SADs and SAIDs for Opaquely Structured Data

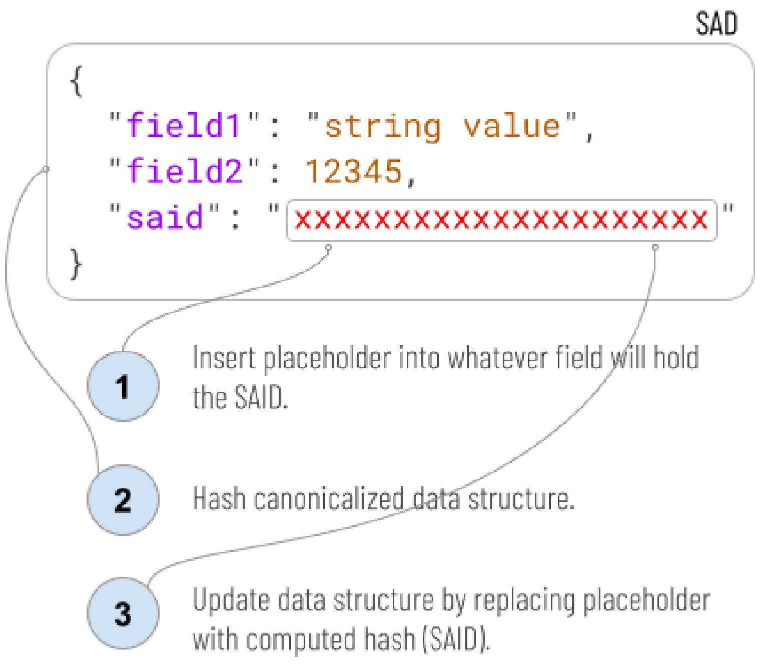
Daniel Hardman — June 2023

Basic SADs and SAIDs

SADs (self-addressing data) and SAIDs (self-addressing identifiers) are innovations that allow decentralized graphs of data with verifiable integrity and useful caching properties. SAIDs can be tested against their corresponding SAD to detect tampering, and SAIDs can replace SADs anywhere that the bytes of the SAD are unnecessary or separately fetchable. Structuring data containers as SADs and then referencing them via their SAID is the core idea of ACDCs. It allows credentials and other assertions to be shared securely, efficiently, and with graduated disclosure.

But what if we want to securely reference, share, and cache data that isn't structured the way ACDCs expect? The world is full of data like this (consider YouTube's video catalog, or the set of documents on your company's SharePoint, or your personal collection of digital photos, for example). It would be nice if we could build an interoperability bridge between it and KERI/ACDC features.

Usually, the relationship between SAD and SAID is described as one of a *data structure* to a *field*. The SAD is the data structure, and the SAID is a field within that structure, reserved and populated *ex post facto* with a hash of the rest of the structure:


SAD
{
"field1": "string value",
"field2": 12345,
"said": "xxxxxxxxxxxxxxxxxxxxxxx"
}
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).

This fits nicely with arbitrarily extensible, structured data formats important to early work on KERI: JSON, CBOR, and MsgPack. All of these formats are transparent serializations of data structures that support straightforward saidification.

Opaquely Structured Data

However, many data formats are relatively *opaque* about their internal structure. This does not mean they lack structure; rather, it means that the level of abstraction at which it is most convenient to interact with them is higher-level than *data structure* with *fields*. Examples include: office and desktop publishing documents, spreadsheets, databases, photos, videos, compiled software binaries, web pages, medical data, CAD models, 3D printing recipes, genomes, and so forth.

Such formats *do* embody data structures — often, dramatically sophisticated ones — but unlike JSON, CBOR, and MsgPack, 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."

The *data-structure-with-fields* approach to saidification doesn't map easily to opaquely structured data. Most of these formats limit extension, or impose a lot of constraints on it. Custom fields in an opaquely structured format may be invisible to or even stripped by handling software. Even if we could solve the mechanics of a robust SAID field extension for a given format, 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. And the set of opaquely structured data formats grows constantly.

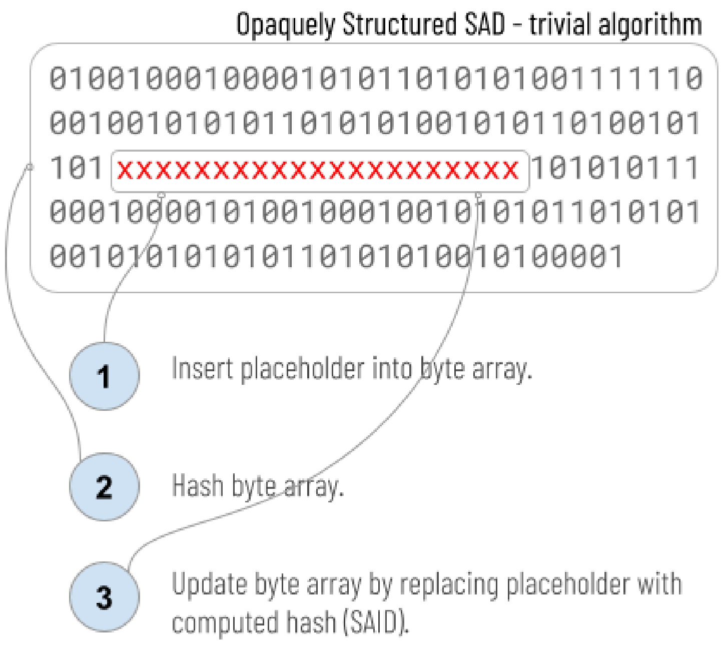
Despite these challenges, opaquely structured data formats are incredibly important and ubiquitous. Having the ability to Include them in decentralized graphs of verifiable data is highly desirable. We want a way to reference them as SADs, so we can cache and integrity-proof them in the same way we do with the serialization formats where SADs and SAIDs began.

Solution

Part 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 viewed another way, as a 3-field data structure: bytes-before, SAID, bytes-after). This completely eliminates canonicalization problems

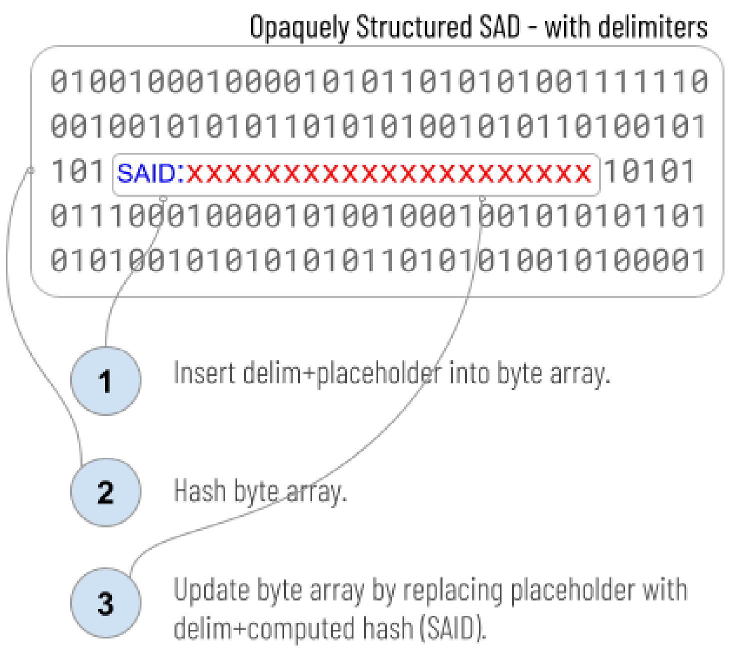
— any bytewise difference is significant — and it gives us an easy and wholly generic, robust saidification algorithm. Rather than traversing structure to find a field in the data, we simply scan for the placeholder (which must be unique enough that it is unlikely to occur naturally):


Opaquely Structured SAD - trivial algorithm
0100100010000101011010101001111110
0010010101011010101001010110100101
101 XXXXXXXXXXXXXXXXXXXXXX 101010111
0001000010100100010010101011010101
00101010101011010101001010100001
1 Insert placeholder into byte array.
2 Hash byte array.
3 Update byte array by replacing placeholder with computed hash (SAID).

Part 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*, since the SAID is an unpredictable byte sequence. The saidified file might already have bytes that look like a SAID...

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


Opaquely Structured SAD - with delimiters
0100100010000101011010101001111110
0010010101011010101001010110100101
101 SAID:XXXXXXXXXXXXXXXXXXXXXXXX 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).

Here, we show the delimiter as the simple prefix "SAID:", but we could make this delimiter more elaborate if we decide something less likely to occur naturally is needed. Regardless of the details, we can now find the SAID after insertion by simply searching the bytes for the predictable delimiter followed by a recognizable SAID in CESR (self-describing, typed, length-encoded) format.

Where and how we insert 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 from part 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, dumb 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). 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). 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 I had software for recording recipes, one per file artifact, and it lacked metadata support, I could simply put a note at the end of every recipe's instructions: "(SAID: <my SAID value>)".

Part 4: externalized SAIDs

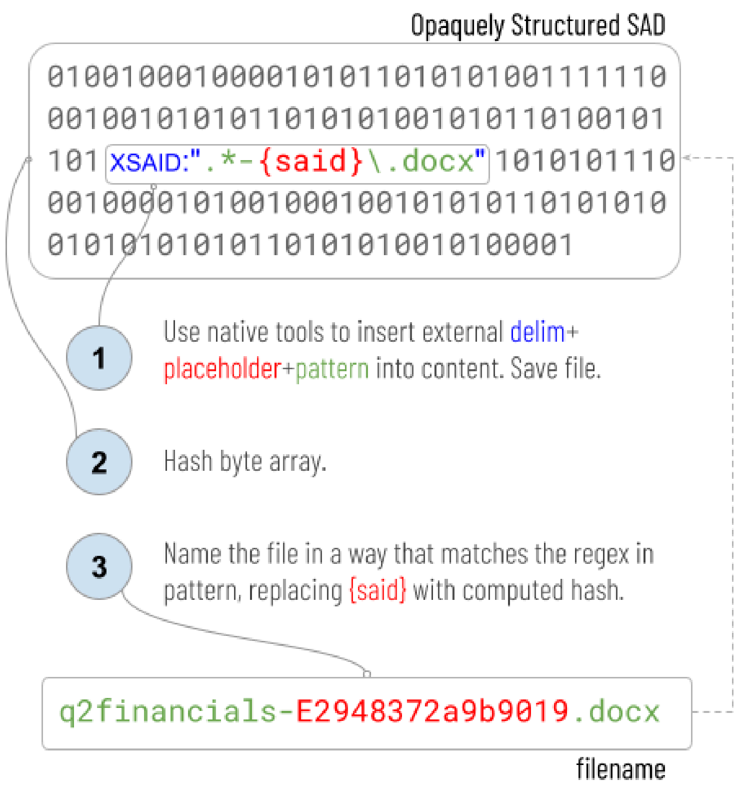
For many opaquely structured data formats, the preceding algorithm with delimiters is sufficient. If it is, we should use it.

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, but modifying the file later, with non-native tools, by poking a SAID value into the byte stream, 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) is a good example of an opaquely structured data format that has this constraint.

For these cases, we take advantage of one additional location that is always an option for writing, without special tools: *the filename itself*.

It may seem like putting a SAID in a filename is a futile exercise. Won't the same flexibility that allows us to embed a SAID in a filename allow someone else to remove the SAID?

Yes. However, we can insert *into the file content* a regex to express a constraint that makes such an edit detectable, in the same way that an edit to a saidified JSON file is detectable. Here's how it works:


Opaquely Structured SAD
0100100010000101011010101001111110
0010010101011010101001010110100101
101 XSAID:".*-{said}\.docx" 1010101110
0010000101001000100101010110101010
0101010101011010100010100001
1
Use native tools to insert external delim+ placeholder+pattern into content. Save file.
2
Hash byte array.
3
Name the file in a way that matches the regex in pattern, replacing {said} with computed hash.
q2financials-E2948372a9b9019.docx
filename

If we follow this final externalized SAID algorithm, we can saidify any opaquely structured file that can't be modified after it's written, by simply updating its name.

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.

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 first is appropriate for file types that can be rewritten after they are saved by native tools. It inserts a SAID at a location found by searching for a delimiter. The second 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. They can be rendered tamper-evident, referenced in ACDCs, and cached just like data structures built from JSON, CBOR, and MsgPack.