Elision, Redaction, and Noncorrelation in Smart Documents

Authors: Wolf McNally, Christopher Allen, Shannon Appelcline, Blockchain Commons**Revised:** Aug 23, 2022**Status:** DRAFT

Introduction

Blockchain Commons has been doing research and development in the area of creating structures that can represent "smart documents." Such stuctures can facilitate many uses without requiring design from scratch. Furthermore, multiple capabilities can be composed together in a single document, including:

- Encryption
- Signing
- Sharding (M of N share recovery)
- Elision (redaction)
- Selective disclosure
- Merkle tree verification
- Efficient binary encoding

Our reference work in progress is a structure simply called `Envelope` that is currently implemented in Swift as a proof-of-concept architecture. `Envelope` is part of a capability suite called Secure Components. Documentation for Secure Components and `Envelope` specifically can be found here.

Secure Components has a number of goals:

- Make it easy to represent common encryption design patterns (such as sign-then-encrypt and encrypt-then-sign) based on algorithms that are considered best practices.
- Represent structured data using efficient binary formats such as CBOR.
- Interoperate with structures of particular interest to blockchain and cryptocurrency developers, such as seeds and HD keys.
- Support protocols such as Distributed Identifiers (DIDs).
- Support innovative constructs such as Sharded Secret Key Reconstruction (SSKR).
- Support complex metadata (assertions about assertions).
- Support semantic knowledge graphs.
- Support mutable and immutable architectures.
- Allow for the future extension of functionality to include additional cryptographic algorithms and methods.
- Provide a reference API implementation in Swift that is easy to use and hard to abuse.

This document will introduce the `Envelope` type and its approaches to noncorrelation, data minimization, and selctive disclosure.

Envelope

The Envelope type efficiently supports everything from enclosing basic plaintext messages to innumerable recursive permutations of encryption, signing, sharding, and representing semantic graphs.

Here is its notional structure in Swift:

```
struct Envelope {
    let subject: Envelope
    let assertions: [Assertion]
}
```

Assertions combine two specific parts, `subject` and `predicate`, both of which themselves are also of type `Envelope`:

```
struct Assertion {
    let predicate: Envelope
    let object: Envelope
}
```

This assertion-predicate-object triplet may be more easily understood by considering its linguistic usage:

> "Alice (subject) knows (predicate) Bob (object)."

Note that there can be many assertions associated with each subject:

> "Alice (subject) knows (predicate) Bob (object) and dislikes (predicate) Carol (object)."

In "Envelope notation" this would be written:

```
"Alice" [
    "knows": "Bob"
    "dislikes": "Carol"
]
```

Plaintext Subject: A Simple Example

Despite the recursive hierarchy of `Envelope`, its simplest usage just encloses a plaintext message, which may be any CBOR-encodable structure. Here we just use a UTF-8 string:

```
"Hello."
```

> *An `Envelope` with a plaintext `subject` ("Hello") and no assertions.*

Signature: An Example

From here, assertions such as signatures can be added:

```
"Hello." [
    verifiedBy: Signature
]
```

The subject "Hello" is verified by cryptographic signature.

This is an `Envelope` with the same `subject` ("Hello") and a single assertion, where the `predicate` is `verifiedBy` and the `object` is a cryptographic `Signature`.

The reader is referred to the Secure Components documentation for a discussion of the specific cryptographic algorithms that `Envelope` uses.

Encryption: An Example

Continuing the above example, the `subject` could be encrypted:

```
EncryptedMessage [
    verifiedBy: Signature
]
```

An encrypted message is verified by cryptographic signature.

The important thing here is that even though the `subject` has now been encrypted, the signature can still be verified. This is possible because each element in an `Envelope`, including the `subject`, carries its own cryptographic digest, and that certain tranformations of the envelope, including encryption and elision, *do not change* this digest. Because of this property, these sorts of transformations also do not invalidate upper-level digests of the `Envelope` as a whole.

Therefore an `Envelope` is a Merkle tree that may be transformed in certain ways without invalidating it.

Public Key Encryption: An Example

Continuing the above example, we can add an assertion containing the symmetric key used to encrypt the subject, itself encrypted to the public key of a recipient:

```
EncryptedMessage [
    verifiedBy: Signature
    hasRecipient: SealedMessage
]
```

An encryption is verified by cryptographic signature and that may be decrypted by a recipient who controls a specific private key.

Adding the second assertion `hasRecipient` *does* invalidate (modify) the digest of the envelope as a whole, but because the encrypted `subject` and the `verifiedBy: Signature` assertion have not changed relative to each other, the signature can *still* be verified.

The novel structure of `Envelope` opens up numerous applications that take advantage of this fact.

Elision

The term *elide* means "to leave out." In an `Envelope`, elided elements are replaced by their Merkle tree digest, therefore allowing the same digests to be calculated for the entire tree despite those absences.

Elision vs. Redaction

The term *redaction* is often used interchangably with *elision*. There is, however, a crucial semantic difference between the two terms: elision is *what* is accomplished, while *redaction* is one purpose for which data might be elided.

In fact, when eliding data there are at least two major goals that can be accomplished:

- **Redaction.** When one *redacts*, one is choosing to withhold information with no affordance or expectation that the receiving party can or will recover it.
- **Referencing.** When one *references*, one withholds information with every affordance and expectation that the receiving party can and might choose to recover the elided information.

For example, my photo might be embedded as a JPG within a verifiable credential. Embedding has the advantage that it's right there to be interpreted by anyone who receives the document. In this case, there are two reasons I might want a version of my credential with the photo elided:

First, I might *redact* it because it is priviledged or irrelevant to the transaction I want to perform.

Alternatively, I might want the data to be smaller, while still allowing the retrieval of the photo by interested parties. In this case, a *dereferencing method* would need to be included that shows *how* to retrieve the information. This method can either be built into the verifiable credential by the issuer, or can be added later by the holder as an additional assertion on the whole document. Either way, anyone who retrieves the photo can absolutely know it is correct because it exactly matches the elided element's digest in the document's signed Merkle tree. In this sense, elision and unelision are *isomorphic* and *reversible*.

Encryption might be seen as a special kind of elision, where the original message is still present as ciphertext, and may only be "unelided" (unencrypted) using the proper key. In this way, encryption is also isomorphic and reversible, and even interchangeable with elision.

Elision via Target Set

The structure of `Envelope` makes it easy to choose to remove a "target set" of digests in the Merkle tree. More commonly, the target set is composed of digests to *reveal*: in other words, elision hides everything that is not specifically revealed in the target.

To understand the creation of a target set, it's useful to keep this schematic in mind:

```
envelope {
    subject [
        assertion1(predicate1: object1)
        assertion2(predicate2: object2)
        ...
        assertionN(predicateN: objectN)
    ]
}
```

... where all of the symbols shown above (e.g. `envelope`, `subject`, `assertion1`, and `predicate2`) are called *positions*. Each position is *itself* an `Envelope`, and therefore may carry its own assertions, recursively. Each position also carries a digest, which may be used to refer to the element at that position.

The reader is referred to the Secure Components documentation on elision for fully-worked examples.

Elision: An Abstract Example

Here we'll simply show what it looks like when each separate part of an example `Envelope` is elided:

```
"Alice" [
    "knows": "Bob"
]
```

If the `envelope` digest is elided, the resulting envelope is just the top-level digest of the Merkle tree:

```
ELIDED
```

If just the `subject` is elided:

```
ELIDED [
    "knows": "Bob"
]
```

If just the `predicate` of the assertion is elided:

```
"Alice" [
    ELIDED: "Bob"
]
```

If just the `object` of the assertion is elided:

```
"Alice" [
    "knows": ELIDED
]
```

If the digest corresponding to the `assertion` as a whole is elided:

```
"Alice" [
    ELIDED
]
```

We note again that all the parts above that may be separately elided, may alternately be separtely encrypted with the same guarantees on preserving the Merkle tree.

Elision: A Practical Example

An `Envelope`-based document representing a verifiable credential might look like this:

```
{
    CID(4676635a6e6068c2ef3ffd8ff726dd401fd341036e920f136a1d8af5e829496d) [
        "certificateNumber": "123-456-789"
        "continuingEducationUnits": 1.5
        "expirationDate": 2028-01-01
        "firstName": "James"
        "issueDate": 2020-01-01
        "lastName": "Maxwell"
        "photo": "This is James Maxwell's photo."
        "professionalDevelopmentHours": 15
        "subject": "RF and Microwave Engineering"
        "topics": CBOR
        controller: "Example Electrical Engineering Board"
        isA: "Certificate of Completion"
        issuer: "Example Electrical Engineering Board"
    ]
} [
    note: "Signed by Example Electrical Engineering Board"
    verifiedBy: Signature
]
```

> *The envelope contains info on identifier with THESE characteristics, and this envelope is verified by a signature from the Electrical Engineering Board.*

Using elision to power selective disclosure, the employer of an employee with this credential could warrant to a third-party that their employee has such a

credential, without revealing anything else about the employee. This is done as follows:

1. Elide privileged information from the certificate, while keeping only necessary information.
2. Enclose the envelope in another envelope to which the employer adds its own assertions that need to be non-repudiable; this is the employer's *warranty*.
3. Enclose the warranty in another envelope, which is then signed by the employer.

```
{
    {
        {
            CID(4676635a6e6068c2ef3ffd8ff726dd401fd341036e920f136a1d8af5e829496d) [
                "expirationDate": 2028-01-01
                "firstName": "James"
                "lastName": "Maxwell"
                "subject": "RF and Microwave Engineering"
                isA: "Certificate of Completion"
                issuer: "Example Electrical Engineering Board"
                ELIDED (7)
            ]
        } [
            note: "Signed by Example Electrical Engineering Board"
            verifiedBy: Signature
        ]
    } [
        "employeeHiredDate": 2022-01-01
        "employeeStatus": "active"
    ]
} [
    note: "Signed by Employer Corp."
    verifiedBy: Signature
]
```

The envelope contains info on identifier with LIMITED characteristics, as verified by a signature from the Electrical Engineering Board, with hiring information for the identified individual verified by a signature from Employer Corp.

Creating or validating this complex document only takes a few lines of code.

Note that in this example the elision was performed by the employer, who possesses a copy of the `Envelope`-based certificate, but who is neither the holder of the certificate (the employee), not the original issuer (the certification board). The ability for anyone to perform elision on any document is one of the major advantages of elision over more complex methodologies such as Zero-Knowledge

Proofs.

Noncorrelation

Bit sequences are said to be *correlatable* if by examining them there is any way to determine whether they are projections of (produced from) the same image (source). If there is no practical way to learn whether a set of sequences are projections of a common image, or even likely to be, they are said to be *noncorrelatable*.

Different sorts of structures may be correlatable or noncorrelatable by design. For instance, in Secure Components, a `Signature` is noncorrelatable because each signature is constructed using entropy. Therefore two signatures produced from the exact same image will nonetheless bear no resemblance to each other or the image that was signed, or the key used to sign them.

On the other hand, a `Digest` is correlatable by design. Two digests produced from the same image will always be exactly the same.

Finally some structures are said to be *quasicorrelatable* if they bear a resemblance in size to the image from which they were produced. For example, an `EncryptedMessage`, even though it is produced using entropy and is therefore noncorrelatable at the bit level, is nonetheless a fixed number of bytes larger than than the plaintext, and is therefore quasicorrelatable.

To strengthen the security of `Envelope`, work may sometimes need to be done to reduce correlation. An optional assertion containing random data may be used to do so (see "Decorrelation" below.)

Uses and Liabilities of Correlation

Sometimes correlatability is desired and necessary. For instance, when eliding some assertions from the `Envelope` while still allowing verification of its signatures.

Other times, correlatability is not particularly desired, such as with `EncryptedMessage` where "known plaintext attacks" must be countered by the use of entropy in the form of a nonce.

A simple string, like "Hello", when elided twice, produces the exact same digest each time, and is therefore correlatable. Comparing two envelopes from which the same string has been elided might provide clues as to the original string, thus creating an undesired correlation.

Decorrelation

Decorrelation is any technique used to decrease the possibility of correlating two projections as having a common source image.

A common technique for decorrelation is *salting*, with the typical use-case being passwords that have been salted then hashed. In the event the password database is compromised, even two users with the same password cannot be identified by the attacker as having such, because their hashed passwords each use a different salt.

Salting is sufficient for passwords because they are short. However, additional considerations need to be undertaken when smaller messages may often come from a smaller or fixed vocabulary, or when larger messages may be quasicorrelated due to their size.

Consider this example envelope:

```
"Alpha" [
    note: "Beta"
]
```

"Alpha" has the note "Beta."

If we elide the `subject`, `predicate`, and `object` separately, we get:

```
ELIDED [
    ELIDED: ELIDED
]
```

This can still be attacked via brute force, because the elided elements are short, human-readable strings, and the predicate `note` is from the family of "known predicates" and therefore always produces a well-known digest.

The `Envelope` reference API has a built-in function `addSalt()` that is intended to address undesired correlatability. When constructing an `Envelope`, `addSalt()` adds an assertion to an envelope, the object of which contains a variable amount of random data:

- For small elements, the number of bytes added will generally be from 8...16.
- For larger elements the number of bytes added will generally be from 5%...25% of the size of the element.
- The `addSalt()` function also allows a specific number of salt bytes be added as padding, for situations where decorrelation might be performed by making a series of variable-size messages more similar or equal in size.

So a simple `Envelope` containing a text string that has been salted would look like this:

```
"Hello" [
    salt: Data
]
```

"Hello", which has been salted.

When this envelope is elided, the top-level digest is guaranteed to bear no resemblance to the digest of the original, unsalted envelope.

Returning to the example above, every part of an envelope can be individually salted as needed. In this case, each of the `subject`, `predicate`, and `object` have been salted:

```
{
    "Alpha" [
        salt: Data
    ]
} [
    note [
        salt: Data
    ]
    : "Beta." [
        salt: Data
    ]
]
```

> *The subject "Alpha" has been salted, and has an assertion whose predicate* ***`note`*** *has been salted, and whose object "Beta" has been salted.*

The elided version of this envelope looks just like the previous one, but its digests bear no resemblance to the fixed digests produced without salting:

```
ELIDED [
    ELIDED: ELIDED
]
```

For More Information

See our Crypto Envelope Presentation for more on the fundamentals of envelopes; and our Envelope Privacy Requirements Presentation for more on elision and non-correlation. Also see our Envelope docs, especially the sections on Noncorrelation and Elision and redaction.

You may also want to look at our poster, which overviews the topics of this talk.

We are currently holding regular biweekly meetings on the development of the `Envelope` specification. If you are interested in using `Envelope` for the escrow or recovery of cryptographic secrets and metadata, please contact us.