Software Identification for Cybersecurity: Survey and Recommendations for Regulators[*]

Olivier Barais
Univ. Rennes, Inria, IRISA
Rennes, France
olivier.barais@irisa.fr

Roberto Di Cosmo
Inria, Université Paris Cité
Paris, France
dicosmo@dicosmo.org

Ludovic Mé
Inria, Univ. Rennes, IRISA
Rennes, France
ludovic.me@inria.fr

Stefano Zacchiroli
Institut Polytechnique de Paris
Palaiseau, France
stefano.zacchiroli@telecom-paris.fr

Olivier Zendra
Inria, Univ. Rennes, IRISA
Rennes, France
olivier.zendra@inria.fr

March 28, 2025

1 In quest of an identifier for open-source components

Global momentum around software supply chain security has increased over the last few years. High-profile cybersecurity incidents — together with new regulatory imperatives — underscore the **need for robust, verifiable identification of all software components**, to clearly identify software artifacts whose vulnerabilities are the root cause of potential attacks.

Indeed, in the European Union, the Cyber Resilience Act (CRA) [11] lays out strong requirements for transparency and accountability across digital products. Similarly, recent U.S. Executive Orders [3, 5] reinforce these same principles at US federal level. On May 12, 2021, President Biden signed the *Executive Order on Improving the Nation's Cybersecurity* [3]. Section 4, "Enhancing Software Supply Chain Security," highlights the pressing need for rigorous and predictable mechanisms to ensure that software—particularly "critical software"—resists attack and can be trusted. Critically, it mandates:

> *[...] ensuring and attesting, to the extent practicable, to the integrity and provenance of open source software used within any portion of a product.*

[*]This work was supported by France's Agence Nationale de la Recherche (ANR), program France 2030, reference ANR-22-PTCC-0001. This funding enables Inria to operate the "transfer program at Campus Cyber" (PTCC) for the benefit of the entire French academic community and the ecosystem of cybersecurity players/suppliers/users. The PTCC implements a coherent innovation policy and supports various existing knowledge and technology transfer mechanisms. Support for project leaders ranges from the initial idea through to validation of potential financial backing.

On January 16, 2025, a follow-up *Executive Order on Strengthening and Promoting Innovation in the Nation's Cybersecurity* [5] further clarifies the US government's stance. Acknowledging that open source software (OSS) plays a "critical role in Federal information systems," the order requires agencies to better manage their use of open source software and contribute to OSS cybersecurity. Federal guidance now expects structured security assessments, thorough patch management, and responsible engagement with OSS communities.

This regulatory activity on both sides of the Atlantic indicates a concerted effort to reduce cybersecurity risks by systematically identifying, verifying, and tracking software artifacts, with particular emphasis on open source ones. At the heart of this effort is the recognition that *software identification* — the practice of assigning robust, unique labels to software components — helps ensure trust, visibility, and accountability in both critical and routine operations. Yet, existing naming schemes and repository-based references often prove ephemeral, inexact, or insufficiently secure.

By contrast, **content-based, persistent software identifiers enable unambiguous references that outlive any particular development platform or hosting service**, thus providing a pathway to meet these newly mandated obligations. We contend that **SWHIDs (Software Hash IDentifiers)** [8,12,13] — currently progressing to an international standard under ISO/IEC 18670 [6] — offer a relevant and practical solution to unify software identification requirements in the EU Cyber Resilience Act, the U.S. Executive Orders, and broader international policy frameworks[1]. Combined with the *Software Heritage archival infrastructure* [7,14], SWHIDs enable transparent, verifiable identification of source code at various granularities (files, directories, version control systems commits, etc.) and, by extension, of any derived artifact.

In subsequent sections, we examine the current situation, which requires a reflection on the challenges related to the identifications of open source components (Section 2). Next, we analyze prevalent identifier architectures deployed in open source development and Software Bill of Materials (SBOM) standardization frameworks (Section 3). Building on this foundation, we introduce Software Heritage persistent Identifiers (SWHIDs), emphasizing their structural advantages and pivotal role in addressing compliance requirements outlined in modern regulatory regimes (Sections 4 and 5). Next, we formulate a strategic implementation roadmap to facilitate cross-sector adoption of SWHIDs, delineating actionable pathways for integration across governmental policy frameworks, industrial supply chains, and open source communities (Section 7). Finally, we summarize with a *Call to Action* for regulators and industry and OSS projects.

2 Context

2.1 Increased Attention On Software Supply Chain Security and Traceability

Regulatory pressure to secure the global software supply chain has mounted globally, propelled by new directives such as the CRA in the EU and recent Executive Orders in the U.S. The latter direct federal agencies to adopt strong software provenance practices, particularly for open source software. This mirrors the broader goal of ensuring that all software components — whether proprietary or open source — of IT products on the market are valid, trustworthy, and not vulnerable to hidden malicious code or supply chain compromises. Multiple convergent forces explain this heightened attention:

[1]This type of regulations is either currently in effect in various countries or emerging, depending on current political shifts. As such, these laws and regulations align with prevailing global trends, particularly in the Western world. It is therefore likely that similar legislation will continue to be enacted now and in the future. Consequently, we are focused on identifying the most appropriate way to address this need and are proposing a suitable solution. While politicians change, the underlying needs remain.

1. **High-profile compromises in open source libraries** have shown that a single compromised dependency can cascade into critical infrastructure [2,3,4].
2. **Widespread use of open source in government IT** demands a reliable way to track vulnerabilities and apply patches.
3. **SBOM (Software Bills of Materials) requirements** from regulators increasingly ask for granular, reliable references to each software artifact included in a product.
4. **Ephemeral code hosting issues.** Even when identifiable, software artifacts may disappear or move to different public hosting places, making it hard or impossible to retrieve them for subsequent inspection, audit, or reuse [23].

Among the biggest challenges is ensuring not only that each piece of software can be unambiguously identified but also that *the artifact corresponding to that identifier remains readily available and pristine* for the foreseeable future. Traditional naming schemes or platform-specific references can break if repositories are shut down or renamed or of their timeline/history is overwritten. Thus, a more permanent solution — one that guarantees ongoing accessibility and verifiability — is urgently sought.

2.2 "Software Identification" and its Challenges

Software identification is meant to *assign a stable, unique label to a software artifact* so that all stakeholders can refer to it unambiguously. Three broad challenges emerge:

1. **Forging Trust**: Government and industry strive to avoid ambiguity about which piece of code is in use. Conventional labels ("LibraryXYZ 2.0") can lead to ambiguity, name collisions, and stale references, which can translate into very concrete attack vectors in the current era of increasing and potentially very severe attacks targeting the open source software supply chain.
2. **Vulnerability Disclosures**: When security researchers disclose a new vulnerability (and related exploit, if available), its quick and accurate mapping to the affected software versions or individual files is essential.
3. **Compliance and Governance**: Regulatory frameworks increasingly insist on traceable software life cycles. When references are ephemeral or break over time, compliance with processes that expect software artifacts to remain available in the long-run is strongly jeopardized.

Moreover, ephemeral code hosting and a tangle of version-control practices make an all-encompassing "single source of truth" elusive. To address these issues, a *content-based* identification system — where the identifier of an artifact is derived from the artifact itself — appears uniquely suited to meet the long-term viability and security demands imposed by evolving cybersecurity regulations.

2.3 Emerging Regulatory Context (CRA and Beyond)

The EU's Cyber Resilience Act (CRA)

The Cyber Resilience Act (CRA) [11] in the European Union advocates stringent standards for transparency and software traceability. By mandating robust identification and vulnerability management, the CRA ensures that organizations can:

[2] `https://nvd.nist.gov/vuln/detail/cve-2024-3094`
[3] `https://nvd.nist.gov/vuln/detail/cve-2021-44228`
[4] `https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack`

- Enumerate the software components contained in an IT product using a **Software Bill of Materials (SBOM)** document.
- Track software updates and patches accurately.
- Comply with licensing and distribution requirements across jurisdictional boundaries.

U.S. Executive Orders on Cybersecurity

Recent U.S. Executive Orders [3,5] reflect parallel objectives:

- May 2021: Executive Order on Improving the Nation's Cybersecurity [3]
  Emphasizes "Enhancing Software Supply Chain Security" and underscores the need to "ensure and attest" the integrity and provenance of open source software present in IT products.
- January 2025: Executive Order on Strengthening and Promoting Innovation in the Nation's Cybersecurity [3]
  Explicitly recognizes the critical role of open source software in US federal systems, urging agencies to conduct security assessments, patch management, and best practices for OSS contributions.

Both sets of regulations converge in their *call for advanced identification techniques that can verify software origin*, track vulnerabilities, and meet transparent reporting obligations. **A coherent, persistent identifier infrastructure that accommodates the development practices of open source software would unify these objectives across agencies and industry players alike.**

3 Software Identifiers: Concepts and Approaches

Software identification has become a crucial area for software supply-chain security and transparency [3,22]. In practice, we observe two broad categories of identification mechanisms for software components: *intrinsic* and *extrinsic*. This section introduces these two notions, then surveys several major existing schemes, discussing their advantages and limitations.

3.1 Intrinsic vs. Extrinsic Identifiers

Intrinsic Identifiers. An *intrinsic* identifier is derived directly and solely from an artifact's content (for instance, via a cryptographic hash computed on a byte-sequence representation of the artifact). Since the identifier depends on the actual bits of the file or source, it provides strong guarantees of *uniqueness* and *integrity verification* without requiring a central authority [13]. Examples of intrinsic identifiers for software artifacts include pure SHA256 checksums, as well as the Software Hash IDentifiers (SWHIDs) [6,12], which are typed and incorporate Merkle DAG hashing of file contents and directory structures. Intrinsic identifier schemes typically excel at:

- **Tamper detection:** Re-computing the hash quickly reveals if an independently-retrieved artifact has been altered since its original identifier was computed.
- **Minimal external dependencies:** No need for a registry or naming authority to maintain unique IDs.
- **Collision resistance:** If well-chosen cryptographic hashes are used, hash collisions are astronomically unlikely.

A drawback of intrinsic identifiers is often *limited human readability*, which can complicate searching or "brand recognition." Hence, purely intrinsic IDs may be inconvenient for referencing software in vulnerability databases or licensing catalogs.

Extrinsic Identifiers. An *extrinsic* identifier ties software identity to externally maintained metadata such as vendor and product strings, version labels, or references to package manager repositories/specific software ecosystems [1, 2, 4, 19]. These approaches ease *human use* — because names and versions can be crafted in ways that are more recognizable than cryptographic hashes — and they can integrate with well-established registries (e.g., the National Vulnerability Database [9]). However, extrinsic identifiers rely on:

- **External registries or naming conventions:** A product name or ecosystem-based ID depends on the whims of the namespace maintainers, that can decide to recycle it, making it point to different products without downstream identifier users being able to stop that or even detect it.
- **Potential conflicts or name collisions:** Different ecosystems may reuse the same name or version scheme for unrelated artifacts.
- **Administrative overhead:** Maintaining a large dictionary or "vendor" field, for instance, can be costly.

These issues with extrinsic identifiers for software modules have been known for decades, see, e.g., [24], but only recently their potentially catastrophic impact on security has started to be noticed: dependency confusion arising from extrinsic identifiers [17] have been shown as very efficient attack vectors [10, 15, 18] which impact a large spectrum of industry players [20].

Hence, while extrinsic identifiers can be highly effective for discoverability, they do not provide sufficient guarantees for integrity and security.

3.2 Major Existing Identification Approaches

Table 1 summarizes prominent software identifier schemas, their categorization, and key references.

Table 1: Identifier schemes for software artifacts or products, and their classification as intrinsic (content-based) or extrinsic (externally assigned).

Scheme	Type	Notes / References
SWHID (ISO/IEC 18670)	Intrinsic	Hash-based, derived from Merkle DAG [6, 13].
Checksum (SHA256)	Intrinsic	Simple direct hash of artifact bytes [22].
SWID (ISO/IEC 19770-2)	Extrinsic	Tag assigned by vendor [1, 25].
purl	Extrinsic	Ecosystem-based package name and version [4, 22].
CPE	Extrinsic	Dictionary-based (vendor, product, version) [19].
SPDXID	Extrinsic	Arbitrary label in an SBOM [2].

SWID (ISO/IEC 19770-2) (not to be confused with SWHID below) SWID tags embed identification metadata and optional file hashes in an XML file distributed with installed software [1]. The "tagId," however, is typically assigned by the software publisher and is not purely derived from the artifact's bits.

- *Pros:* Integrates well with asset and license management tools [25]; recognized ISO standard.
- *Cons:* Dependent on vendor definitions, meaning *extrinsic* identity. Hashing is optional.
- *Category:* Extrinsic (with optional intrinsic hashes included).

Package URL (purl) Package URL (purl) is a standardized URL-like format to identify software packages based on ecosystem, namespace, name, and version [4, 22].

- *Pros:* Provides a uniform reference across multiple package registries; widely used in SBOM tooling [21].
- *Cons:* Not derived from the file content; depends on naming in package ecosystems or development forges.
- *Category:* Extrinsic.

Common Platform Enumeration (CPE) CPE uses a standardized name format, e.g., `cpe:2.3:a:vendor:product:version`, heavily employed by the National Vulnerability Database [9, 19].

- *Pros:* Strong alignment with vulnerability data (CVEs).
- *Cons:* Requires consistency in "vendor" and "product" fields; collisions and ambiguities occur if naming is inconsistent.
- *Category:* Extrinsic.

SWHID (ISO/IEC 18670) (not to be confused with SWID above) Software Hash IDentifiers (SWHIDs) are purely content-based, leveraging Merkle DAG hashing of file contents, directories, and objects commonly found in version control systems such as commits, tags, and repository states [6, 8, 12, 13].

- *Pros:* Provide guaranteed integrity checks; no single authority needed for ID creation; well-suited to archiving.
- *Cons:* Cryptographic hashes are not human-meaningful.
- *Category:* Intrinsic.

Checksum-Based IDs (e.g., SHA256) The simplest intrinsic approach is a raw hash checksum. Many SBOM or distribution workflows embed SHA256 or SHA512 checksums to confirm authenticity or detect tampering [22].

- *Pros:* Easy to compute; trivial to verify if content matches.
- *Cons:* Storing only a checksum says nothing about naming, version, or location; no direct link to vulnerabilities or licenses.
- *Category:* Intrinsic.

SPDX Identifiers SPDX [2] is a Linux Foundation project to define a standard SBOM format. An "SPDXID" references each file or package in an SBOM, but it is assigned arbitrarily (e.g., `SPDXRef-Package`), thus extrinsic. Nonetheless, users often include hash checksums in the SPDX file for verification [16].

- *Pros:* Popular in open source compliance; supports embedding multiple referencing schemes (SWID, SWHID, purl, checksums).
- *Cons:* The core "SPDXID" itself is not content-derived; extrinsic by default.
- *Category:* Extrinsic (with optional intrinsic hashes included).

3.3 Key Takeaways

Real-world SBOMs and supply-chain tooling often combine both **extrinsic** references (to integrate with existing databases, vulnerability feeds, or licensing tools) and **intrinsic** references (for strong integrity checks and guaranteed uniqueness) [22].

As a result, best-practices may encourage publishing, for instance, a *purl* or *SWID plus* a cryptographic hash or SWHID to ensure both discoverability and verifiability.

4 Software Hash IDentifiers (SWHIDs) and Software Heritage

4.1 Software Hash IDentifiers (SWHIDs)

Software Hash IDentifiers (SWHIDs) [8,12,13] are persistent, content-based cryptographic identifiers designed to reference software source code artifacts such as source code files, source trees, commits, and other objects typically found in version control systems. They integrate well with the Software Heritage archive [7, 14] and can be used to reference any object archived there, but they are not tied to the archive and are used more broadly in the IT market already. Key aspects of SWHIDs include:

- **Content-based**: By hashing byte representations of the content and metadata of the referenced object, even a single-bit change in the object yields a different identifier.
- **Persistent and Tamper-Resistant**: SWHIDs do not rely on resolvers or hosting domains. They are rooted in cryptographic hashes that remain valid regardless of where the code is hosted.
- **ISO/IEC 18670 Standardization**: The upcoming standard [6] formalizes these identifiers, making SWHIDs an internationally recognized solution.

SWHIDs allow to cryptographically encapsulate both the data content and its context within a Merkle DAG structure [6], **ensuring that each identifier is intrinsically tied to the exact artifact it references**.

At their core, SWHIDs conform to the following syntax:

$$\texttt{swh:}\langle \textit{schema\_version} \rangle\texttt{:}\langle \textit{object\_type} \rangle\texttt{:}\langle \textit{object\_id} \rangle$$

where:

- ⟨*schema_version*⟩ identifies the version of the SWHID standard (e.g., `1` for the current version);
- ⟨*object_type*⟩ indicates the type of the referenced object, one of: `cnt`, `dir`, `rev`, `rel`, or `snp` (explained below);
- ⟨*object_id*⟩ is a cryptographic hash (e.g., SHA1) uniquely identifying the referenced object.

Since each identifier is derived from the underlying artifact itself, duplication is eliminated at scale when using SWHIDs to identify objects within large collections: *identical* files (or directories, revisions, etc.) across multiple repositories map to the *same* SWHID.

Core SWHID Categories. The SWHID specification [6] defines five core object types, each associated to a dedicated object type:

1. *Content (cnt)*: A *single file blob* whose identifier is derived by hashing its raw content as a byte sequence. For example:

   ```
   swh:1:cnt:94a9ed024d3859793618152ea559a168bbcbb5e2
   ```

   Any change to the file's bytes yields a distinct hash.

2. *Directory (dir)*: A tree object referencing a directory-like objects, with all the files and/or subdirectories it contains. The hash covers not only the identifiers of these child entries but also metadata such as file names. For example:

```
swh:1:dir:d198bc9d7a6bcf6db04f476d29314f157507d505
```

This ensures two directories with exactly the same contents in the same structure map to the same identifier. Conversely, any change in any of the contained files or directories, or associated metadata, will result in a different SWHID.

3. *Revision (rev)*: Commonly referred to as a *commit*, it points to a root directory and includes commit metadata (timestamp, author, parent commits, message, etc). For instance:

```
swh:1:rev:309cf2674ee7a0749978cf8265ab91a60aea0f7d
```

4. *Release (rel)*: Analogous to a *tag* in version-control systems. It references a specific revision and may include a user-friendly version name, descriptive message, cryptographic signature, etc. For example:

```
swh:1:rel:22ece559cc7cc2364edc5e5593d63ae8bd229f9f
```

5. *Snapshot (snp)*: Captures the state of all branches of an entire version control system repository at the time the SWHID is computed, mapping branch names to specific revisions or releases. For example:

```
swh:1:snp:c7c108084bc0bf3d81436bf980b46e98bd338453
```

Such content-based identifiers outlast the lifetime of any particular hosting platform. Even if a forge ceases operation or a repository is renamed or rewritten in a destructive manner, **the SWHID persists indefinitely because it reflects the artifact's *content* rather than its external location.**

Qualified Identifiers. Beyond these core identifiers, *qualified* SWHIDs can include optional *qualifiers* that provide additional context or specificity [6]:

- *lines qualifier* (`lines=...`): Indicates line ranges within a file. For example:

  ```
  swh:1:cnt:94a9ed024d3859793618152ea559a168bbcbb5e2;lines=112-116
  ```

  references only lines 112 to 116 of a particular content object.

- *origin qualifier* (`origin=...`): Captures the URL (or other location) where the artifact was originally observed. For instance:

  ```
  swh:1:rev:309cf2674ee7a0749978cf8265ab91a60aea0f7d;
  origin=https://github.com/example/repo
  ```

- *anchor, path, ...* and other contextual qualifiers: These can pinpoint subdirectories, sub-elements, or further metadata crucial for a precise reference. For example:

  ```
  swh:1:dir:d198bc9d...;path=/docs;anchor=readme-section
  ```

By including these qualifiers, SWHIDs offer a rich, yet stable, method for referencing not only entire objects but specific portions within them or outside contexts. They remain backward-compatible with the core format, ensuring that stakeholders can resolve the essential reference even if they do not recognize or care about the added qualifiers.

4.2 Software Heritage as a Universal Archive

Software Heritage is a non-profit initiative that archives publicly accessible source code along with its development history [12]. Once ingested, code is never deleted. This ensures:

- **Comprehensive Coverage**: Continuous crawling of major forges, distribution archives, and package repositories. Software Heritage is the largest public archive of source code and accompanying development history having archived, as of March 2025, more than 23 billion unique source code files and almost 5 billion unique version control system commits, coming from more than 350 million software projects.
- **Verifiability**: Any artifact can be retrieved from the archive, ensuring long-term availability regardless of repository closures or reorganizations.
- **Merkle-DAG Storage**: Source code artifacts are stored in a cryptographically sound data structure, enabling scalable de-duplication and reliable integrity checks [13].

With these features, SWHIDs and the Software Heritage archive jointly form a solid backbone for compliance, reproducibility, and vulnerability management — fully aligned with the goals articulated in both the CRA and U.S. Executive Orders.

5 Comparative Analysis of Existing Identifiers

Existing identification methods — such as package manager names, Git commit hashes, or platform-specific references — offer limited guarantees of future verifiability, especially if the original repository disappears or version tags are renamed. None inherently guarantees the *persistence* and *global verifiability* that emerging regulations demand. Common pitfalls include:

- **Platform Lock-in**: Vendor or hosting site might go offline, breaking references.
- **Namespace Collisions**: Multiple projects or forks reuse the same naming convention, causing confusion.
- **No Lifelong Guarantee**: If the source code moves, the original identifier no longer resolves.

By contrast, **SWHIDs** (in conjunction with Software Heritage) address these shortcomings through a content-based, technology-agnostic approach. This architecture provides a reliable foundation for policy compliance and software best practices alike [22].

5.1 Key Advantages of SWHIDs with Qualifiers

Qualified SWHIDs not only provide precise, fine-grained references to software artifacts but also combine *intrinsic* and *extrinsic* identification in a single schema, offering three distinct benefits:

1. **Intrinsic Identification (Cryptographic Guarantees).** Because SWHIDs are computed from the *content* of each artifact, they deliver tamper-proof uniqueness. Any minor change to the underlying data alters the cryptographic hash, producing a *different* identifier. This property endures regardless of where or how the artifact is hosted, thereby preventing collisions or ambiguities.
2. **Traceability Within the Software Heritage Archive.** Since each artifact referenced by a SWHID can identify a node in the Software Heritage Merkle DAG (provided that the referenced artifact has actually been archived there), qualified SWHIDs integrate seamlessly with the archive's structure. Revisions, directories, and other nodes are interlinked,

enabling stakeholders to *traverse the development history* of a project, pinpoint code provenance, and verify relationships between artifacts. With qualifiers such as `lines` or `path`, one can even trace the use or evolution of specific fragments of a code base, solidifying the global audit record of collaborative software development.

3. **Extrinsic Context (Availability and Origin).** The `origin` qualifier captures the external source from which the software artifact was fetched. Crucially, because Software Heritage also *archives* that origin, it preserves availability for future reference. Even if the original repository goes offline or relocates, the archive holds a snapshot of that location, ensuring that *both* the intrinsic hash and the extrinsic origin URL remain resolvable. Hence, SWHIDs combine the best of both worlds: robust content-based identification, enriched by the historical and contextual information provided by extrinsic references.

By coupling intrinsic cryptographic guarantees with the broader traceability offered by the Software Heritage archive — and by recording where the artifacts were first encountered — SWHIDs with qualifiers deliver a comprehensive, future-proof approach to software identification.

5.2 Impact of SWHIDs on CRA, U.S. Executive Orders, and Other Regulations

Mutual Reinforcement of Global Mandates

The **CRA's** focus on transparency, coupled with the **U.S. Executive Orders'** emphasis on secure open source usage, sets a global trend in favor of deep software traceability. Governments and industries share overlapping concerns: rapid security patching, unambiguous references, and reproducibility.

SWHID: Essential to Meeting Regulatory Requirements

- **SBOM Precision**: Tying each component to a cryptographic identifier ensures the SBOM remains accurate over time, even if code moves to another location.
- **Regulatory Alignment**: Because SWHIDs do not depend on ephemeral structures, they are well suited to cross-jurisdictional regulations, whether in the EU, U.S., or elsewhere.
- **Open Source Compliance**: The calls to secure OSS supply chains require that each OSS artifact is referenced unambiguously. SWHIDs fulfill this requirement by linking code to an immutable record within the Software Heritage archive.

6 Proposed Position: SWHID as the Cornerstone

Position Statement: SWHIDs, backed by the Software Heritage archive, should serve as the default reference mechanism for software artifacts under the Cyber Resilience Act and related U.S. Executive Orders.

Critical Use Cases

1. **Vulnerability Management**: SWHIDs let security teams rapidly and unambiguously locate at-risk components.
2. **Software Bill of Materials (SBOM)**: Defining SBOM entries via SWHIDs streamlines compliance and automates patch workflows.

3. **Provenance Tracking**: The open source emphasis in the U.S. Executive Orders particularly benefits from a tool to confirm the lineage of code. SWHIDs provide these assurances cryptographically.

Benefits

- **Global Interoperability**: Works across different version-control systems and package ecosystems.
- **Longevity**: Once assigned, the identifier outlasts hosting changes or platform shutdowns.
- **Auditable Development**: The Merkle-DAG structure — which underpins how SWHIDs are computed and is materialized by the Software Heritage archive—pinpoints the entire evolution history of a software product, fostering accountability and transparency for both public and private code bases.

7 Implementation and Roadmap

7.1 Strategies for Stakeholders

1. **Policy Makers**:
   - Reference **ISO/IEC 18670** [6] (SWHIDs) in CRA guidelines and U.S. federal compliance frameworks.
   - Incentivize the usage of SWHIDs across government procurements and open source community funding programs.
2. **Software Vendors**:
   - Incorporate SWHID generation tools into CI/CD pipelines, ensuring that each release and patch has a stable, verifiable ID.
   - Ensure that the open source parts of products — be they developed in-house or reused from 3rd parties — are archived by Software Heritage.
   - Collaborate with major forges to standardize references to SWHIDs in compliance reports.
3. **Open Source Communities**:
   - Publish official releases with associated SWHIDs, making the trust chain more straightforward for government and enterprise users.
   - Ensure that code and associated development history is archived by Software Heritage, especially but not only at release time.
   - Adopt best practices for referencing third-party dependencies by SWHIDs in documentation and SBOMs.

7.2 Recommended Steps and Timeline

- **Short Term (6–12 Months)**: Incorporate SWHIDs into existing SBOM generation tools, referencing them in procurement guidelines and vulnerability advisories.
- **Medium Term (1–2 Years)**: Formalize SWHIDs in major open source project governance. Create or extend standards (SPDX, CycloneDX) to reflect SWHIDs for all dependencies.

- **Long Term (>2 Years)**: Normalize SWHIDs as a fundamental layer in both EU and U.S. regulatory requirements. Support universal integration so that regulatory audits rely on content-based identifiers by default.

8 Conclusion and Call to Action

In an era defined by systemic cybersecurity challenges, *content-based, persistent software identifiers* offer a uniquely powerful response to new regulatory demands. The Cyber Resilience Act and U.S. Executive Orders on cybersecurity collectively affirm that robust, verifiable identification is necessary to ensure trust, compliance, and innovation in the software ecosystem.

Coupling **SWHIDs** and the **Software Heritage** archive provides:

1. **Trust and Transparency**: Stakeholders can unequivocally track each artifact's provenance.
2. **Regulatory Alignment**: Mandates on SBOM, open source security, and patch management can be met with minimal friction.
3. **Innovation and Cost Savings**: Government agencies and the broader ecosystem benefit from simplified compliance checks and lower risk of supply chain compromise.

Call to Action:

- **Regulators**: Incorporate SWHIDs explicitly into both the CRA implementation guidance and the U.S. federal software procurement rules.
- **Industry and OSS Projects**: Adopt SWHIDs in continuous integration pipelines and SBOM tooling, ensuring consistent usage and verifiability.
- **Sustainability**: Continue investing in the open, neutral mission of Software Heritage to ensure perpetual access to the reference archive.

As evolving cybersecurity legislation accelerates the need for persistent and reliable software identification, **SWHIDs and Software Heritage** stand ready to serve as a cornerstone of next-generation software supply chain security.

Bibliography

[1] ISO/IEC 19770-2:2015 - Information technology – Software asset management – Part 2: Software identification tag, 2015. URL: https://www.iso.org/standard/65666.html.

[2] Software package data exchange (spdx) specification, version 2.2. https://spdx.github.io/spdx-spec, 2020.

[3] Executive order 14028: Improving the nation's cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-improving-the-nations-cybersecurity/, 2021. Last accessed 2023-12-01.

[4] Package URL (purl) Specification. https://github.com/package-url/purl-spec, 2021. Last accessed 2023-12-01.

[5] Executive order 14144: Strengthening and promoting innovation in the nation's cybersecurity. https://www.federalregister.gov/documents/2025/01/17/2025-01470/strengthening-and-promoting-innovation-in-the-nations-cybersecurity, 2025. Last accessed 2025-03-19.

[6] ISO/IEC 18670 - Information technology – Software Hash Identifier, 2025. URL: https://www.iso.org/standard/81554.html.

[7] Software heritage. https://www.softwareheritage.org, 2025. Last accessed 2025-03-19.

[8] SWHID working group. https://www.swhid.org, 2025. Last accessed 2025-03-19.

[9] National vulnerability database (nvd), Ongoing. Accessed 2023-01-10. URL: https://nvd.nist.gov.

[10] ActiveState. Dependency confusion. Accessed: 2025-03-15. URL: https://www.activestate.com/resources/quick-reads/dependency-confusion/.

[11] European Commission. Cyber resilience act. https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act, 2024. Last accessed 2025-03-19.

[12] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. Identifiers for digital objects: The case of software source code preservation. In *Proceedings of the 15th International Conference on Digital Preservation (iPRES 2018)*, Boston, MA, USA, September 2018. URL: https://hal.inria.fr/hal-01865790.

[13] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. Referencing source code artifacts: a separate concern in software citation. *IEEE Computing in Science & Engineering*, 22(5):33–46, September 2020. URL: https://ieeexplore.ieee.org/document/8847248, doi:10.1109/MCSE.2019.2963148.

[14] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017: 14th International Conference on Digital Preservation*, 2017.

[15] Pascal Geenens. Dependency confusion attacks, February 2023. Accessed: 2025-03-15. URL: https://www.radware.com/blog/security/threat-intelligence/2023/02/dependency-confusion-attacks/.

[16] Ibrahim Haddad. Strengthening license compliance and software security with SBOM adoption: A definitive SBOM guide for enterprises. Technical report, August 2024. URL: https://www.ibrahimatlinux.com/wp-content/uploads/2024/08/SBOM_Report-082024.pdf.

[17] Jacob. Package dependency resolution in nuget and naming conflicts, January 2014. Accessed: 2025-03-15. URL: `https://stackoverflow.com/questions/21153475/package-dependancy-resolution-in-nuget-and-naming-conflicts`.

[18] Eyal Katz. 5 examples of dependency confusion attacks, February 2025. Accessed: 2025-03-15. URL: `https://spectralops.io/blog/5-examples-of-dependency-confusion-attacks/`.

[19] National Institute of Standards and Technology (NIST). Common platform enumeration (cpe). Security Content Automation Protocol (SCAP) Specifications, Ongoing. Accessed: 2024-03-16. URL: `https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe`.

[20] Franklin Okeke. Dependency confusion attacks: New research into which businesses are at risk, August 2023. Accessed: 2025-03-15. URL: `https://www.techrepublic.com/article/dependency-confusion-attacks-new-research-into-which-businesses-are-at-risk/`.

[21] Philippe Ombredanne. Foss component identification with package urls. Presentation, March 2023. URL: `https://aboutcode.org/wp-content/uploads/2023-03-30-FOSS-purl.pdf`.

[22] NTIA Multistakeholder Process on Software Component Transparency. Software Identification: Challenges and Guidance. Report, National Telecommunications and Information Administration (NTIA), March 2021. Last accessed 2023-12-01. URL: `https://www.ntia.gov/files/ntia/publications/ntia_sbom_software_identity-2021mar30.pdf`.

[23] Can Ozkan, Xinhai Zou, and Dave Singelee. Supply chain insecurity: The lack of integrity protection in sbom solutions, 2024. URL: `https://arxiv.org/abs/2412.05138`, `arXiv:2412.05138`.

[24] Guy L. Steele. Common lisp the language, 2nd edition. chapter 11.5: Name Conflicts. Digital Press, Burlington, MA, 1990. Accessed: 2025-03-15. URL: `https://www.lix.polytechnique.fr/~liberti/public/computing/prog/lisp/cltl/clm/node116.html`.

[25] David Waltermire, Brant A Cheikes, Larry Feldman, and Greg Witte. Guidelines for the creation of interoperable software identification (SWID) tags. Technical report, April 2016. URL: `https://nvlpubs.nist.gov/nistpubs/ir/2016/nist.ir.8060.pdf`.