Bytewise and Externalized SAIDs

Daniel Hardman

2026-01-24

Abstract

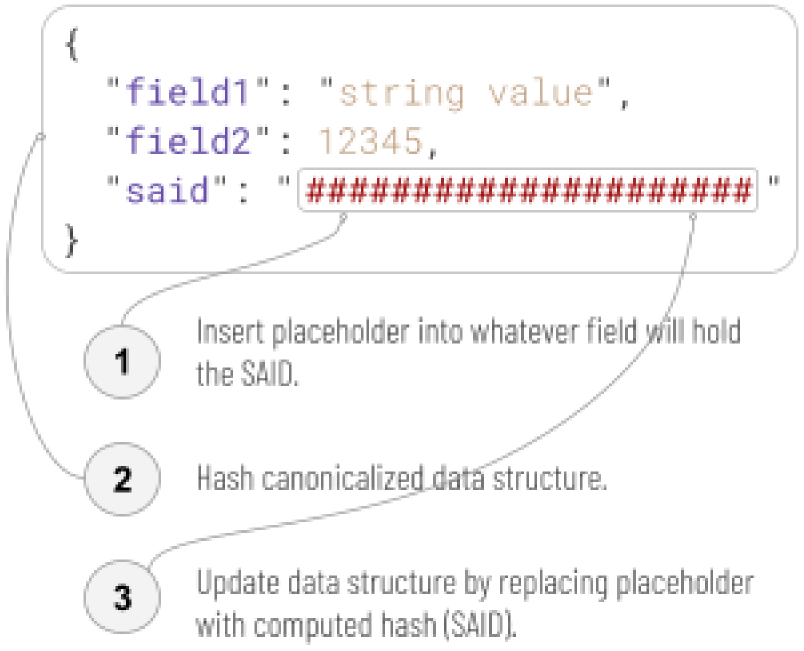
Self-certifying identifiers enable decentralized, tamper-evident references to data and support the construction of authenticated graphs with strong integrity and caching properties. Existing self-addressing identifier (SAID) mechanisms, including those defined in the CESR specification, assume data structures that can be canonically serialized and safely rewritten after hashing. However, many widely used file formats—such as documents, media files, and proprietary application artifacts—are opaquely structured and cannot be modified or normalized without specialized tooling.

This paper introduces two generic algorithms that extend self-addressing identifiers to arbitrary files treated as byte streams. The first embeds a self-certifying identifier directly within file content using a delimiter-based placeholder scheme, while the second externalizes the identifier to a constrained filename when in-place modification is impractical. Together, these approaches allow most file types to participate in authenticated, content-addressable data graphs, provided that authors can introduce an insertion point (or exsertion instruction) at creation time and that subsequent handling preserves the file's bytes (or else any byte-changing transformation is treated as producing a new identity).

1. CESR SADs and SAIDs

The CESR spec defines self-addressing data (SAD) as a data structure that contains a special kind of reference to itself — a self-addressing identifier (SAID). Space for the SAID is initially reserved in the SAD by storing a placeholder value. Once full, the data structure is canonicalized and hashed. The output hash is then encoded using CESR rules to make it self-describing, and inserted into the SAD, replacing the placeholder. [1]

When a SAD holds its SAID, the correspondence between the two can be checked, making the data structure versioned and authenticatable. [2, 3, 4, 5, 6, 7] Any change to the SAD or SAID breaks the correspondence. The SAID can also be used by itself as a tamper-evident reference to or compression of its associated SAD, enabling extremely efficient caching. Embedding additional, foreign SAIDs in a given SAD is a linking strategy that enables complex and decentralized graphs of authenticated data. These graphs are far more efficient, secure, and robust across untrusted storage and internet disruption than the generic Semantic Web, which has no such guarantees. They are a core building block of authentic chained data containers (ACDCs). [8]


{
"field1": "string value",
"field2": 12345,
"said": "######################"
}
1 Insert placeholder into whatever field will hold the SAID.
2 Hash canonicalized data structure.
3 Update data structure by replacing placeholder with computed hash (SAID).

2. Opaquely Structured Data

Imagining the relationship between SAD and SAID as one of a data structure to a field works well for structured serialization formats like JSON, CBOR, or MsgPack. However, what if we want to securely reference, share, and cache data that isn't structured the way CESR's saidification algorithm expects? The world is full of data like this. Consider YouTube's video catalog, or the set of documents on a company's SharePoint, or a personal collection of digital photos, or an invoice attached to an email as a PDF. It would be helpful if such data could also participate in efficient, robust, authenticated data graphs.

Some excellent work has been done in this area. IPFS allow a file to be hashed and added to a global, decentralized web of content-addressable storage. Content identifiers (CIDs) in IPFS are somewhat like SAIDs. [9] Ethereum Swarm [10] and Git Large File Storage [11] also have analogs. However, these systems are primarily container-oriented: they focus on storage and retrieval of the data, not on the more limited question of how to model its identity and interrelationships. They therefore do too much to be a pure solution to our current challenge.

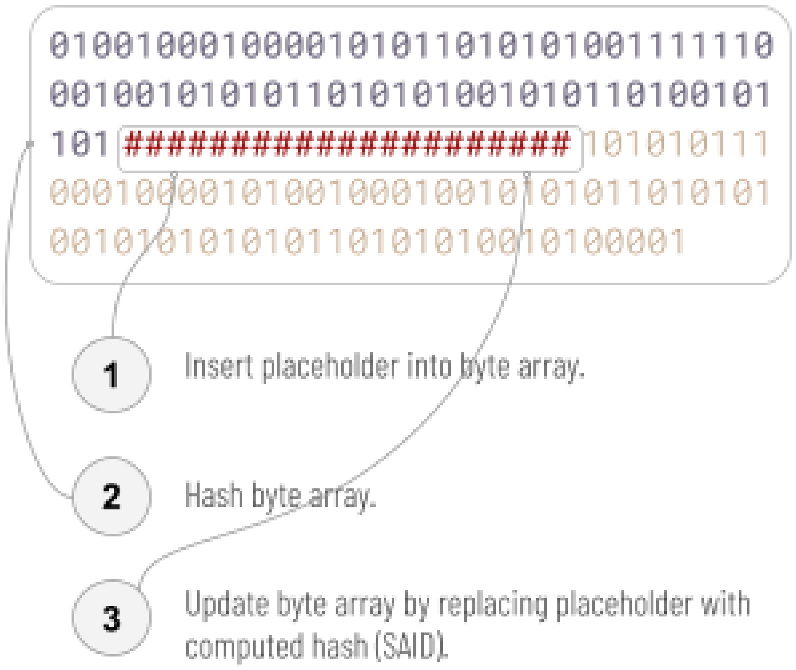
Data formats such as video, documents, and spreadsheets do embody data structures — often, dramatically sophisticated ones — but programmatic access to them tends to run through an application or an API and a library that hides fields and structural details. In many cases, the formats are also wholly or partly proprietary. Both of these characteristics lead us to describe such formats as opaquely structured.

We could imagine adding a custom field to hold a SAID in an opaquely structured data format. However, fields in such formats may be invisible to, or even stripped by, handling software. Even if we could solve the mechanics, computing the hash of the data structure minus the SAID field is problematic, since each format has its own canonicalization rules, and they are often poorly or entirely undefined in public documentation. Even more challenging, the set of opaquely structured data formats grows constantly.

3. Solution

3.1 Byte array

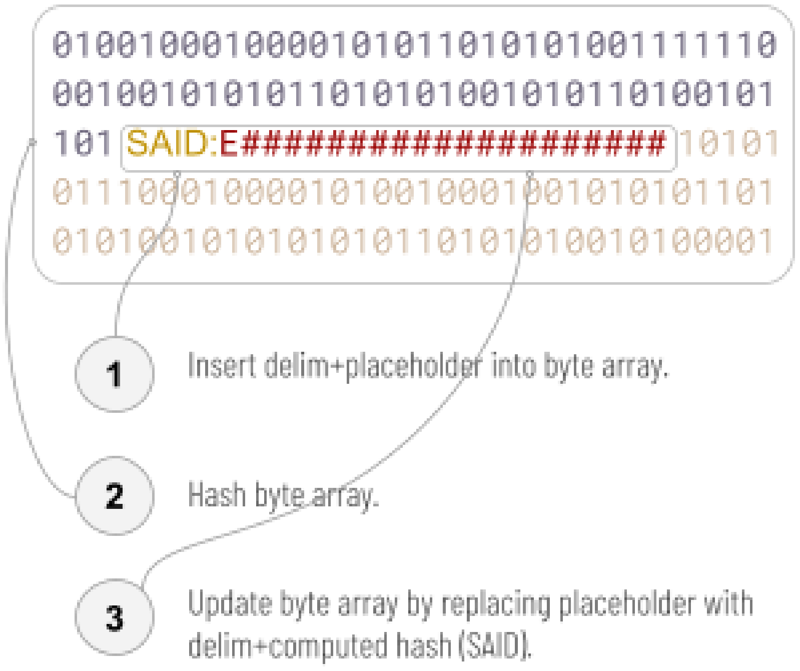
The first step toward a solution is to embrace the opacity of these formats and simply view them as byte arrays (or alternatively, as a 3-field data structure: bytes-before, SAID, bytes-after). This completely eliminates canonicalization problems — any bytewise difference is significant — and it gives us an easy and wholly generic, robust saidification algorithm. Rather than traversing structure to find a field in the data, we simply scan the byte stream for the placeholder:



3.2 Delimiters

Although this makes saidification easy, it creates a new problem, which is that there's no reliable way to find the SAID once it's been inserted. After all, the SAID is a byte sequence that we can't predict in advance.

The simple solution is to use delimiters in both step 1 and step 3:


0100100010000101011010101001111110
0010010101011010101001010110100101
101 SAID:E######################## 10101
0111000100001010010001001010101101
0101001010101010110101010010100001
1 Insert delim+placeholder into byte array.
2 Hash byte array.
3 Update byte array by replacing placeholder with delim+computed hash (SAID).

We call this modified algorithm (including minor enhancements described in 3.5 below) the *bytewise SAID algorithm*, a SAD that uses it a *bytewise SAD*, a SAID produced by it a *bytewise SAID* (or *bSAID* for short), and a delim+placeholder an *insertion point*.

Notice that in addition to adding a delimiting prefix, Figure 3 changes the placeholder from a series of red # characters to red # characters preceded by an "E" character. This is just an example. The formal requirement for insertion point syntax is that the sequence consist of the 5-byte delimiter "SAID:", followed by either a SAID template or an actual SAID. A SAID template is a 1- or 2-character CESR primitive code for a digest (e.g., "E" for a Blake3 hash), followed by as many "#" bytes (0x23, number sign when interpreted as ASCII) as are required to make the template exactly as long as the SAID that will replace it (e.g., 43 #s for a Blake3 hash, since the full length of a Blake3 SAID is 44, including the "E" prefix). In ABNF:

3.2.1 ABNF

```
insertion_point = "SAID:" placeholder
placeholder = template / said

code44 = "E" / "F" / "G" / "H" / "I"
code88 = "0D" / "0E" / "0F" / "0G"

template = (code44 "#"*43 / code88 "#"*86)
said     = (code44 base64url-43 / code88 base64url-86)

; Base64url encoding per RFC 4648, Section 5 (no padding)
base64url-char = ALPHA / DIGIT / "-" / "_"
```

```
base64url-43   = 43base64url-char
base64url-86   = 86base64url-char
```

The `base64url` productions above follow the unpadded base64url encoding defined in RFC 4648, Section 5. [12] Padding characters (=) are not permitted, and the encoded length is fixed by the selected CESR digest code.

A regex that correctly matches insertion points in an arbitrary byte stream is:

3.2.2 Regex

```
SAID:([EFGHI](?:[A-Za-z0-9_-]{43}|#{43})|0[DEFG](?:[A-Za-z0-9_-]{86}|#{86}))
```

Note that this regex is case-sensitive and MUST be applied to a byte stream (for example, using a raw byte pattern such as `rb"..."` in Python), not to Unicode text. The character class is written explicitly to avoid locale- or implementation-dependent behavior.

Recall that standard JSON saidification accepts any JSON object, normalizes it, finds a field that will hold the new SAID (using a field named "d" as a default), replaces any prior value of that field with a string that contains exactly as many # characters as the SAID will replace, then hashes the data and replaces the # characters with the SAID.

In contrast, the bytewise SAID algorithm does no normalization, since we know nothing about the format of the byte stream. It locates the insertion point by searching for the appropriate byte pattern, puts the placeholder into template form if it is an actual SAID value, then hashes the entire byte stream. Thus, the bytewise SAID algorithm might hash a byte stream containing "E"+ 43 "#" bytes, whereas the JSON algorithm might hash a byte stream containing 44 "#" bytes. The reason for this divergence is that it is not dangerous to change the size of a JSON file, but it may be dangerous to change the size of a byte stream in an unknown format. Therefore, the placeholder MUST contain a prefix that specifies which of several SAID types the document author intends, and the number of "#" characters that follow must add up to the exact length of the new SAID, so the file size will be unchanged.

After saidification, we can find the SAID by searching the bytes for the delimiter, "SAID:", followed by a recognizable SAID.

For determinism, a bytewise SAD MUST contain exactly one primary insertion point. If multiple byte sequences match the insertion-point syntax, the leftmost occurrence in the byte stream is treated as the primary insertion point, and all others are interpreted as echoes as described in Section 3.5.1. A file that contains no valid insertion point, or that contains multiple non-identical insertion points, is invalid input to the algorithm.

3.3 Inserting the SAID

So far, our solution has glossed over how we choose a location for the SAID. Should it go at the beginning, the middle, or the end?

The answer to this question is usually uncomplicated: we prefer a location where a file format allows document metadata. Most opaquely structured file formats offer explicit support for arbitrary metadata. HTML allows `<meta>` tags; PDFs and most Adobe file formats support XMP metadata; JPEG and MPEG support Exif; Microsoft Office formats support arbitrary keyword tagging. We can insert a delim+placeholder using the native tools of an opaquely structured file type (e.g., with application features or a library/API), then save the file, open the resulting byte stream, and saidify using our naive bytewise SAID algorithm.

It may seem like inserting our SAID with a library/API is a troublesome new requirement (a dependency on a file format library); after all, saidification of JSON just requires JSON features from a programming language. Our initial algorithm in 3.1 was even more primitive in its requirements. However, three considerations prevent this new dependency from being a burden:

1. If software is already creating opaquely structured data anyway, then it is extremely likely to be doing so through an API/library; the need for a library is a defining characteristic of that data category even before we saidify. (Any software that deals with opaquely structured data as raw bytes must have minimal library-like knowledge. Example: code can write HTML as raw bytes, but if it does, it probably understands tags a little bit, so asking it to add a `<meta>` tag is trivial.)
2. Ordinary users don't need a library or upgraded tools. They can create the SAID metadata using existing, tested features in whatever programs they already use to work with the content. There is good documentation about how to do it, and we don't have to maintain it.
3. We are only imposing this library requirement on writers, not readers. Readers can discover the SAID value with a generic, naive byte scan, written once for all file types.

An additional benefit of this approach is that many metadata schemes for opaquely structured data already recognize a particular metadata item that maps to the "identifier" concept in Dublin Core (ISO 15836, a metadata standard). [13] This happens to be the very semantic that SAIDs express. So, by setting metadata for the identifier concept using the `SAID:<value>` convention, we are simultaneously giving the document a SAID, and giving it a permanent identifier that existing, SAID-ignorant metadata features will accept. This is an interoperability and backward compatibility win, and it introduces a huge audience to SAIDs without any special effort, tooling, or documentation from SAID proponents.

A few opaquely structured data formats lack explicit metadata support. For example, source code written by programmers could be considered opaquely structured data, and there is no universal metadata mechanism for source code. Markdown is another format without metadata (unless you count YAML prefixes [14]). If a format doesn't support metadata, then we give the fallback answer: insert the SAID as a comment, using whatever commenting mechanism the format recognizes. This answer works even if the comment is as primitive as a parenthetical note in ordinary, human-editable sentences that comprise the text of a file. For example, if a chef had software for recording recipes, one per file artifact, and it lacked metadata support, the chef could simply put a note at the end of every recipe's instructions: "(SAID: <SAID value>)".

3.4 Externalized SAIDs

For many opaquely structured data formats, the bytewise SAID algorithm is simple and sufficient.

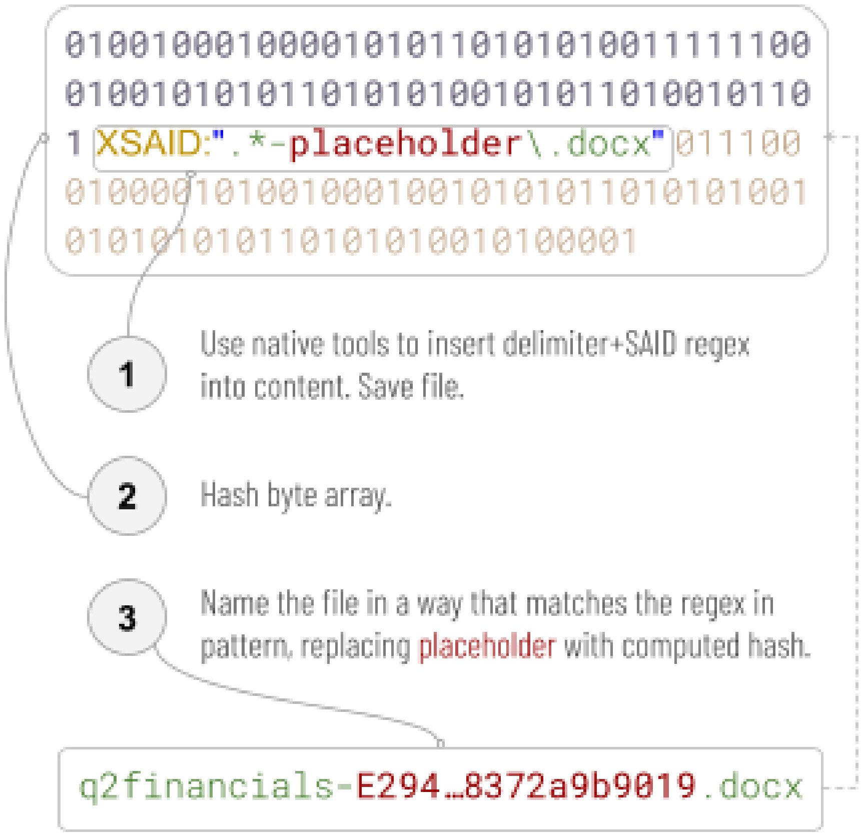
However, in certain cases a difficulty remains. Some file types are compressed, encrypted, or have their own internal integrity checks. If so, using native tools to insert a placeholder somewhere before a file is saved may be easy. However, using non-native tools to poke a SAID value into the byte stream later may be impossible. Either encryption/compression renders the delim+placeholder byte sequence unrecognizable, or an arbitrary change to the bytes of the file breaks the integrity checks and make the file invalid. Microsoft's `.docx` files (a `.zip` container holding XML documents, media assets, and metadata) exhibit this challenge because rewriting arbitrary bytes after save can invalidate internal structure unless container semantics are preserved. [15]

PDF files present a related difficulty for post-save bytewise modification. The PDF file format relies on cross-reference tables that record absolute byte offsets for objects, along with a trailer and a `startxref` pointer that locates the most recent cross-reference section. As a result, arbitrary byte insertions or replacements in an already-written PDF can invalidate object offsets unless the cross-reference and trailer structures are also updated—typically via a format-aware, incremental update process. Consequently, safely inserting new content into a PDF after save generally requires native or format-specific tooling rather than naïve byte patching. [16, 17]

For these cases, we take advantage of one additional location that is always an option for writing, without special tools: the filename. Writing the SAID there lets us saidify any opaquely structured file that can't be modified after it's written, by simply renaming it. This resembles an approach proposed for self-certifying pathnames in SFS, except that the input to the hash is content, where SFS's was a server and public key. [18]

This may seem like a futile exercise. The same flexibility that allows us to embed a SAID in a filename will allow someone else to remove the SAID. However, before saidifying, we can insert into the file content a regex to express a constraint on the file's name. This makes such a rename tamper-evident, much as an edit to a saidified JSON file is.

We call this (including minor enhancements described in 3.5 below) the *externalized SAID algorithm*, a SAD that uses it an *externalized SAD*, a SAID produced by it an *externalized SAID* (or *xSAID* for short), and the combination of delimiter+regex an *exsertion instruction*:


010010001000010101101010100111111100
01001010101101010100101011010010110
1 XSAID:".*-placeholder\.docx"011100
01000010100100010010101011010101001
0101010101101010100101000001
1 Use native tools to insert delimiter+SAID regex into content. Save file.
2 Hash byte array.
3 Name the file in a way that matches the regex in pattern, replacing placeholder with computed hash.
q2financials-E294…8372a9b9019.docx

This algorithm avoids any rewrite of the data structure after it's saved by its native tools, but it guarantees that a SAID is carried with the file wherever it goes, because any person or tool that sees the opaquely structured data in its decompressed/decrypted form can discover that a naming constraint exists on the file. If the file is (re)named improperly, it becomes an invalid match for the file's content. However, a proper name can be restored by renaming again in a way that conforms to the requirements in the regex.

Our placeholder is the same as with the bytewise algorithm: either a template or an actual SAID value of exactly the right length. As with the bytewise algorithm, the hash is always over a byte stream with a placeholder in template form. The delimiter changes a bit: instead of SAID, we have XSAID, to make it clear that the actual SAID value is externalized instead of appearing inline in the content. We also need quotes to contain the regex that contains the placeholder. We use the placeholder to break the regex into a pre-regex and a post-regex. The filename then must consist of anything that matches pre-regex, followed by the calculated SAID, followed by anything that matches post-regex. This allows the SAID to take up less than the full filename, preserving some of the normal naming flexibility so the manager of the file's container can also make human-friendly choices about the name, within constraints the content author sets.

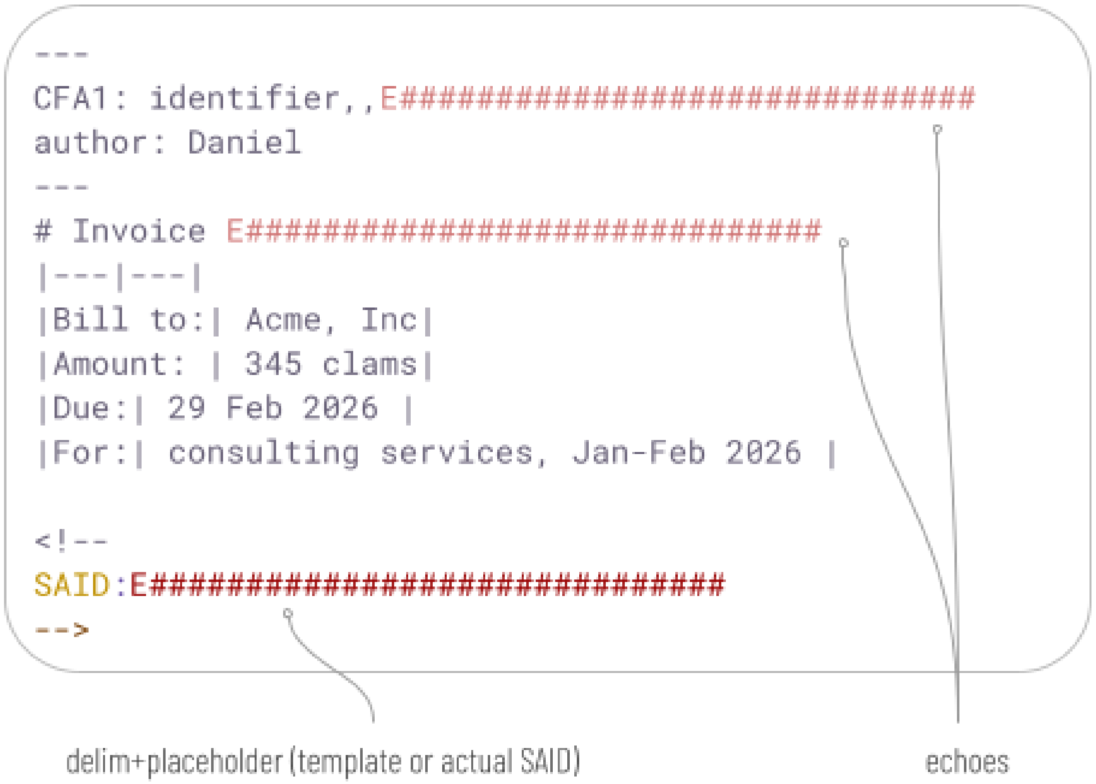
3.5 Multiple SAID references in a single file

The default saidification algorithm produces exactly one SAID for a given JSON object, but since objects can be nested, it is possible for a single JSON file to contain many SAIDs that roll up hierarchically. Saidification must proceed from greatest to least depth so the SAID of outer objects includes the SAIDs of inner objects.

Neither the bytewise algorithm nor the externalized algorithm can deal with the concept of nesting, since the internal structure of the file MUST be treated as opaque. However, we offer two enhancements that make the algorithms more flexible and convenient.

3.5.1 Bytewise echoes In the bytewise algorithm, the first (leftmost) occurrence of a valid insertion point defines the primary insertion point. All other occurrences of byte sequences that match the same placeholder pattern are treated as echoes. During SAID computation, the primary insertion point and all echoes are replaced with the placeholder in template form; once the SAID value is known, the calculated SAID replaces the placeholder at the primary insertion point and all echoes simultaneously.

For example, a markdown file could use echoes to display a SAID in YAML frontmatter and in a visible title, and place the less user-friendly delimiter+placeholder in an HTML comment:



```
---
CFA1: identifier,,E##############################
author: Daniel
---
# Invoice E##############################
|---|---|
|Bill to:| Acme, Inc|
|Amount: | 345 clams|
|Due:| 29 Feb 2026 |
|For:| consulting services, Jan-Feb 2026 |

<!--
SAID:E##############################
-->
```

3.5.2 Combined algorithms A single file may use both the bytewise and the externalized algorithms. If the bytewise algorithm is practical for a given file, the externalized algorithm is not necessary to communicate the SAID. However, it may be useful to bind the filename and file content together more strongly.

In such a case, the placeholders in the insertion point and the exsertion instruction are required to match. Given that constraint, both placeholders and all echoes are placed in template form. Then the bytewise algorithm is applied, causing the SAID to be calculated and the insertion point and all echoes (including the echo in the exsertion instruction) to be updated. Then the filename is updated as well.

3.6 Threat Model and Limitations

The techniques described in this paper are designed to provide tamper-evident identity for digital artifacts by binding a self-certifying identifier to a bytewise representation of content. They do not provide confidentiality, access control, or resistance to format-aware semantic transformations.

The threat model assumes that adversarial or accidental modifications to an artifact—such as byte insertion, deletion, rewriting, or unauthorized renaming—should be detectable by recomputing and checking the SAID. Any change to the byte stream after saidification necessarily produces a different SAID and is therefore treated as a new version of the artifact.

Workflows that intentionally rewrite content (for example, transcoding media, reserializing documents, normalizing metadata, or applying lossy compression) are out of scope for bytewise equivalence. In such cases, the resulting artifact is expected to carry a different SAID, reflecting a distinct identity rather than a failure of the mechanism.

Delimiter collisions within arbitrary byte streams are theoretically possible but can be made vanishingly unlikely by choosing delimiter patterns that are improbable in the target formats. This paper does not attempt to define canonical delimiter choices for all formats; instead, it specifies the syntactic requirements for safe detection and replacement.

Finally, these techniques assume that insertion points or exsertion instructions are introduced by the author at creation time, using native tooling or format-appropriate mechanisms. Artifacts that do not preserve these markers through ordinary handling cannot reliably retain a stable SAID under this model.

3.7 Implementation

A reference implementation in python of the bytewise and externalized SAID algorithms, as well as the standard saidification algorithm for JSON, is packaged as a bash script, `saidify`, at https://dhh1128.github.io/keri-tools. This implementation is intended as a reference implementation for conformance testing and illustrative purposes.

4. Summary

The SAD and SAID mechanism can be extended to arbitrary file types, including many that are commercially important and that do not support the standard saidification algorithm. Implementation is straightforward for readers (a generic byte scan suffices), and authors typically rely only on native tooling or libraries already used to create or edit the underlying format. We treat files as byte arrays and use one of two algorithms. The bytewise algorithm is appropriate for file types that can be rewritten after they are saved by native tools. It inserts a SAID at a location marked by a delimiter. The externalized algorithm works for file types that cannot be rewritten after they are saved. It saves a placeholder in the file and then externalizes the SAID to a filename in a way that conforms to the placeholder's requirements.

These techniques assume (1) that an author can introduce an insertion point or exsertion instruction into the artifact at creation time (typically via metadata or comments), and (2) that "sameness" is defined bytewise after saidification. If a workflow rewrites bytes nondeterministically (e.g., transcoding, reserialization, or metadata-stripping that alters the byte stream), the resulting artifact is simply a new version with a different SAID; this is consistent with the tamper-evident semantics that SAIDs are intended to provide.

Using these techniques, we can — without changing file formats or tools at all — allow existing, rich, ubiquitous file types to participate as first-class citizens in the verifiable data graphs enabled by SADs and SAIDs.

References

[1] Smith, S. 2024. Composable Event Streaming Representation (CESR). Trust Over IP Foundation. https://trustoverip.github.io/tswg-cesr-specification/.

[2] Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E.. 1986. Making data structures persistent. In Proceedings of the eighteenth annual ACM symposium on Theory of computing (STOC '86). Association for Computing Machinery, New York, NY, USA, 109–121. https://doi.org/10.1145/12130.12142.

[3] Miller, A., Hicks, M., Katz, J., and Shi, E. 2014. Authenticated data structures, generically. SIGPLAN Not. 49, 1, 411–423. https://doi.org/10.1145/2578855.2535851.

[4] Goodrich, M., Tamassia, R., Triandopoulos, N., and Cohen, R. 2003. Authenticated data structures for graph and geometric searching. In Cryptographers' Track at the RSA Conference (pp. 295-313). Springer, Berlin, Heidelberg.

[5] Tamassia, R. 2003. Authenticated Data Structures. In: G Di Battista, U Zwick (eds) Algorithms - ESA 2003. ESA 2003. Lecture Notes in Computer Science, vol 2832. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-39658-1_2

[6] Smith, S. and Gupta, V. Decentralized Autonomic Data (DAD) and the three R's of Key Management. 2018. In Rebooting Web of Trust Spring 2018. Available at https://bit.ly/3YIAqVj (accessed August 17, 2024).

[7] Sheff, I., Wang, X., Ni, H., van Renesse, R., Myers, A. 2019. Charlotte: Composable Authenticated Distributed Data Structures, Technical Report. arxiv.org.

[8] Smith, S. 2024. Authentic Chained Data Containers (ACDCs). Trust Over IP Foundation. https://trustoverip.github.io/tswg-acdc-specification/.

[9] Benet, J. 2014. IPFS - Content Addressed, Versioned, P2P File System. https://doi.org/10.48550/arXiv.1407.3561.

[10] Swarm: storage and communication infrastructure for a self-sovereign digital society. 2021. https://www.ethswarm.org/swarm-whitepaper.pdf.

[11] git-lfs. Git Large File Storage. https://git-lfs.com/.

[12] Josefsson, S. 2006. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. IETF. https://doi.org/10.17487/RFC4648

[13] Dublin Core Metadata Initiative. 2024. Dublin Core Metadata Element Set, Version 1.1. http://dublincore.org/documents/dces/.

[14] Oren, B. K., Evans, C., and I döt Net. 2009. YAML Ain't Markup Language (YAML™) Version 1.2. https://yaml.org/spec/1.2/spec.html.

[15] ISO and IEC. 2016. ISO/IEC 29500-1:2016. Information technology — Document description and processing languages — Office Open XML File Formats — Part 1: Fundamentals and Markup Language Reference. https://www.iso.org/standard/71691.html.

[16] International Organization for Standardization. 2008. *ISO 32000-1:2008 — Document management — Portable document format — Part 1: PDF 1.7*. ISO. Available at https://www.iso.org/standard/51502.html

[17] Adobe Systems Incorporated. 2006. *PDF Reference, Sixth Edition: Adobe Portable Document Format Version 1.7*. Adobe Systems Incorporated. Available at https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/pdfreference1.0.pdf

[18] Mazières, D., Kaashoek, M. F. Sep 1998. Escaping the Evils of Centralized Control with self-certifying pathnames. Proceedings of the 8th ACM SIGOPS European workshop: Support for composing distributed applications. Sintra, Portugal: MIT. https://dl.acm.org/doi/pdf/10.1145/319195.319213.