ELUV.IO

Content Security Model Overview

Introduction - Blockchain-controlled Content and Trustless Content Security

The present digital content supply chain depends on multiple trusted technological systems -- systems that are relied upon to safeguard content from being stolen or tampered with by unauthorized parties -- as content passes from owner to consumer. The failure of any of these trusted systems puts the content at risk of being stolen, tampered with, or counterfeited. Content is protected by storing it in a 'trusted storage system' and providing access to the content uses a 'trusted authentication system' (which typically consists largely of a 'trusted application' running on top of a 'trusted database'). Authorization to access content for viewing uses a 'trusted rights management system' which itself may consist of many layers and variations of technology depending on the generation and type of protection ranging from DRM server software, encryption software and hardware, and watermarking software. All of these subsystems rely on 'trusted operating systems' and computer hardware.

Each of these systems is typically 'gated' meaning that the 'trust' relies in the fact that these systems and networks are only accessible to their administrators and the organization or the group who runs them, and the group that manufacture them, and these systems are protected from the outside world through firewalls (which themselves must be 'trusted'), and yet, recent events have shown that even the lowest common denominator of commoditized server software could be tampered with even before reaching these gated environments and become untrustworthy[1][2]

The paradigm of engineering sufficiently "trustworthy" systems is becoming increasingly difficult to sustain successfully as more and more content flows over the Internet to ever more variations of rights management, ever more points of vulnerability exist in the increasingly complex technological supply chain, and the value of digital content and incentive to steal grows. Fortunately, there is an entirely new paradigm now possible and with innovation, a "trustless" content Internet is possible.

Specifically a blockchain (decentralized ledger) offers an entirely new paradigm for protecting and controlling access to content. Most importantly it allows content to be stored on and distributed by systems and networks that are not 'gated', and not centrally managed by an organization and group, and not "trusted" -- (i.e. a decentralized system). Transactions around valuable resources require that the two parties participating in the transaction can trust that the other party is authentic -- who they claim to be --

---

[1] "The Big Hack: How China Used a Tiny Chip to Infiltrate US Companies - Bloomberg." 4 Oct. 2018, https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies. While the existence of this particular hack is disputed, the energy around the story illustrates that a compomise of this nature is now possible given the moderrn IT supply chain.

[2] "New Evidence of Hacked Supermicro Hardware Found in ... - Bloomberg." 9 Oct. 2018, https://www.bloomberg.com/news/articles/2018-10-09/new-evidence-of-hacked-supermicro-hardware-found-in-u-s-telecom. Accessed 9 Oct. 2018.

and that the value offered in the transaction is not being spent elsewhere. In today's financial systems, and in today's digital content access systems, these two points of trust are verified by going through one or more "trusted" third party entities. In the digital content supply chain this has led to the current state of vulnerability as these "trusted" parties fail intentionally or unintentionally, and has led to centralization of control in the supply chain around a few large companies that have been able to scale the delivery channel to the consumer (i.e. companies that have dominant Internet infrastructure).

Blockchain ledgers decentralize this process by systematically distributing trust over a network of actors that are incentivized economically to behave in a "trustworthy" manner by carrying out a common protocol. The protocol uses public key cryptography to verify the authenticity of the participants in a transaction and uses a decentralized consensus mechanism whereby all participants have the same view -- the ledger -- recording the network's state. Participants are incentivized to further the consensus ledger by earning value for extending it with valid transactions, and vice versa, the consensus mechanism ensures that it is impractically difficult for a bad actor to tamper with previous transactions in the ledger or add invalid transactions as long as a majority of participants are well-intentioned. There are many variations of the consensus protocol details and what "majority" means but this essential idea underpins all blockchains.

Implementation Details for Content Access Control via the Blockchain Ledger

The Content Fabric builds upon these features of decentralized blockchain ledgers by backing all content access control -- operations to create, update, or access content -- with blockchain transactions executed through a native ledger embedded in the Content Fabric software stack. The system ensures that all parties are authentic, and its consensus ensures that only valid (authorized) transactions on the content can be carried out. Unlike all other blockchain associated content management/distribution systems to date, the Content Fabric intrinsically couples control over the content's modification and access to the blockchain, while maintaining scalable storage and distribution outside of the blockchain. All other systems to date defer the implementation of content access control and distribution to external/3rd party systems running separately from the blockchain ledger, or attempt to store content in the blockchain and are thus not scalable.

The Content Fabric design allows content access control and authorization to take advantage of a key feature of blockchain ledgers for "programmable" transactions between parties. Each transaction on the blockchain can execute a small piece of code that represents the terms of access for each content object (asset). This small piece of code is referred to as a Smart Contract. (The current Content Fabric implements a blockchain that is compatible with the Etherum protocol and exposes an Ethereum Virtual Machine interface for applications, including support fo Ethereum Smart Contracts.)

The 'ledger' is charged with three essential functions:

- Providing the authoritative 'directory' of all content including the only trusted reference to the list of versions of each content object and the final verification 'hash' (the ground truth) for each of these versions
- Execution of the 'access control' logic allowing users to read and write content as well as
    - Contract terms enforcement
    - Commercial terms reconciliation (payments and credits)
- Recording all access operations on content in the fabric (the "ledger" function)

The blockchain is essentially an ordered list of transactions. Each transaction is performed by a blockchain participant and could have side effects: a state change in a particular account or contract, a transfer of value, or one or more blockchain 'events'. Transactions are identified by a 'transaction ID' and the content of a transaction as well as the 'transaction receipt' are available to all blockchain participants. Because the ledger is public, transactions typically don't store private or secret information - they mostly store 'proofs' of the activities they are recording, for example in the fabric, the final 'checksum' of a new content after an update. The way in which transactions can offer public verification of a particular action without revealing the details of the action belongs to a class of cryptography called Zero Knowledge Proofs.

Actors

All participants in the blockchain fall into these two categories:

- Account owners
- Contracts

Account owners are primarily people in control of their actions against the blockchain (for example creation or update of content, accessing or viewing content, etc).

Applications operated by people or automated processes are also account owners. These applications are trusted by the people who run them to do what they were constructed to do and they are trusted to operate the blockchain accounts they have been given access to.

On the other hand Contracts are entirely autonomous participants - they operate exclusively based on their 'code'. For example a Contract written to pay 2 credits to each user who supplies a particular record signed by a signature accepted by this Contract, will forever behave the same way and pay the 2 credits when the signature is matched, and decline to pay otherwise.

An "account owner" is identified by its 'address' on the blockchain and owns a public / private key pair that it uses to sign its transactions against the blockchain.

A contract is identified by its 'address'. The contract will only have an address if it has been successfully deployed by its creator. The creator is known because the creation of the contract is done in a 'transaction' where the 'from' address is the contract creator.

Definition of Key Concepts

The fabric implementation uses a few key concepts to scope the mechanisms by which content is managed:

- Content Space - sets the base policies controlling access to all associated content objects
    - Represented by the Smart Contract (Content Space Contract)
    - The Content Fabric can largely operate as a single, global Content Space but additional Content Spaces can be created for special purpose use
- Fabric Nodes
    - Each node has a blockchain account, represented by its public/private key pair

- Fabric Users
    - Each user has a blockchain account represented by its public/private key pair
- Library
    - A repository of Content, setting the policies for how all of its Content objects work
    - Created inside a Content Space
    - Represented by a Smart Contract (Library Smart Contract) which is determined by the containing Content Space
    - Has a “Fabric User” as an owner
- Content
    - The direct representation of a digital asset
    - Created in a Library
    - Represented by a Smart Contract (Content Smart Contract) which is determined by the containing Library
    - Each Content object has an instance of the Content Smart Contract

Fabric users can act in various roles

- Library Owners
    - The owner of a library can dictate the behavior of Content objects inside the library (for example who can create Content, who owns the Content once created, how Content is accessed or commercialized)
- Content Owners
    - The owner of a Content object controls reading and writing access to the Content
    - A Content object can have multiple owners and they might have slightly different privileges (for example modifying or updating the content object versus publishing the content object for consumer access and determining commercial terms)
    - The degree of control over the Content is set by the Library
- Consumers
    - Users who don’t own content and can access content based on the Content objects contract terms (including commercial terms such as access or viewing charge)

Logical flow of a Content Object Lifecycle through the Ledger

As shown in Figure 18, all operations on content objects in the fabric follow a two part flow of first invoking the appropriate blockchain transaction on the appropriate contract address, and then using the proof of a valid transaction -- indicating that the operation is authorized -- to make an authorized API call on the Content Fabric. Specifically, the client of the fabric API securely obtains his/her public/private keys, and creates a blockchain transaction signed with the private key. On successful completion of the transaction (which can perform any necessary authorization logic), the client creates an authorization token including the transaction ID, and passes this token in the corresponding Content Fabric API. Optionally, the fabric API may require the client to call a finalization method on the contract in order to complete the API transaction.

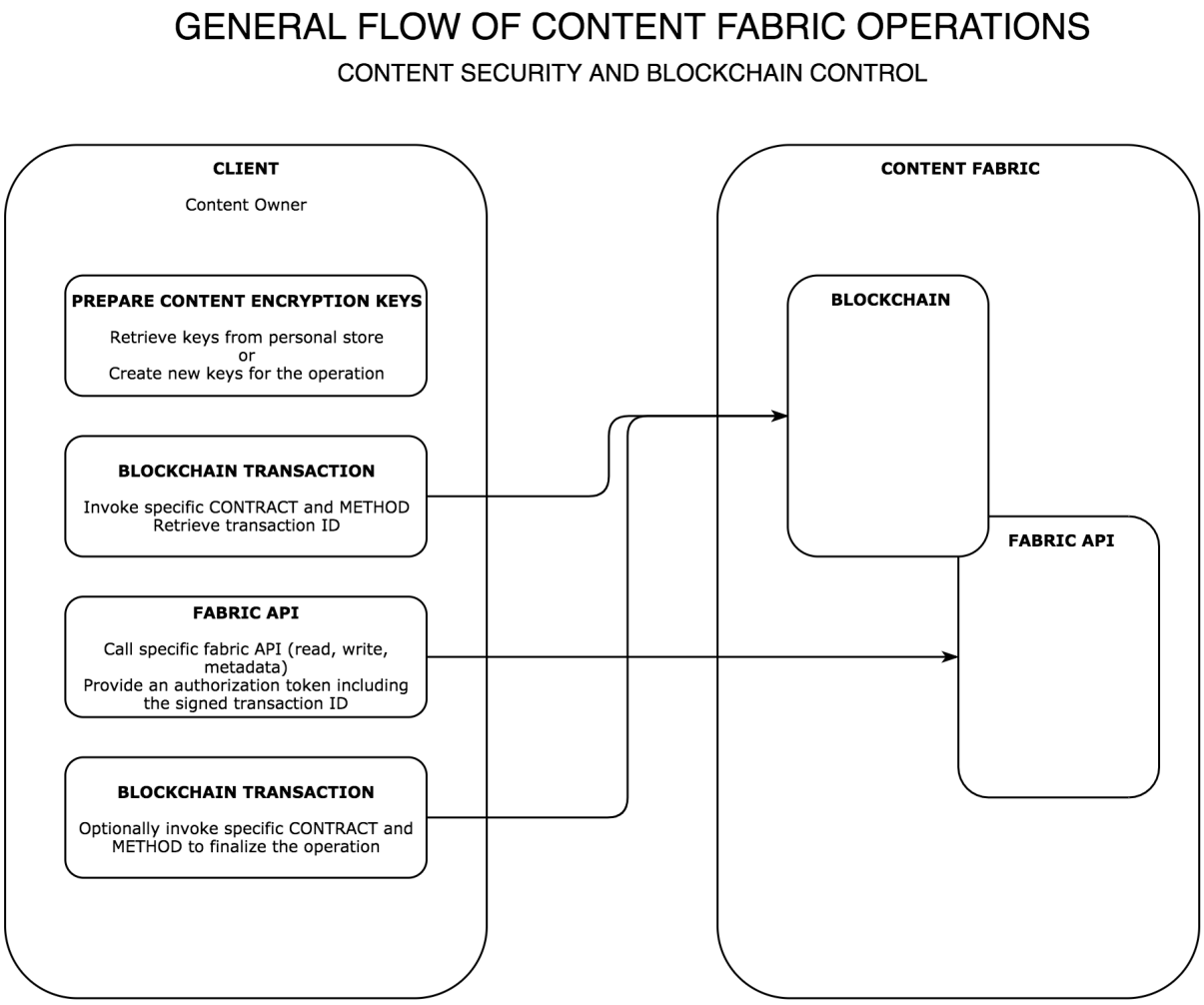

Figure 18. General process in the Fabric for securing content access control and privacy

Detailed Security Model for Publishing and Consuming Content

Content Space and Content Library Creation

Publishing a content object into the Content Fabric assumes a Content Library has been created and that Library exists within a Content Space. A Content Library is created within a Content Space and as such requires that the Content Space gives the user permission to do so. For example the global Content Space will allow all users to Create Libraries for a fee. The Content Space is assumed to be created by the originator of the Content Fabric but additional Content Spaces can also be created by participants in the fabric for special use (such as for private or semi-private subsets of the Content Fabric with dedicated private nodes).

Because of their genesis roles, Content Spaces need to be trusted by the Fabric Nodes, and Nodes are configured to trust Content Spaces by their maintainers. A new Content Space is created by deploying the Content Space Contract and configuring Fabric Nodes to recognize the new Space. (New spaces are rare and this process is executed by specialized operations teams).

To create a Library, an end user or client program makes a simple API call directly, or via a user interface. The API implementation executes a method on the Content Space contract 'CreateLibrary()' which in turn will create a new instance of the Library Smart Contract for that particular Library based on the parameters specified. The calling user becomes the owner of the Library Contract and as such will be able to further configure the Library Contract.

Content Creation

The creation of a new content object and updates to existing objects involves the Content Owner ("Alice") (or her application) carrying out the following steps:

1. Calling a method on the Library Contract to look up the possible content types and their required security groups offered by the Library
2. Calling CreateContent on the corresponding Library Contract passing the content type and the chosen security groups. This returns a transaction ID and a new Content ID.
3. Calling CreateContent on the Content Fabric passing in an authorization token containing the transaction ID obtained above and the Content ID, signed by the content creator. This returns from the Content Fabric a valid write token.
4. Generating the Content Encryption Key Set: an AES symmetric key and a proxy re-encryption public, private key pair (AFGH) for each security group.
5. Encrypting the Content Encryption Key Set for three parties, the Owner, the fabric's Key Management Service and any Owner Delegates, using their respective public keys
6. Calling the fabric method to SetSecurityMetadata to store this data mapped to these three entities
7. Uploading content parts to the fabric, encrypting each Part with the AES content key first, and then using the AFGH key
8. Calling Finalize on the Library Contract passing in the 'content hash', which is a new version of the content, uniquely identified by this hash. [Note that each content update API invocation returns the potential 'content hash' if a version of the content were to be finalized without further modifications.]
9. Calling Finalize on the Content Fabric passing an authorization token including the Transaction ID obtained above and signed by the Owner (the creator).

Finally, the Node supporting the operation calls the Confirm method on the Library Contract passing in the final content hash, and signing the transaction with the node's key to prove that this node supported the operation.

Updating content in the fabric follows an analogous process, except we skip steps 1 - 3 and instead call WriteAccessRequest() on the Content Contract to authorize the update, and then OpenWrite on the fabric using the obtained transaction ID in the authorization ID. Steps 4-9 are the same.

Content Consumption

The consumption of content from the fabric by the Content Consumer ("Bob") involves the Consumer (or his application) carrying out the following steps:

1. Creating an Ephemeral Key Set consisting of a proxy re-encryption public/private key (AFGH) encrypted with the public key of the consumer
2. Encrypting the Ephemeral Key Set for the fabric's Key Management Service using its public key
3. Calling AccessRequest() on the Content Object's Blockchain Contract passing in the encrypted Ephemeral Key Set

4. The contract records the Ephemeral Key Set in the Contract's state using a unique key, escrows any value required by the AccessRequest from the Consumer's credit, and returns the transaction ID
5. Consumer calls ContentOpen on the Fabric passing in an authorization token containing the transaction ID obtained above, signed by the Consumer
6. The Node calls the KMS (the delegate) passing in the authorization token above (including the Transaction ID and signed by the Consumer) for the authorized AccessRequest()
7. The KMS verifies the request by verifying the Transaction ID (Consumer's signature and success status) and then generates
    a. (a) the proxy re-encryption key using the AFGH key in the Ephemeral Key Set, and returns the re-encryption key to the Node
    b. (b) an encrypted version of the content AES key, encrypted with the Consumer's public key
8. The KMS calls AccessGranted() on the Content Contract recording the re-encryption key and the Content Key encrypted for the Consumer, obtained above. The contract releases the "value" from escrow to the Content Owner.
9. The Node uses the re-encryption key to re-encrypt the content from the original AFGH key space into the consumer's ephemeral AFGH key space.
10. The Consumer reads the re-encryption content delivered by the node and the encrypted key blob recorded by the AccessGranted() contract method. The consumer is now able to decrypt the content as follows:
    a. Extract the AES content decryption key from the encrypted key blob using its private key
    b. First decrypt using its ephemeral AFGH secret key
    c. Then decrypt the result using the AES content decryption key obtained above
11. When required or beneficial, Consumer calls the Content Contract's AccessComplete() method

Proxy Re-encryption Overview

The re-encryption of content published by the Owner for an authorized Consumer occurs without the fabric software or host computer it runs on having any access to the plain text Content or to the AES Content Encryption Key, allowing for a "trustless" re-encryption of the content. This capability utilizes very recent advances in an area of cryptography known as proxy re-encryption. Proxy re-encryption is a form of public / private key cryptography that allows data encrypted with one user's public key to be transformed such that it can be decrypted with another user's private key. The re-encryption transformation is 'permissioned' in the sense that it is only possible when the original, encrypting user generates a re-encryption key that is based on their private key and the public key of the target user. The re-encryption key itself is protected and does not expose useful information about the original encrypted data. Further, this re-encryption key is used by a third party - a proxy - to re-encrypt the data without it ever becoming unencrypted.

Proxy re-encryption therefore performs an extremely useful and powerful function for a secure (trustless) content management system to prevent unintentional or intentional unauthorized access to content; specifically, it allows for secure, encrypted data to be easily shared with other users without exposing valuable private keys with any intermediary technology or allowing a malicious end-user to trivially share keys that could be used to decrypt (and steal) other content in the system. To date there is no large scale content management system that provides for such trustless content protection.

Proxy Re-encryption in the Fabric's Content Security

Unlike traditional content management systems the Content Fabric is designed to run in a distributed, trustless environment. Security assumptions are different in a trustless environment; in particular, it is not valid to entirely delegate content security to the content nodes themselves. The Content Fabric achieves strong content security through a two-layer encryption strategy that merges more traditional encryption with proxy re-encryption.

As detailed in the previous section, in the Content Fabric, secure data is published encrypted with a set of keys generated and stored by the content publisher. Two distinct sets of keys are generated for each piece of content by default: a symmetric key and a public / private key pair for proxy re-encryption. The symmetric key and algorithm is currently AES-128. The Fabric's proxy re-encryption is implemented with pairing-based cryptography so a pairing-friendly elliptic curve must be used. The Fabric currently uses the curve BLS12-381, which provides the desired security level with good performance.

While each form of cryptography independently provides strong security guarantees, the two distinct key sets are used to effect the trustless model. The symmetric content keys are managed securely in the fabric. The keys for proxy re-encryption are managed by a separate, independent online system (a Key Management Service). When a user is granted access to a content object the symmetric key is transmitted directly to the authorized user. This key by itself is insufficient to decrypt the data and no useful information can be recovered with just this key. To perform the proxy re-encryption the authorized user generates their own set of BLS12-381 keys. The system creates a re-encryption key based on the original content owner's key and the content-specific key of the end user. This key is then transmitted to one or more nodes in the Content Fabric. These nodes then proxy the encrypted data, using the provided re-encryption key to transform the data in realtime into the target key of the end user. The end user then decrypts first with their private key and then the symmetric key. This form of two-tier encryption guarantees that the keys the end-user controls cannot be used to directly decrypt the original source data that is stored in the fabric.

Key generation, storage and management are all performed automatically and transparently on behalf of both the content owner and content consumers. All encryption and decryption is performed in real-time while data is stored and retrieved from the fabric. While the Fabric's two-tier approach adds some additional overhead, performance is assured with optimal implementation of the underlying algorithms. A highly efficient, scalable library is used for all server-side processing and the same library is cross-compiled into Web Assembly (WASM) to execute in client software, including all modern browsers. Performance testing has shown that the WASM-based client decryption is fast enough to support secure, high-resolution streaming video.

Detailed Encryption and Key Management Flows

Content Creation & Publishing

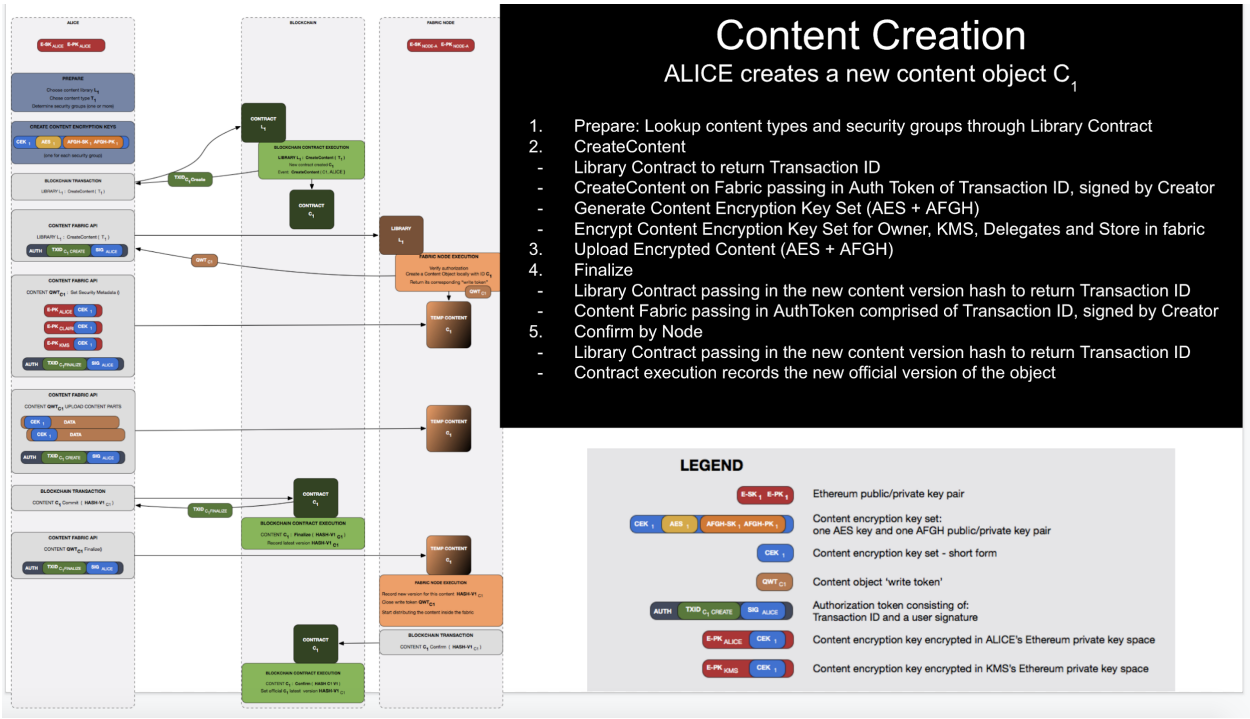

Figure 1. Content Creation Key Flow

Content Creation Implementation Details

The sequence of security operations to securely publish content to the Fabric is shown in Figure 1, and the implementation details described in steps 1-9 below. The Eluvio Content Fabric provides two client software implementations, elv-client-js (JavaScript source code), and qfab_cli (Go source code). The Content Fabric has functionally distinct daemons for blockchain functions (elvmasterd) and content/data functions (qfab). The implementation details will refer to these services by name.

1. Preparation (Dark Grey)

- Client creates an Ethereum ECIES public/private key pair E-SK1, E-PK1 using standard Ethereum secp256k1 crypto primitives, implemented in elv-client-js using ethers.js https://github.com/ethers-io/ethers.js/ and implemented in the qfab_cli using go ethereum https://github.com/ethereum/go-ethereum All crypto libraries are compiled from c source code to WASM (elv-client-js) and using c-go interfaces for qfab_cli.
- Client creates a Content Encryption Key Capsule (CEK1). CEK1 consists of the following keys:
    - An AES128 symmetric private key for the content (AES1), implemented using the milagro open source libraries (https://github.com/apache/incubator-milagro-crypto-c/blob/develop/src/aes.c). generated using a secure random number generator also provided by milagro (https://github.com/apache/incubator-milagro-crypto-c/blob/develop/src/rand.c) The AES encryption uses AES GCM mode with a 12 byte random initialization vector (per 1 MB encryption block), implemented using the milagro secure random number generator.

    - An AFGH key pair (AFGH-SK1, AFGH-PK1), which uses BLS12-381 elliptic curve, implemented using https://github.com/apache/incubator-milagro-crypto-c/blob/develop/src/rom_curve_BLS381.c . The key is 32 bytes and has no separate initialization vector; as described below, the padding is provided by the first level AES encryption of the content, itself random.
- Client calls a method on the Library Contract to look up the content types supported

2. Publishing of content starts by the Client calling the CreateContent method on the Library Contract (L1) (via elvmasterd), passing in the content type and security group, which creates a transaction on the Fabric blockchain, creating a new ContentObject (C1) and returns a transaction ID (TxID C1 Create) and the new Content Object's ID. The blockchain implementation follows the standard Ethereum contract-based state transition https://ethereum.org/en/whitepaper/#ethereum-state-transition-function
3. This is followed by the client calling CreateContent on the Content Fabric data layer (qfab) passing in an authorization token containing the transaction ID obtained above and the Content ID, signed by the client using standard ECDSA signatures as implemented in the open source ethereum client libraries https://github.com/ethers-io/ethers.js/ and https://github.com/ethereum/go-ethereum . The signature is seeded with a cryptographically secure random number generator provided in these libraries.
4. This returns from the Content Fabric a valid write token (QWT C1)
5. The Client then encrypts the Content Encryption Key Set (CEK1) for the Owner using the Owner ECIES key (E-PK alice), and optionally for the Fabric's Key Management Service using the KMS ECIES key (E-PK KMS) (for later re-sharing via reencryption), and for any Owner Delegates, and calls the Fabric method to SetSecurityMetadata to store this data mapped to these three entities in the Fabric. The Client transmits the encrypted capsules to the Fabric using standard HTTPS (min TLS v1.1) for storage as encrypted parts in the Fabric (and referenced by the private, authorized metadata of the Content Object).
6. Then the Client encrypts and uploads each content part using the elements in the CEK1. The Client first encrypts the part with the AES content key (AES1) and then uses the result as input to the AFGH encryption (AFGH1), and finally uploads the parts to the Fabric (qfab), repeating this process for the totality of parts. The encryption of each part occurs on the client host before the part is uploaded, and uses AES GCM (per 1 MB encryption block), and then AGFH, with details as described in step 1. Encrypted parts are transmitted from the Client to the Fabric node over a standard HTTPS encrypted channel (min TLS v1.1)
7. Once all parts are uploaded, the Client calls Finalize on the Library Contract passing in the 'content hash', which is a new version of the content, uniquely identified by this hash (Hash-v1 C1). This hash is calculated using a Merke proof of the part hashes comprising. [Note that each content update API invocation returns the potential 'content hash' if a version of the content were to be finalized without further modifications.]
8. Calling Finalize on the Fabric (qfab) passing an authorization token including the Transaction ID obtained above and signed by the Owner (the creator) (Tx ID C1 finalize).
9. Finally, the Node supporting the operation confirms that all parts have been successfully published by calling the Confirm method on the Library Contract, passing in the final content hash, and signing the transaction with the node's key to prove that this node supported the operation, using an ECDSA signature as described in 3. The node receives back a transaction id confirming the transaction.

Content Access and Read

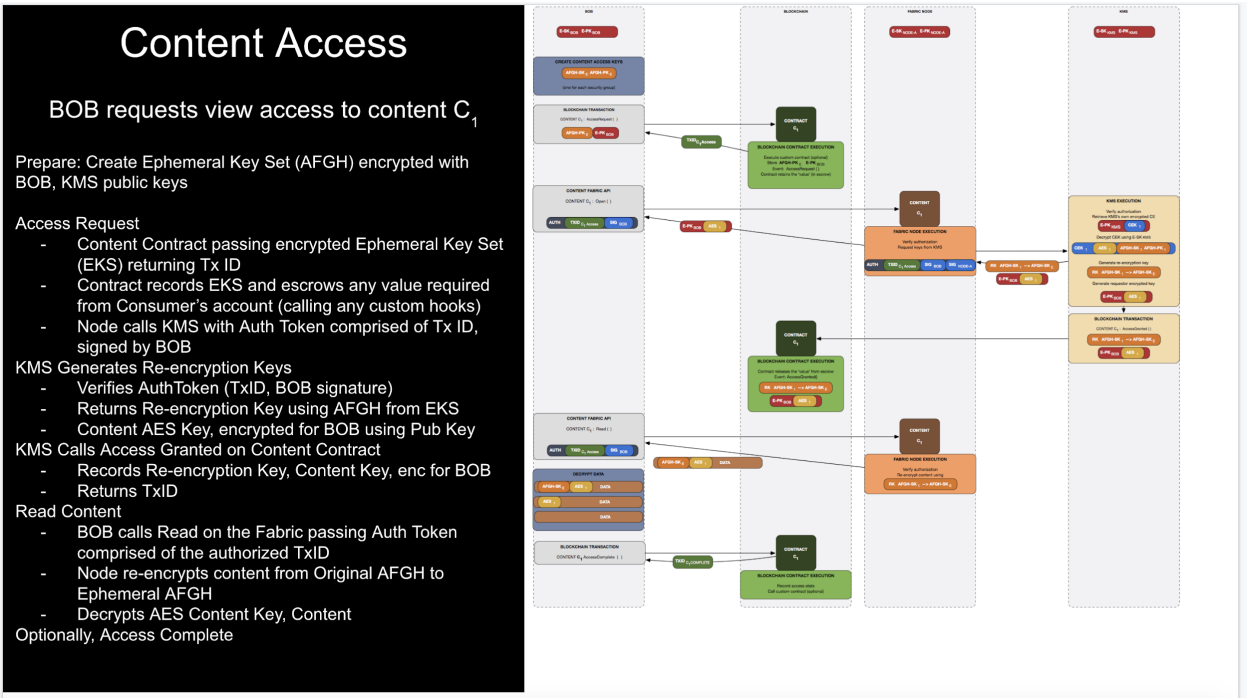

Figure 2. Content Access Key Flow

Content Access Implementation Details

The sequence of security operations to securely publish content to the Fabric is shown in Figure 1, and the implementation details described in steps 1-9 below.

1. Preparation (Dark Grey)

- The accessing Client creates the target AFGH key pair, which we will call the Ephemeral Key Set (AFGH-SKE, AFGH-PKE) for re-encryption using the BLS12-381 elliptic curve implementation at https://github.com/apache/incubator-milagro-crypto-c/blob/develop/src/rom_curve_BLS381.c
- For the 32 byte private key, the implementation uses a key derivation function (KDF) with as input the user's private Ethereum Key (E-SK2, E-PK2) and the unique Content ID.

2. Access Request
    a. The Client makes an AccessRequest call on the Content Object contract to get a transaction id for access (TxID C1Access), which will be used to authorize transmission of the public AFGH key (AFGH-PKE) to the Fabric.
    b. The Client makes an Open request on the Fabric (via qfab on the egress Node) passing the authorization token (TxID C1Access) signed by the Client (ECDSA) and AFGH-PKE, transmitted over HTTPS (TLS v1.1 min)
    c. The egress Node (via qfab) calls on the KMS (via elvmasterd) passing the authorization token signed by the client (ECDSA), API parameters, and token signed by the Node (ECDSA), over HTTPS. As in Content Publishing, the ECDSA signature implementation uses the open source ethereum client libraries https://github.com/ethers-io/ethers.js/ and https://github.com/ethereum/go-ethereum seeded with a cryptographically secure random number generator provided in these libraries.

3. KMS Generates Re-encryption Key
    a. Using AFGH proxy re-encryption as described in https://eprint.iacr.org/2005/028.pdf and the milagro implementation of BLS12-381 as described above, the KMS returns a re-encryption key to qfab on the egress Node (RK AFGH-SK1 -> AFGH-SKE). The KMS also calls AccessGranted on the Content Object contract.
4. Egress Node Re-encrypts
    a. The egress Node (via qfab) performs the re-encryption of the Content Object using the re-encryption key (RK AFGH-SK1 -> AFGH-SKE).
5. Client Reads and Decrypts the Data
    a. The Client makes an API request to the KMS (via elvmasterd) passing in the authorization token (TxID C1Access) also signed by the Client (ECDSA), to retrieve the symmetric content key (AES1).
    b. Client calls Read on the Fabric for data (via qfab on the egress Node) for one or more parts. Part-by-part, the Node re-encrypts the part data using the AFGH client-specific re-encryption key as described in https://eprint.iacr.org/2005/028.pdf and transmits that part to the Client over HTTPS.The Client decrypts each re-encrypted part using the ephemeral AFGH key (AFGH-SKE) and then decrypts the content using the AES key (AES1).

Content Integrity Verification and Origin Traceability

The Content Fabric's unique architecture and security model allows for creating a first-ever blockchain-verifiable (tamper proof) content versioning system that can both provide integrity verification of content and a traceable history of content version changes. This yields clear technology benefits over previous distributed file systems, cloud storage, and version control solutions, where content integrity and version integrity is only as good as the trusted solution, and potentially provides a foundational platform for building "trustworthy" content systems and media solutions to combat today's societal challenges over the lack of traceability of content and counterfeit content. We detail the technology we have created for fast proofs of content version integrity and the recording of content version history into the blockchain for traceability.

The World Wide Web has standardized on a host to host security framework - TLS (previously known as SSL), the foundation of HTTPS (secure HTTP). A client connects to a server and through TLS it creates a secure channel that along with the Certification Authority system allows the client to trust that:

a. The server is authentic, i.e. it is talking to the intended server
b. All communication is private, even though it travels through a potentially large number of intervening devices
c. All communication has integrity, i.e. data is not modified in any way by any intervening steps

As shown previously, the Content Fabric security model provides for authenticity of the parties, and privacy of the content. This leaves (c), the integrity of the content, to be addressed. Managing content in the fabric is more complex and differs from the host to host communication addressed in TLS in three ways:

1. Content is created and updated by multiple users, not just one
2. Content lives in the Content Fabric for a long time
3. Content is accessed by many users, not just one

Thus an effective solution must serve all three requirements without degrading user experience (fast, transparently). Recall that a content object is structured as a reference tree where the nodes in the tree refer to the hashes of the constituent parts (see ***content structure section***). The Content Fabric uses a novel fast proof algorithm that allows any client reading an object to verify the hashes of the object's constituent parts in a short time (at most log(n) download and hash operations where n is the number of parts,) and therefore allows a client to verify the integrity of a content object it has read.

The algorithm works as follows:

1. Each content part is broken down in smaller segments - 1, 2 or even up to 10 - 20 MB in size (ideally a size that allows clients do download it in 1-2 seconds)
    a. Each segment is 'hashed' using, by default, SHA-256 (configurable to include upcoming SHA-3 standards)
    b. All segment hashes are organized in a '**Merkle tree**' [3] (https://en.wikipedia.org/wiki/Merkle_tree) [Note: the existing implementation uses a variation of the Merkle tree called a Patricia tree, optimized for the specific use case of the Content Fabric.]
    c. This tree has the following properties:
        i. The root of the Merkle is a hash that reflects every single change in the content - so a given Merkle tree root fully identifies a particular version of the content
        ii. For any given segment, we can calculate what we call the "Merkle proof" that allows a user in possession of the segment to ascertain that the segment is in entirety correct and it resolves to the known root of the Merkle tree. This proof is a list of hashes of all adjacent branches of the tree up to but not including the root hash. The list is of size $\log_n$ (tree height)
2. All content parts are listed in a special metadata store in Common Binary Object Representation (CBOR) format[4]. The user can retrieve the CBOR data blob and verify that (a) the root of the merkle tree for the desired part is present and (b) that the hash of the CBOR data blob is further resolved correctly toward the content version hash as described below.
3. All metadata stores are similarly stored in a CBOR data blob and can be verified the same as #2 above.
4. The content object "reference store" is a CBOR data blob containing the hash of the content parts data blob #2 and metadata #3.

Using this structure, a user in possession of any part of the content object (data or metadata) can very quickly verify that this data is correct in entirety and resolves to the known hash of the content object version (see Figures 19 and 20).

The 'content object version' is recorded in the blockchain upon creation of the content object and update of the content object. As described in the previous section, the content object version is recorded in the Commit transaction following any write to content. This transaction encodes the address of the account that performed the write operation and in combination with the object proof, can prove unambiguously the blockchain account, and therefore the actor, responsible for a content operation. (See Figure 21 for the use of the "From" and "To" addresses in an example, block and transaction within it.) Applying this

[3] Merkle Tree - Wikipedia article, https://en.wikipedia.org/wiki/Merkle_tree
[4] CBOR - Wikipedia article. https://en.wikipedia.org/wiki/CBOR

capability across the functioning of the fabric allows for a first-ever, transparent, provable chain of record for content.

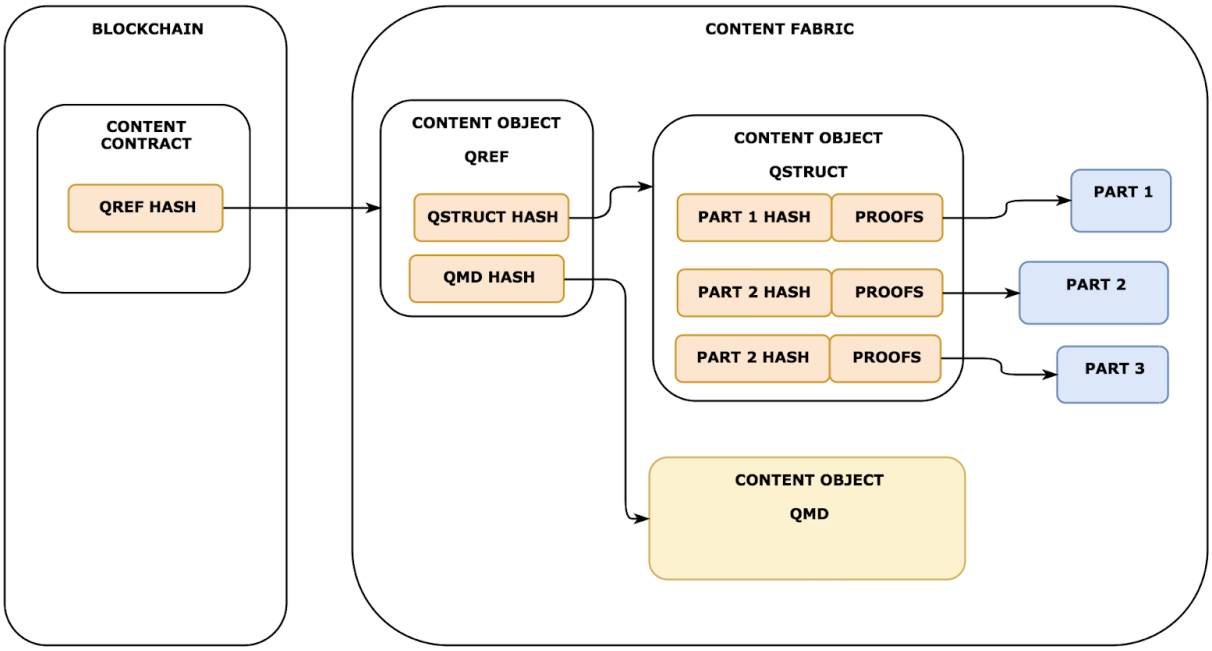

Figure 19. Content object version proof mechanism with record in the blockchain.

Example content object verification tree

Content Verification Info - VERIFIED!


```
{
 "hash": "hq__QmQedd9AuQSdoBjMzNUVp7BaqWRXAJiJGAdcLXLWWKy
 "qref": {
  "valid": true,
  "hash": "hqp_QmQedd9AuQSdoBjMzNUVp7BaqWRXAJiJGAdcLXLWWK
 },
 "qmd": {
  "valid": true,
  "hash": "hqp_QmU4wn795KbSGvTH2uki6UwWT1xbv1yqDCM7bEdweYi
  "check": {
   "valid": true,
   "invalidValues": []
  }
 },
 "qstruct": {
  "valid": true,
  "hash": "hqp_QmSHqXe9cr5EtgsYcg7dP9irZ4MHsTFo4Rc5e4hh8oHfPT
  "parts": [
   {
    "hash": "hqp_QmUv75ETf9x22cjNNbTncVPMuKZpXLPHF7tVz2D9ACt
    "proofs": {
     "rootHash": "acda1564dfc542064b7becdbc2d107c84fe1f942337bf
     "chunkSize": 49,
     "chunkNum": 49,
     "chunkLen": 50,
     "finalized": 116
    },
    "size": 2634
   },
   {
```


Content Verification Info - FAILED!


```
{
 "hash": "hq__QmZ7oxvpECVgvED7QwMHgmKZGpEEoHkRhH
 "qref": {
  "valid": true,
  "hash": "hqp_QmZ7oxvpECVgvED7QwMHgmKZGpEEoHkRh
 },
 "qmd": {
  "valid": true,
  "hash": "hqp_QmPaddxjFwCc9TPAg652MSRoV9iada8gjB1F
  "check": {
   "valid": true,
   "invalidValues": []
  }
 },
 "qstruct": {
  "valid": false,
  "error": "Hashes do not match",
  "hash": "hqp_QmUfPJ4ehcbi17fH26QvwY6ViFhACaMsENyv
 },
 "valid": false
}
```

Figure 20. Examples of successful and failed verification of the content object proof API in a client application.

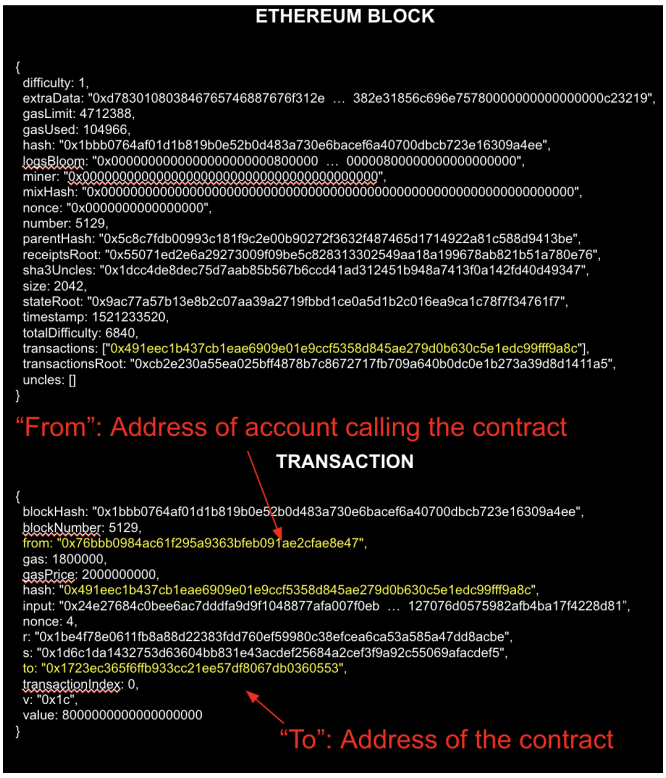

Figure 21. Example blockchain transaction. Note the use of the From and To fields recording the addresses of the entities that made the transaction. The content version proof hash is recorded in the transaction and the address details provide a tamper-proof record of who is responsible for an object version.

Example content part proofs

Given a content part with the root hash `"6bfc296a266462eb34c1fbad68d40a7f9805f0d4d6c2b75b0a5664fd5e844da2"` and a segment size of 10MB, the following proof (Figure 22) is constructed such that if a user is in possession of any of the 10MB segments, the entire hash tree can be calculated up to the 'root_hash'. (Please see the Appendix for a complete version of this content proof.)


```
{
  "root_hash": "6bfc296a266462eb34c1fbad68d40a7f9805f0d4d6c2b75b0a5664fd5e844da2",
  "proofs": [
    {
      "byte_beg": 0,
      "byte_end": 10485759,
      "proof": [
        "6bfc296a266462eb34c1fbad68d40a7f9805f0d4d6c2b75b0a5664fd5e844da2",
        "33355cb42bb31bae5cbb1881afaa4b612b050654ec173b7a627489f844dc8d26"
```

```
      ]
    },
    {
      "byte_beg": 10485760,
      "byte_end": 20971519,
      "proof": [
        "6bfc296a266462eb34c1fbad68d40a7f9805f0d4d6c2b75b0a5664fd5e844da2",
        "2bdff03703f7c71dd27e58ac52dc675a183b9df336ace85c928cb239b5fe095c"
      ]
    },
    {
      "byte_beg": 20971520,
      "byte_end": 31457279,
      "proof": [
        "6bfc296a266462eb34c1fbad68d40a7f9805f0d4d6c2b75b0a5664fd5e844da2",
        "b0409db2ba129ba4279378e22bfebcc492e196544b9a09c9056aa9de4927f07a"
      ]
    },
    {
      "byte_beg": 31457280,
      "byte_end": 38057641,
      "proof": [
        "6bfc296a266462eb34c1fbad68d40a7f9805f0d4d6c2b75b0a5664fd5e844da2",
        "0fd7816bd9f06f3cf976aa24034e5de704cae01eae1f1efa4e2013538f3e02ea"
      ]
    }
  ],
  "proof_data": {
    "0fd7816bd9f06f3cf976aa24034e5de704cae01eae1f1efa4e2013538f3e02ea":
"+FugPFTGTXwdYttZq5dn0MN7mxhQ326xWs46OfS9OQp0Jym4OAMAAAAAAAAAADgAQAAAACptkQCAAAAALxUxk18HWLbWauXZ9DDe5sYUN9usVrOOjn0vTkKdCcp",
    "2bdff03703f7c71dd27e58ac52dc675a183b9df336ace85c928cb239b5fe095c":
"+FugPedu9h8GXkgK0RrW0OD3SMiFqcLCElw7SYmLEEyOa1S4OAEAAAAAAAAAAACgAAAAAAD//z8BAAAAAG3nbvYfBl5ICtEa1tDg90jIhanCwhJcO0mJixBMjmtU",
    "33355cb42bb31bae5cbb1881afaa4b612b050654ec173b7a627489f844dc8d26":
"+FugOgJLxu2N9IQYm7cXjVahA3nG+i5WKgjclAbdoPB4iw+4OAAAAAAAAAAAAAAAAAAAAAAD//58AAAAADoCS8btjfSEGJu3F41WoQN5xvouVioI3JQG3aDweIsP",
    "6bfc296a266462eb34c1fbad68d40a7f9805f0d4d6c2b75b0a5664fd5e844da2":
"+JGAgICgMzVctCuzG65cuxiBr6pLYSsFBlTsFzt6YnSJ+ETcjSaAgKAr3/A3A/fHHdJ+WKxS3GdaGDud8zas6FySjLI5tf4JXICAgICgD9eBa9nwbzz5dqokA05d5wTK4B6uHx76TiATU48+AuqAgKCwQJ2yuhKbpCeTeOIr/rzEkuGWVEuaCckFaqneSSfweoCA",
    "b0409db2ba129ba4279378e22bfebcc492e196544b9a09c9056aa9de4927f07a":
"+FugNuieiyxh4owFusQG1Xj7t2Ogj4gGoC4NiRXqQLbjRuy4OAIAAAAAAAAAAABAAQAAAAD//98BAAAAAObonossYeKMBbrEBtV4+7djoI+IBqAuDYkV6kC240bs"
  }
}
```

Figure 22. Example content object proof for fast verification of content integrity. A user that has downloaded any of the segments can verify the root hash without needing to download the other segments, thanks to the Merkle tree storage of the sub proofs for each Part within the object.