Bytewise and Externalized SAIDs

Daniel Hardman
daniel.hardman@gmail.com
August 2024

Abstract — Self-addressing identifiers (SAIDs) help build decentralized, authenticatable graphs of data with useful caching properties. However, the CESR spec only imagines SAIDs for data structures that can be serialized in a canonical way, then modified after serialization without side effects. This paper proposes two new algorithms that allow arbitrary files of any format to be saidified. Using this approach, nearly any file type can participate fully in authenticated data graphs, without format upgrades or special tooling.

Keywords — KERI, CESR, information theory, verifiable data, authenticated data structures, cryptographic identifiers, decentralized identifiers, content-addressable

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 its SAID is present, the correspondence between a SAD and its SAID can be checked, making the data structure versioned and authenticatable. [2, 3, 4] Any change to the SAID 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). [7]

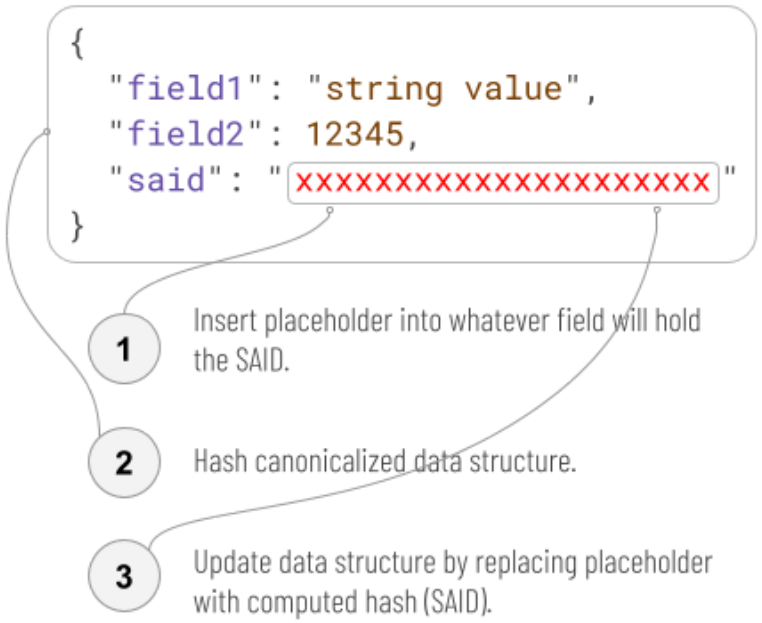

Figure 1: CESR's saidification algorithm

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.

These data formats 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 formats like these 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 for the placeholder (which must be unique enough that it is unlikely to occur naturally):

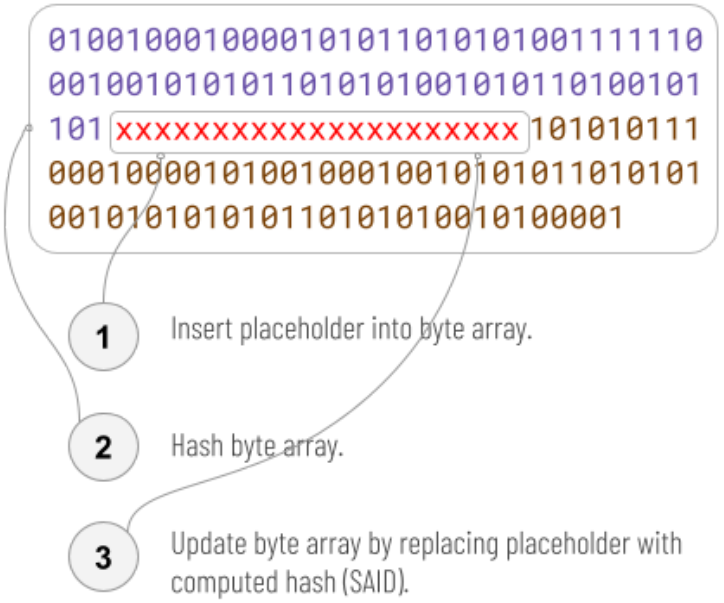

Figure 2: naive algorithm for a SAID in a byte array

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:

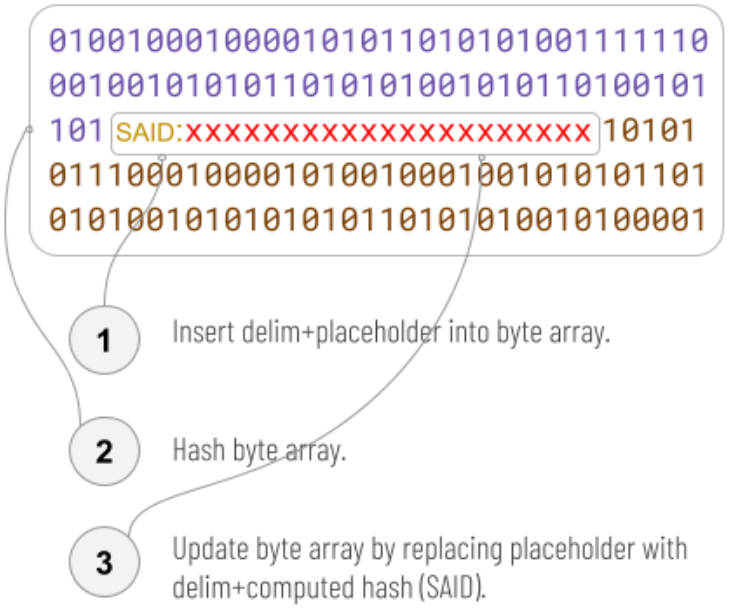

Figure 3: byte array with delimiter + SAID

We call this modified algorithm the ***bytewise SAID algorithm***, a SAD that uses it a ***bytewise SAD***, and a SAID produced by it a ***bytewise SAID***.

In Figure 3, we show the delimiter as the simple prefix "SAID:". We could make this delimiter more exotic if needed. Regardless of the details, we can now find the SAID after insertion by searching the bytes for the delimiter followed by a recognizable SAID in CESR (self-describing, typed, length-encoded) format.

3.3 Inserting the SAID

So far, our solution has glossed over where and how we insert the placeholder — and later, the actual SAID value. We said that opaquely structured filetypes tend to expose internals through libraries or APIs. Such APIs are unlikely to allow arbitrary access to a byte array, and even if they did, won't we have trouble deciding where it's safe to make changes?

The answer to this question is usually uncomplicated: our algorithm depends on a logical location, not an exact offset. We insert the SAID wherever its 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 insert the SAID as arbitrary metadata.

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:

- 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.)
- 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.
- 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). [8] 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 [9]). 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 of a folder that includes the main document content plus attached graphics and metadata) use an opaquely structured data format that has this challenge. [10]

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 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 the ***externalized SAID algorithm***, a SAD that uses it an ***externalized SAD***, and a SAID produced by it an ***externalized SAID***:

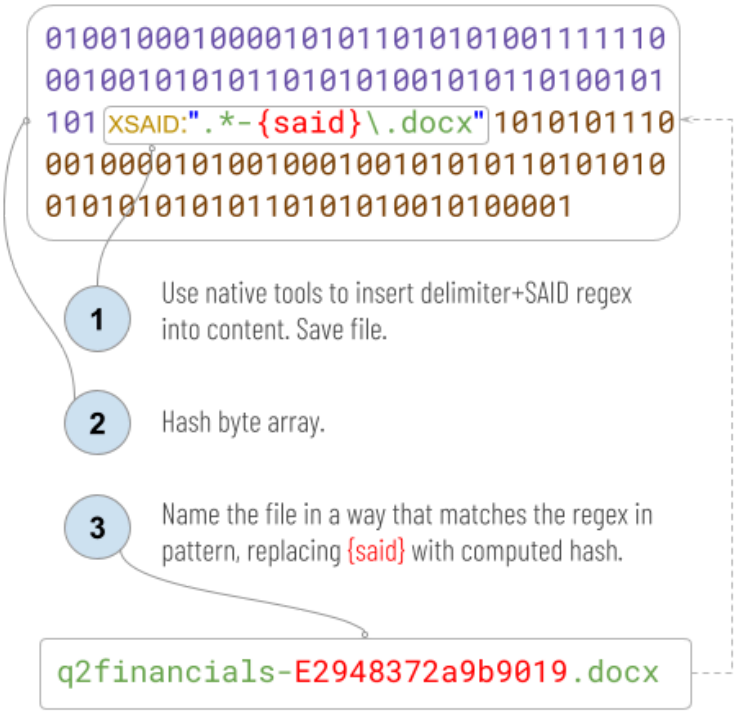

Figure 4: byte array with delimiter + SAID regex

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 from 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 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 need to differentiate, since our validation algorithm changes in a corresponding way. We also need quotes to contain the regex that follows the placeholder. The regex 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.

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 easy, and does not require specialized tooling. 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.

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

[2] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. 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] A Miller, M Hicks, J Katz, and E Shi. 2014. *Authenticated data structures, generically*. SIGPLAN Not. 49, 1, 411–423. https://doi.org/10.1145/2578855.2535851.

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

[5] R Tamassia. 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] S Smith and V Gupta. *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] S Smith. 2024. *Authentic Chained Data Containers* (ACDCs). Trust Over IP Foundation. https://trustoverip.github.io/tswg-acdc-specification/.

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

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

[10] 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.