and assurance processes. Authorities may also operate monitors, verify reproducibility, enforce reporting requirements, or investigate incidents related to compromised or noncompliant wallet software. Their involvement ensures that legal and procedural safeguards complement the system's technical guarantees.

Together, these actors form a decentralized but interconnected ecosystem that supports secure, transparent, and auditable wallet software distribution as depicted in Figure 2. Each actor independently contributes to a collective security posture, ensuring that no single point of trust can undermine the system, and that the integrity of digital identity in the EU remains verifiable by all.

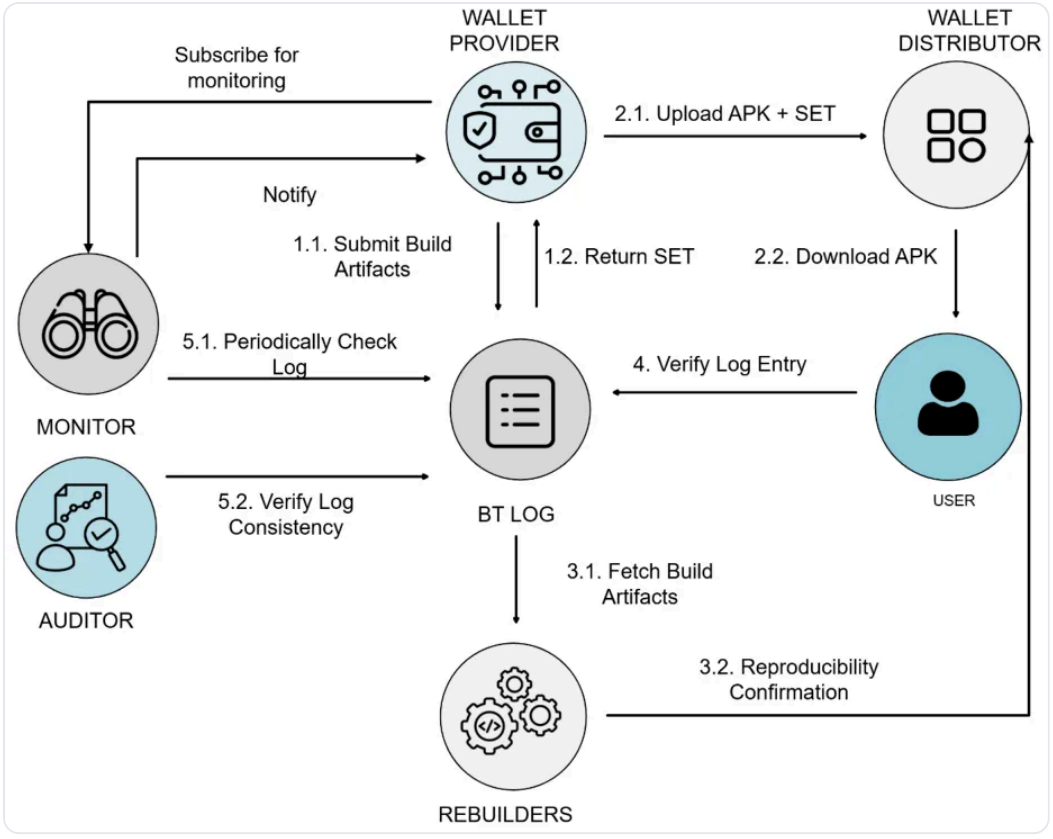


Figure 2: Wallet Binary Transparency Overview

4.2. Designing Wallet Binary Transparency

The Wallet binary transparency system, inspired by Mobile App Distribution Transparency (MADT) [11], is organized into a structured series of interlocking phases:

Phase 1: Wallet Provider Release Submission

In this initial phase, wallet developers prepare a new release by compiling the wallet application from source. Before distribution, the developer signs the resulting binary using an approved cryptographic key, ensuring that the origin of the build can be authenticated. Crucially, the signed binary is not distributed immediately. Instead, the developer submits a Build Attestation to a publicly accessible binary transparency log. This attestation includes the binary hash, version metadata, signature, and a unique identifier.

Upon submission, the transparency log appends the entry to its Merkle-tree data structure and issues a Signed Entry Timestamp (SET). This SET serves as a tamper-evident receipt, proving that the release has been recorded in the log. Protocol 1 summarizes this phase.

1. Wallet Developer (WD) compiles the wallet application from source using a reproducible build environment to produce a binary artifact (e.g., wallet.apk).
2. WD signs the binary using an approved cryptographic key (or Sigstore certificate), producing a detached signature.
3. WD generates a build attestation containing a cryptographic hash of the binary, version metadata (e.g., release version, build timestamp), signature, and key identity.
4. WD submits the build attestation to the Log Operator via an authenticated request.

5. WBT appends the attestation as a new leaf in its Merkle tree and returns a Signed Entry Timestamp (SET) to WD. The SET includes the time of inclusion, the Merkle tree root hash, and a cryptographic signature from the log operator.
6. WD packages the binary, its signature, and the SET for release. The binary is not made available to end users until the SET is received and included in the release metadata.

Protocol 1: Developer Release Submission Protocol This protocol ensures that every release is cryptographically signed, publicly recorded, and verifiable before it is distributed.

Phase 2: Wallet Distribution and Publication

Following successful log submission, the signed binary and its associated SET are published on official distribution channels—such as national app stores, government portals, or approved third-party platforms. Because the build has already been logged, this phase is strictly for dissemination, not verification.

To support transparency, developers may also publish source code, build scripts, dependency manifests, and environment specifications, enabling reproducibility. Distribution metadata should include a reference to the log entry (e.g., log index or hash) so that downstream verifiers can easily locate the corresponding record.

1. Wallet Developer publishes the release bundle through approved App Distribution Platforms.
2. To support transparency and reproducible builds, WD may also publish:
    - Source code (or a reference commit in public version control)
    - Build scripts (e.g., Gradle, Dockerfile)
    - Dependency lock files (e.g., dependencies.lock)
    - Environment specifications (e.g., toolchain versions, OS configs)
3. WD include in the published metadata a clear reference to the transparency log entry, allowing end users, auditors, or automated systems to retrieve and verify the corresponding log record from the Wallet Binary Transparency Log.

Protocol 2: Wallet Distribution and Publication Protocol

Phase 3: Independent Rebuilding and Confirmation

In this phase, independent rebuilders download the source code and attempt to reproduce the distributed binary using the published build environment. This step verifies that the binary was compiled from the expected source without hidden modifications.

If the rebuilt binary matches the original (hash-equivalent), the rebuilder publishes a Reproducibility Confirmation. This confirmation includes the build hash, the original log entry reference, and a cryptographic signature from the rebuilder. Multiple independent confirmations serve as strong evidence that the binary was correctly built and not tampered with during the compilation process.

If the rebuild fails to produce a matching hash, the rebuilder can instead submit a Mismatched Build Alert, which is also logged and may trigger further investigation or halt deployment by downstream systems.

1. Independent Rebuilder (IR) retrieves the necessary materials:
    - Source code at the specified commit
    - Build scripts, dependency manifests, and toolchain configurations
    - Reference to the original log entry (UUID or log index)
    - Official binary (for hash comparison)
2. IR builds the binary using the provided environment in a clean, isolated context (e.g., Docker, Nix). The result is a candidate binary for verification.
3. IR computes the hash of the rebuilt binary and compares it with the hash recorded in the original Rekor entry.
    - If hashes match → the build is reproducible
    - If hashes differ → a mismatch is flagged

4.a. IR generates and signs a Reproducibility Confirmation that includes:

- Hash of the rebuilt binary
- Reference to the original log entry
- Timestamp and rebuilder identity
- A digital signature binding these elements

4.b. If the rebuilt binary does not match, IR submits a Mismatched Build Alert to Monitors or Regulatory Entities.

Protocol 3: Independent Rebuilding and Confirmation Protocol

Phase 4: Installation-Time Client Verification

When a user installs or updates the EUDI Wallet on their device, the embedded verification module is activated. This component performs a sequence of automated checks before proceeding with installation:

Only if all checks pass does the system proceed with installation. If any verification step fails—due to a missing log entry, invalid SET, or lack of independent confirmations- the user is alerted, and installation is either paused or blocked entirely, depending on policy settings.

1. Upon installation or update of the EUDI Wallet, the embedded Verification Module (VM) is automatically activated on the User Device (UD).

2. VM checks the digital signature on the binary using a trusted list of developer public keys or authorized signing identities. If the signature is invalid or the signer is untrusted, the process fails immediately.
3. VM retrieves the Signed Entry Timestamp (SET) or UUID associated with the binary and queries the Wallet Binary Transparency Log
    - It fetches the Merkle inclusion proof for the given SET
    - It verifies that the binary hash and metadata match the recorded log entry
    - It confirms the SET's signature using WBT's public key.
4. Depending on policy, VM optionally checks the log for reproducibility confirmations linked to the same log UUID.

5.a. If all checks pass, the installation proceeds.

5.b. If any check fails (e.g., missing SET, invalid signature, or insufficient confirmations), depending on the client policy:

- Verification Module halts or blocks the installation
- A warning or alert is displayed to the user

Protocol 4: Installation-Time Client Verification Protocol

Phase 5: Continuous Monitoring and Auditing

The final phase is ongoing and applies across the lifecycle of all wallet releases. Monitors continuously observe the transparency log to ensure that it behaves correctly: appending new entries in order, serving valid consistency proofs, and maintaining a singular view of the log for all clients.

If a monitor detects irregularities, it may trigger automated alerts, notify regulatory authorities, publish public warnings, or initiate software rollback and revocation procedures. Monitors thus serve as the system's early warning mechanism and reinforce collective vigilance.

1. Monitors continuously query the WBT on a regular schedule to fetch newly appended entries.
2. Monitors compare observed log entries against expected release data and policy baselines:
    - Confirms that each release is signed by an authorized developer
    - Flags unexpected signing identities or key reuse across unrelated releases
    - Log responses match known, trusted WBT public key
3. Auditors continuously query the WBT to fetch updated Log Checkpoints and consistency proofs between checkpoints. Auditors verify that:
    - Each checkpoint is valid and consistent with previous states

- No forked views or equivocation (i.e., conflicting checkpoints) are presented to different clients

4. If needed, Monitors and Auditors trigger response actions from Regulatory Bodies.

Protocol 5: Continuous Monitoring and Auditing Protocol

5. Implementation Considerations

The following section outlines each of the five system phases, describing how they can be concretely realized using the EUDI Wallet Reference Implementation for Android and Sigstore's open-source infrastructure, including Cosign and Rekor.

Phase 1: Developer Signing and Log Submission

The first phase of the system begins during the CI/CD build pipeline. Once wallet developers produce a new version of the application, in our implementation, an Android APK, they must sign the binary before it is released. This step can either be done using a long-lived, manually managed private key or using Sigstore's Cosign tool for keyless signing. In the second model, Cosign authenticates the developer through an OpenID Connect (OIDC) provider (e.g., GitHub or Google) and obtains a short-lived certificate from Fulcio, which is then used to sign the binary.

Once the binary is signed, Cosign automatically submits a signed metadata bundle to Rekor, Sigstore's transparency log. This bundle includes the binary's digest, the signature, the developer's identity certificate, and other metadata. Rekor assigns a Signed Entry Timestamp (SET) to the log entry, essentially a cryptographic receipt that confirms the binary was immutably recorded in an append-only Merkle tree. The SET, or the unique identifier of the entry (UUID), is preserved and distributed alongside the binary.

This process can be fully automated within the CI pipeline of the EUDI Wallet Reference Implementation. For example, in a GitHub Actions workflow, developers can include Cosign commands to sign the built wallet binary and submit the resulting signature to Rekor, embedding the SET into a release manifest to accompany the published release.

Phase 2: Distribution and Publication

The signed binary, detached signature, and associated transparency-related metadata (including the Rekor UUID or SET) are published via GitHub Releases, national eIDAS portals, or trusted EU-hosted endpoints. All artifacts must reference the associated Rekor log entry. To support reproducibility verification, source code, build instructions, and pinned dependency manifests are also published. These materials are stored in the same repository or linked from the release metadata.

Phase 3: Independent Rebuilding and Confirmation

Auditors and rebuilders clone the source code at the specified commit, recreate the environment using the provided containerization tooling, and rebuild the binary. If the resulting hash matches the one logged in Rekor (using tools such as diffoscope), the rebuilder signs a Reproducibility Confirmation, referencing the original log entry.

Phase 4: Installation-Time Client Verification

The client verification module:

- Validates the Cosign signature using a trusted list of developer public keys or identities (which are included as root of trust for the ecosystem).
- Queries the Rekor instance to verify inclusion of the binary (by UUID or SET) and obtain the inclusion proof.
- Optionally, checks for reproducibility confirmations submitted by trusted rebuilders.

Verification logic is embedded into the mobile codebase. Verification failures (missing log entry, invalid signature, or lack of rebuild confirmations) trigger enforcement actions, such as blocking installation or showing a warning. In Android, verification can be triggered post-download and pre-installation.

Phase 5: Continuous Monitoring

Independent monitors poll the Rekor log regularly via rekor-monitor to fetch new entries and track which identities are signing releases and detect unexpected key reuse.

Build Reproducibility

To evaluate the build reproducibility of the EUDI Wallet reference implementation for Android, we adopt a structured and automated methodology that ensures consistency across builds and surfaces any deviations. The process is organized into four key stages:

We begin by establishing a controlled build environment using Docker. This allows us to encapsulate all build dependencies, including the Android SDK, Gradle, and Java toolchain, within a container, ensuring isolation from host system variations. To further reduce variability, all dependencies are explicitly versionpinned, eliminating differences caused by changing library or plugin versions.

Next, we normalize build inputs to remove sources of non-determinism. We configure the environment with SOURCE_DATE_EPOCH, setting a fixed build timestamp to neutralize time-based variability. We also adjust build tools to prevent the inclusion of volatile data, such as absolute file paths or system-specific metadata, which can otherwise influence the binary output.

The build process itself is automated and repeatable, orchestrated through a Docker-based script that compiles the Android wallet from source in a clean state. This automation ensures that each invocation of the build process follows the exact same sequence of steps and configuration.

Finally, we conduct iterative testing and comparison. The build is executed multiple times under the same controlled conditions, and the resulting APK files are compared using diffoscope, a specialized tool for analyzing binary differences. This step allows us to detect even subtle discrepancies between builds, validating whether the output is bit-for-bit identical and therefore truly reproducible.

Rekor Log Entries

Rekor log entries contain structured metadata that captures key information about a signed software artifact or attestation. The fields in a Rekor log entry are defined by its entry type (e.g., hashedrekord, intoto, rekord, etc.), but all entries share a common structure that includes cryptographic proofs, timestamps, and signing material. For the build reproducibility checks, we focused on build attestations.

Build attestations in Rekor are structured metadata entries that describe how a software artifact was produced. They include cryptographic evidence of the build process and are typically submitted alongside a signed artifact to provide a verifiable link between the source code, the build environment, and the resulting binary. Table 1 gives an overview of the main fields in a build attestation in Rekor:

The verification module on the client device later fetches the build attestation, checks the hash of the binary, validates the signatures, and confirms that the build process matches what is declared in the statement.

Transparency Log Scalability Considerations

To support EU-wide distribution transparency of the EUDI Wallet, a WBT log based on Sigstore's Rekor must be able to scale to handle verification requests from up to 400 million users. The log is queried during app installation, updates, or automated auditing, resulting in significant query volumes and potential traffic spikes on release days. A naïve single-instance deployment of Rekor would be insufficient at this scale and would introduce performance bottlenecks and availability risks.

To address this, Rekor supports sharding [12], allowing logs to be partitioned into multiple independent Merkle trees. Once sharded, Rekor automatically routes entries and queries to the correct tree.

This enables horizontal scalability by distributing load across multiple Merkle tree-backed logs, each independently append-only and queryable. In practice, each shard could correspond to a specific member state or wallet provider, reducing contention and improving response times. Rekor also supports replicated deployments where multiple server instances share the same data backend to support high availability and load balancing across geographic regions.

For high-throughput read access, the log can be backed by static tile APIs [13], which is a technique recently used in certificate transparency logs to enable efficient CDN caching and reduce pressure on Rekor's live API endpoints. Tools like Trillian-Tessera support tile-based Merkle tree publishing, which enables clients to fetch Merkle proofs and checkpoints via cached, immutable blobs hosted on public CDNs, which significantly lowers latency and infrastructure load.

Field	Description
Artifact Digest	The SHA-256 (or similar) hash of the resulting binary, linking the attestation to a specific release.
Build Type	The format or purpose of the attestation, e.g., SLSA provenance v0.2 or in-toto statement.
Builder Identity	The tool or system that performed the build, such as GitHub Actions, a Docker container, or a reproducible CI pipeline.
Source URI and Commit	The Git repository URL and commit hash used as the source for the build.
Build Parameters	Environment variables, tool versions, and other inputs that influence the reproducibility of the build.
Timestamp	The date and time when the build was executed.
Signer Identity	The entity signing the attestation, typically via Cosign, GPG, or another verified identity provider.

Table 1: Build Attestation Fields in Rekor

6. Conclusions:

This project explored how binary transparency can be integrated into the software development and distribution lifecycle of the European Digital Identity Wallet (EUDI Wallet) to strengthen trust and accountability. By extending traditional signing practices with independently verifiable logs and enabling third-party rebuilds, the system protects against a wide range of software supply chain threats, including compromised build pipelines, signing key misuse, and targeted distribution attacks.

Wallet Binary Transparency introduces verifiable audit trails, independent rebuild confirmations, clientside verification, and continuous monitoring. This system shifts trust from opaque processes and centralized authorities to open, verifiable mechanisms. Ultimately, this approach lays the foundation for a digital identity infrastructure that is not only secure and interoperable but also transparent and publicly accountable.

References

[1] “Operation ShadowHammer | Securelist.” Accessed: May 23, 2025. [Online]. Available: https://securelist.com/operation-shadowhammer/89992/

[2] “CCleanup: A Vast Number of Machines at Risk.” Accessed: May 23, 2025. [Online]. Available: https://blog.talosintelligence.com/avast-distributes-malware/

[3] “The 2025 Software Supply Chain Security Report.” Accessed: May 23, 2025. [Online]. Available: https://www.reversinglabs.com/sscs-report

[4] “ENISA Threat Landscape 2024 | ENISA.” Accessed: May 23, 2025. [Online]. Available: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024

[5] M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabber’s Knife Collection: A Review of Open Source Software Supply Chain Attacks,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12223 LNCS, pp. 23–43, 2020, doi: 10.1007/978-3-030-52683-2_2.

[6] “Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies | by Alex Birsan | Medium.” Accessed: May 23, 2025. [Online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[7] K. Thompson, “Reflections on Trusting Trust,” *Commun ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984, doi: 10.1145/358198.358210;GROUPTOPIC:TOPIC:ACM-PUBTYPE>MAGAZINE;WGROUP:STRING:ACM.

[8] “Reproducible Builds — a set of software development practices that create an independently-verifiable path from source to binary code.” Accessed: May 27, 2025. [Online]. Available: https://reproducible-builds.org/

[9] “Binary Transparency.” Accessed: May 23, 2025. [Online]. Available: https://binary.transparency.dev/

[10] “Home · Sigstore.” Accessed: May 27, 2025. [Online]. Available: https://www.sigstore.dev/

[11] M. Lins, R. Mayrhofer, M. Roland, and A. R. Beresford, “Mobile App Distribution Transparency (MADT): Design and Evaluation of a System to Mitigate Necessary Trust in Mobile App Distribution Systems,” pp. 185–203, 2024, doi: 10.1007/978-3-031-47748-5_11.

[12] “Sharding - Sigstore.” Accessed: Jun. 23, 2025. [Online]. Available: https://docs.sigstore.dev/logging/sharding/

[13] “Reflections on a Year of Sunlight - Let’s Encrypt.” Accessed: Jun. 23, 2025. [Online]. Available: https://letsencrypt.org/2025/06/11/reflections-on-a-year-of-sunlight/