# Security White Paper

corvee™

**Disclaimer**
This is a living document and will be updated to conform with necessary business and use decisions and as the product evolves. Questions about the document should be directed to support@corvee.com.

# Table of Contents

## KEY SECURITY FEATURES

- End-to-end encryption
- Server ignorance
- No crackable information stored
- Thrice encrypted on transport
- User control over sharing
- Team-managed account recovery

## Principles

### Security by Design

We cannot compromise data that we don't know. Because we never see and never have a way to access our users' data, we provide them with the highest level of privacy possible.

### Math Above Policies & Software

Even the best-designed policies and software have weaknesses that make them vulnerable to exploits. For this reason, all data in Corvee is transmitted and secured in a provable, practically invulnerable cryptographic manner.

### People Are a Part of the System

People are not perfect, and their behavior should not always be dictated by the software. We must keep our demands on the user to a minimum and abstract the most difficult parts away from them to provide a simple, yet ultimately secure, product.

### Transparency

Every element of Corvee has been designed with security front and center. By making sure that our security is not dependent on secrets, but rather built into every facet of Corvee, we are able to explain how it all works in the document and make our security decisions verifiable by an external party.

### Know Your Audience

While each user is unique, the nature of the business means that users need to solve common problems. By having a deep understanding of our customer and their needs, we are able to provide solutions that work the way they work.

## Security

Corvee is a system that is encrypted from end to end. The basic building block of Corvee's security lies in the fact that all data in the system is encrypted and decrypted using keys derived on the client side from the master password, which remains totally invisible to Corvee at every point, and Corvee cannot access it.

To this end, we have designed our user authentication differently than traditional password-based user authentication. Instead of traditional authentication mechanisms, Corvee focuses on two points: avoiding the pitfalls of traditional user authentication by substituting it with the Secure Remote Password Protocol (SRP) and using the derived key from SRP as a secure and secret PSK (pre-shared key) to enable end-to-end encryption.

## User Authentication

User Authentication is the cornerstone of security. It permits us to distinguish between authorized users, unauthorized users and malicious actors. In line with our guiding principles, we have developed a set of desirable qualities for our user authentication mechanism:

1. Prove client and server ID
2. No eavesdropping
3. Not replayable
4. No secrets exchanged
5. Session key
6. Vault keys
7. Cracking proof
8. User control

## Master Password

Users' master passwords are the ultimate stopgap for our security measures. Because they are never stored anywhere outside the user's own memory, they are impossible to steal. They are used to derive the keys necessary to encrypt and decrypt every document and piece of data on the Corvee system.

## Key Derivation

The user's derived key is used to wrap all the vault keys they have access to. It is derived from the master password of the client, so it is never transmitted to anyone else. The key is derived anew each session and encrypted on the client end, and it is only transmitted to the database once both the client and API verify each other's identity and the client encodes the derived key.

When deriving the key from the password, your master password is first preprocessed by normalizing the password and then salted. It is then passed through a slow-hashing function known as PBKDF2 with 100,000 iterations. This produces a 32-byte key, which is then used to derive the pre-shared key.

### Preprocessing master password

Neither passwords nor any key derived thereof are ever transmitted to a server, and they never leave the client. This means that there is never a point in any workflow where a malicious actor can intercept any passwords, attempted passwords or failed passwords

#### Normalization

The first step in deriving the key is to normalize the password using Unicode equivalence (NFKD normalization). For instance, the Spanish letter ñ can be represented in two different ways: the Latin character "n" (U+006E) followed by the combining tilde "˜" (U+0303) is equivalent to the single "ñ" character U+0041. This can clearly result in the same input from the user producing two different strings, ultimately resulting in different hashes and consequently a malformed key. By normalizing the user's password before hashing the password, Corvee ensures that the user's password always results in the same hash, allowing Corvee to unambiguously produce the same hash during key derivation.

The ghost of Bob (Roberto) Bolaño is very frustrated. He is trying to log onto his favorite blog - CrochetAndKittens. and he keeps getting a "Wrong Password!" message. He knows certainly that his password is "bbolañ0", and there is no chance he typed the wrong password.

Fortunately, all the time he has spent haunting software engineers has helped him understand how most websites use password hashes to verify input passwords. He quickly realizes that the "ñ" in his password was encoded into a different UTF-8 string when he first created his account in 2001. With the newest browser update, his password "bbolañ0" is producing a different hash than when he first created his account, so the hashes no longer match.

He angrily composes a beautifully written and heart wrenchingly crafted support ticket for the dating app, demanding that they implement password normalization prior to hashing so that no one else must suffer his same fate. His support ticket is currently nominated for this year's Nobel Prize in Literature.

**Salting**

Salting passwords before hashing protects batches of passwords against rainbow tables. After the master password is normalized and encoded into an unambiguous UTF-8 string, a random 128-bit salt is generated. This randomly generated salt is then appended to the normalized master password in preparation for hashing. This extra salt prevents users' passwords from hashing to the same value.

## Slow Hashing

Corvee implements slow hashing algorithms to increase the difficulty of mounting password cracking attacks. These attacks rely on the ability of attackers to guess a huge number of passwords and hashes quickly — and by slowing down each guess enough, we make the task so time consuming that it would take many lifetimes to complete.

Slow hashing provides greater protection than regular one-round hashing by iterating the hash function over the previous iteration's output. Corvee implements the hashing algorithm PBKDF2 with 100,000 iterations, significantly slowing down anyone attempting to brute force a password.

### PBKDF2 (Password-Based Key Derivation Function 2)

The resulting concatenation of the normalized master password and the salt is then used to derive the key with PBKDF2 using SHA-256 as its hashing function. The key is derived from over 100,000 iterations of the PBKDF2.

PBKDF2 allows for an arbitrary amount of iterations to increase computational time to slow down potential attackers.

Alice decided to lock her house with a hash key - simple and fast. The first time Alice lost her hash key, she was so scared that she decided to buy 999 extra salted keys to hang on her keychain.  It ended up being in vain, though. Not only did she have to look through a ton of keys to find the right one every time she got home, but the second time she lost them, Craig found them. He was quickly able to try them one by one until he found the correct hash key, cracked the lock, and stole her Sherlock Holmes collection.

Luckily, he did not steal her books on Slow Hashing. Alice then changed the lock to her house from a fast hash to a slow hash - and instead of taking her one second to open the lock, it now took her ten. Even if it was a little slower, she could easily open the lock because she usually picked the right key on the first try. Craig, on the other hand, had no way of knowing which key was the right one, and he could not try every single key until he found the correct one before she got home.

The next time she lost her keychain, she was even able to pick up a refreshing vanilla soy latte with hazelnut before going home, finding Craig crying next to her front door.

## Password Authenticated Key Exchange (PAKE)

PAKE methods are interactive methods used between multiple parties to establish a shared key for secured connection. There are several different types of PAKE, but not all of them satisfy our principles. Commonly, both parties are privy to a password that allows them to establish a shared key for communication. While this is a strong guarantee of data security, it does not satisfy our “Security by Design” principle. If the server knows a password, it becomes a system vulnerability that must be protected. To that end, Corvee uses an Augmented PAKE, which guarantees that the server remains incognizant to any password-like information.

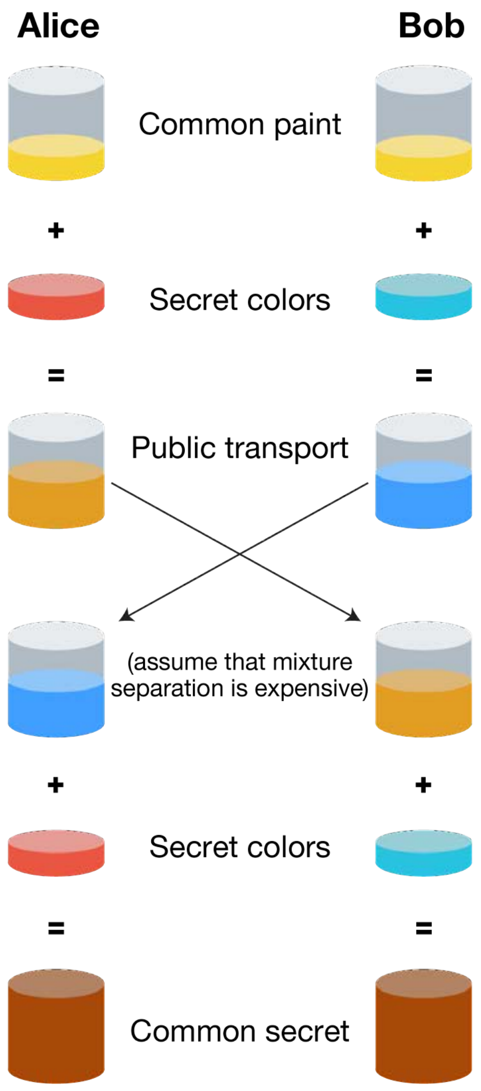
### SRP (Secure Remote Password) Protocol

The SRP Protocol is an Augmented PAKE based on the Diffie-Hellman Key Exchange. This protects against two well-known vulnerabilities: it protects against a man-in-the-middle attack from eavesdropping on enough information to gain access to the system, and it avoids the need for a server to know any password-like information to authenticate its clients. This renders any attack on the system meant to eavesdrop on information useless.

A key benefit of using SRP versus other PAKEs is that it can authenticate a password without ever passing it to the server. This means that the password can simultaneously be the source from which to derive a key to encrypt data without ever revealing the key. We leverage this capacity in Corvee by wrapping vault keys with the derived key, ensuring that the key is never logged, stored or even revealed at any point of our system.

This works because one party can prove another party is in possession of a password without ever seeing it. The secret lies in each party independently generating a key based on their password and a certain operation. Then, the generated key can be provided to the opposing party to again combine with their own private key, such that both parties are in possession of a matching result.

**SRP overview**
A simple analogy of the Diffie-Hellman protocol can be demonstrated using colors (via Wikipedia):



In this example, the secret colors are both unknown to the opposite party. This is analogous to the user password. The addition of the common paint is analogous to the operation performed on the user password. After exchanging the resulting keys, each party performs another operation on the key received from the other party. Without both parties acting honestly, the common secret will not match. This verifies both the client to the server and the server to the client without either having to reveal their secret.

Eve is such a Libra. She always likes to hear both sides on any subject. She likes it so much, in fact, that she's listening in on Carol's chat right with her bank's customer support over the unsecured public WiFi at the public library. It's so much fun!

Carol apparently has had some suspicious activity on her account. "Well," cackles Eve, "that's nothing compared to the suspicious activity she'll have once I get her password!"

Eve starts reading over her log of Carol's network traffic but she can't find Carol's password. She looks again and again, but Carol never once revealed her password! How could her bank prove it was Carol in the chat?

Carol's bank has wisely implemented SRP, so Carol never has to reveal her password to her bank to prove she is who she says she is. Eve listens and listens, but she finally realizes she'll never learn Carol's password.

Eve furiously slams both her hands down equally strongly on her desk while Carol remembers that that $120 charge at the local Ourby's from last week was indeed regretfully hers.

**SRP in depth**
Take a deep breath and brace yourself. We'll get through this together. Here, we lay out a detailed and mathematical explanation of how SRP works. If you want to continue, just know that the next section is a lot easier to read and understand.

SRP is based on arithmetic on the multiplicative group of integers modulo N[1] and the generator[2] g, Let q and N be prime where $q = 2N + 1$. This makes N a safe prime[3] and q a Sophie Germain prime[4]. Furthermore, let N be large enough that computing discrete logs mod N is infeasible. Finally, let $k = H(s,p)$ where s is a small salt and p is the user password sourced secret; and $v = g^x$ where $x \geq k$.

**From here, the protocol proceeds as follows:**

**1. Alice to Bob:**

a. Generate random value a.

b. Send identifying username I and ephemeral user key $A = g^a$.

**2. Bob to Alice:**

a. Generate random value b.

b. Send small salt s and ephemeral host key $B = kv + g^b$.

**3. Both calculate:**

a. $u = H(A, B)$

$S_{Alice} = (B - kg^x)^{(a + ux)} => (g^b)^{(a + ux)}$

$K_{Alice} = H(S_{Alice})$

$S_{Bob} = (Av^u)^b => (g^b)^{(a + ux)}$

$K_{Bob} = H(S_{Bob}) = H(S_{Alice}) = K_{Alic}$

---

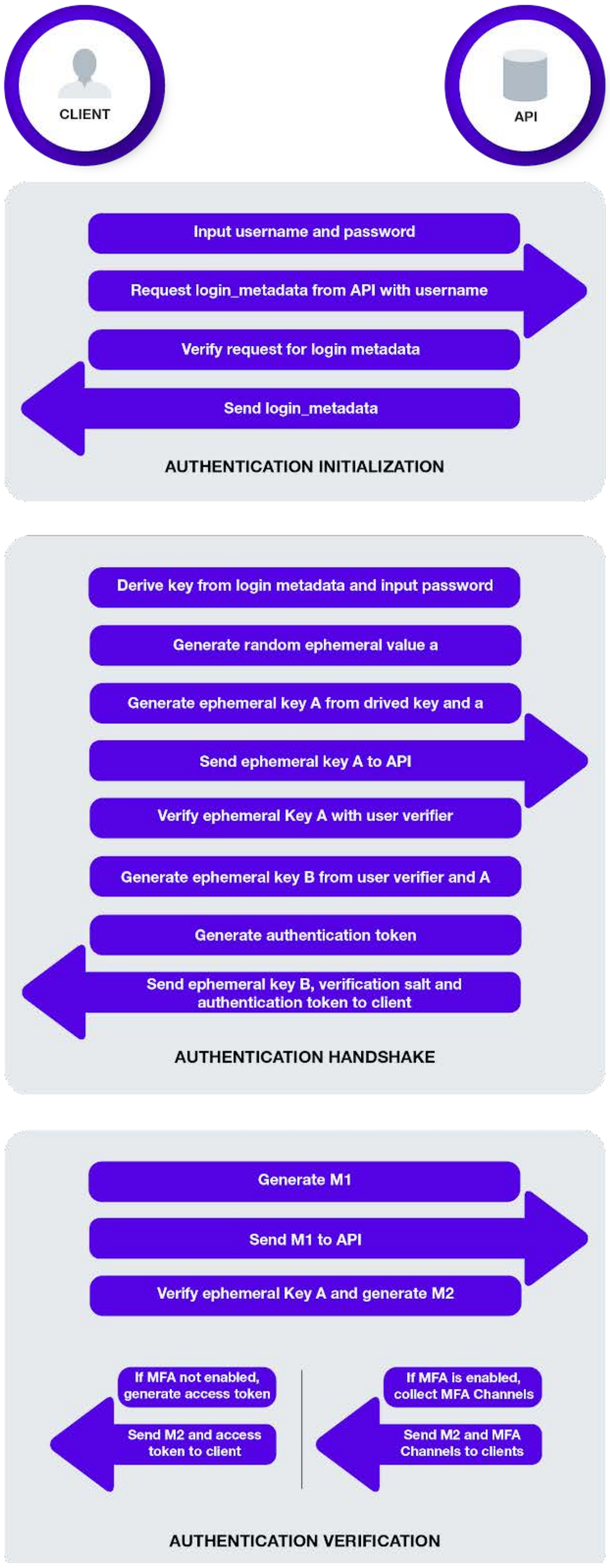
[1]A ***multiplicative group of integers modulo n*** refers to the integers from the set {0, 1, ..., n-1} that are coprime (i.e., relatively prime) to n.

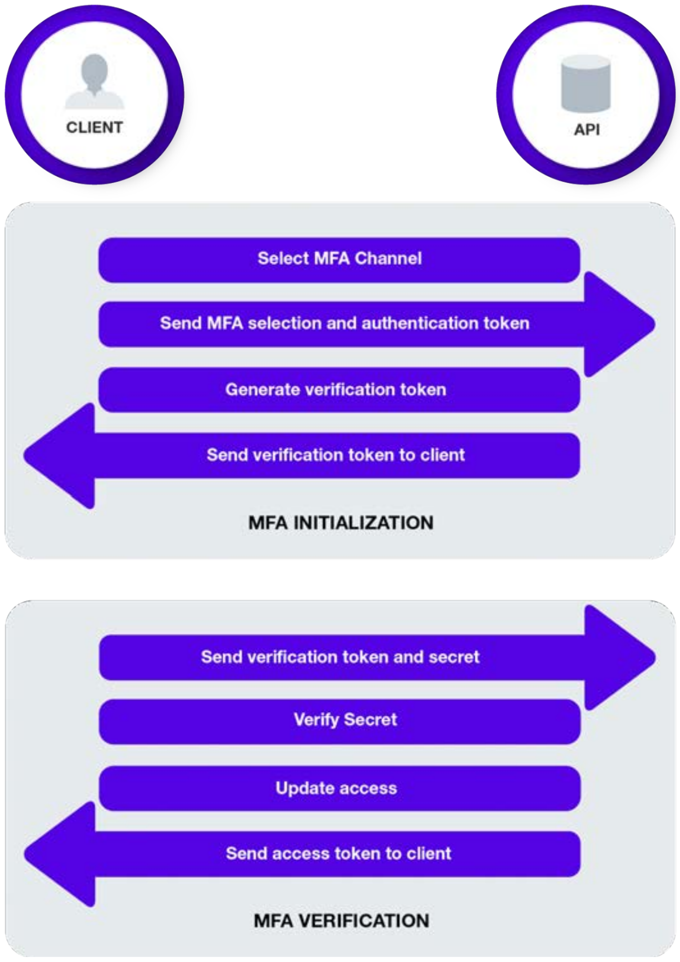
[2]A ***generator g*** for a multiplicative group of integers modulo n is an integer g < n where the powers of g produce all possible residues modulo n coprime to n and give each exactly once.

[3]A ***safe prime N*** is a prime number s.t. N = 2p + 1, where p is also prime.

[4]A ***Sofie Germain prime p*** is a prime number s.t. N 2p + 1, where N is also prime.

At this point, Alice and Bob can compare their keys and see they are the same.
The keys here negotiated become the derived key, the PSK.


CLIENT
API
Input username and password
Request login_metadata from API with username
Verify request for login metadata
Send login_metadata
AUTHENTICATION INITIALIZATION
Derive key from login metadata and input password
Generate random ephemeral value a
Generate ephemeral key A from drived key and a
Send ephemeral key A to API
Verify ephemeral Key A with user verifier
Generate ephemeral key B from user verifier and A
Generate authentication token
Send ephemeral key B, verification salt and authentication token to client
AUTHENTICATION HANDSHAKE
Generate M1
Send M1 to API
Verify ephemeral Key A and generate M2
If MFA not enabled, generate access token
Send M2 and access token to client
If MFA is enabled, collect MFA Channels
Send M2 and MFA Channels to clients
AUTHENTICATION VERIFICATION


CLIENT
API
Select MFA Channel
Send MFA selection and authentication token
Generate verification token
Send verification token to client
MFA INITIALIZATION
Send verification token and secret
Verify Secret
Update access
Send access token to client
MFA VERIFICATION

**Pre-Shared Key**

A new Pre-Shared Key (PSK) is generated with each session.

The first time a user activates their account, the PSK is generated as a random 32-byte alphanumeric token and encrypted using RSA with a SHA-256 hash function using the user's public key. Once the account has been activated and the user has created a password, the PSK is instead the output from SRP from the randomly generated keys from the API and the client.

In either case, the PSK is returned to the client-side server paired with an access token. This token continues granting access to the system until the token expires or the user logs out.

## Multi-Factor Authentication (MFA)

The addition of MFA provides an extra layer of security and significantly increases the difficulty for an attacker to complete a malicious login. Not only would it require successfully bypassing all the cryptographic and security protocols that are inherent to Corvee, but it requires a successful and simultaneous attack on the second factor used to verify a login attempt. Because users who opt in to MFA must complete *both* the password verification and their second, separate authentication factor, an attacker would need to confront two secure systems simultaneously, significantly raising the difficulty of mounting a successful infiltration.

#### SMS
SMS multi-factor authentication adds an extra layer of security to a user's Corvee login by sending them a code via SMS to input as part of the user authentication process. The code sent by Corvee must match the code input by the user in order to successfully authenticate. Because access to a user's SMS messages is unlikely to be tied to their other logins, SMS MFA adds another layer of security to a user's authentication.

#### TOTP
Like SMS MFA, TOTP (Time-Based One-Time Password) authentication requires a user to input a code generated by a third-party app as a part of the signup process.

Examples of apps that enable TOTP authentication include Google Authenticator, Microsoft Authenticator and Authy.

## Account Lockout
In addition to the safeguards in place against password-cracking attacks, there is a simple heuristic method to detect a malicious actor. All password-cracking attacks depend on the ability to try vast numbers of passwords. Not only does Corvee make the possibility of correctly guessing a password within a human lifetime a mathematical impossibility, but it also protects accounts from excessive login attempts.

## Data Encryption
Corvee is an end-to-end encryption system. At no point is any sensitive information stored or transmitted under less than two layers of encryption, and it is usually transmitted under three layers of encryption.

By combining the security of AES-256-GCM encryption and wrapping the keys with RSA-4096 wrapped key management, Corvee operates under strong cryptography that makes unauthorized access to any data impossible.

### Data transmission
Recognizing that data can be intercepted or tampered with during transmission, Corvee ensures that data is never left vulnerable during transit. All connections take place over HTTPS. Finally, all authorized requests to the API must be signed using HMAC.

#### HTTPS
Corvee's first defense is that traffic flows over HTTPS tunnels, verifying the identity of the server by a trusted third party. HTTPS is an extension of the original protocol that powered the internet, (i.e., HTTP [Hypertext Transfer Protocol]). This extension provides for secure communication over a network by empowering a trusted third party to manage and issue certificates, which allows for privacy and integrity of transmitted data, as well as identity verification for accessed websites.

HTTPS likewise enables encryption of all data in support of our E2E encryption model as one of the strategies to keep your data secure under transit.

#### Encrypted tunnel using PSK
All data transmitted from the client to S3 or to the API is transmitted encrypted with AES-256-GCM using the client's PSK.

## AES-256-GCM

Corvee uses the NIST-selected block encryption cipher AES-256-GCM to transmit any data through an encrypted tunnel using the PSK. AES describes a symmetric key encryption algorithm based on the Rijndael block cipher.

Being a block encryption cipher, it uses a block cipher mode of operation to encrypt/decrypt streams of data, such as the Galois/Counter Mode (GCM). GCM is a high-performance mode of operation, which allows for highly secure and efficient encryption and decryption of streams of data. Corvee's adherence to the strongest variant of AES (256) means that any data from Corvee's servers is transmitted securely and privately, while using GCM as the mode of operation allows for efficient and authenticated access to the data prior to transmission over the encrypted tunnel.

AES-256 ensures undecryptability. A brute force key-cracking attack to recover an AES-256 encryption key requires 2^254.3 operations and thousands of terabytes of storage. This high barrier precludes even the most determined attacker from read access to documents transmitted to or from Corvee.

Alice has a secret - she can't get enough about cat memes. Her prized possession is her collection of 344 GB of the finest cat memes ever shared. She doesn't want anyone to know about her passion for felines, though, so she keeps it safe in a misleadingly labelled folder on her computer desktop (it's labelled "not_cats")

However, Alice is worried. She recently found out that Trudy has been breaking into computers around America. She's worried that Trudy might break into her house, open her computer (Alice never locks her screen), and delete her "not_cats" folder, or worse - email it to all her friends and family!

But then Alice has a brilliant idea. What if she changes the files so that when Alice looks at them, she can see cute kittens, but when Trudy does they just look like random files? Alice needs a way to lock and unlock her data. She creates a string that only she knows - her key - and uses it to encrypt all her files with AES-GCM-256. She is now the only person who can open her files so they look like felines.

She now can leave the house without worries that Trudy might be able to know her secret. After all, what would be in a folder labelled, "not_cats"? Not cats, that's for sure.

#### Request signature
Requests can come from any source over the internet. For this reason, it is important to distinguish between authentic, legitimate requests from authorized users and inauthentic, potentially malicious requests from unauthorized actors. Every request to the Corvee API requires a request signature that verifies the sender's legitimacy, as well as that their request has not been tampered with.

##### HMAC-SHA-256
Every authenticated request on Corvee is signed using HMAC (a key-hashed message authentication code). This verifies the data integrity and authenticity of requests to authenticated endpoints on Corvee's API.

Corvee uses SHA-256 as the HMAC's hashing algorithm.

## Sensitive Data
A key feature in Corvee is the proper handling and storage of all sensitive data. As an end-to-end encrypted system, Corvee makes sure to properly handle and store sensitive data using encryption, authentication, anonymization and other methods. By authenticating every request that affects sensitive data, we make sure that only those authorized to view or modify sensitive data can do so.

### What is sensitive data?
Sensitive data, such as documents uploaded to our system along with their contents, are encrypted or properly handled to prevent our system or anyone at Corvee from being able to analyze sensitive data. The entire system is built around the idea of keeping this information unknown or unknowable to the staff at Corvee.

In order to protect this data, Corvee implements a wide variety of strategies to secure necessary data, anonymize relevant data and discard any personal identifying information.

### Symmetric encryption
All data in Corvee is stored encrypted. This ensures that only users in possession of the proper key can decrypt and modify any data on the system.

#### AES-256-GCM
All objects are kept encrypted using an object key with AES-256-GCM. The object key is stored asymmetrically wrapped using the client's RSA-4096 key pair in the database, which in turn is wrapped using the user's key pair.

This user key pair can only be unwrapped using the user's derived key. Only the authorized user can traverse the key chain and decrypt the content.

##### Authenticated algorithm
AES-256-GCM is an authenticated encryption algorithm. This guarantees not only data authenticity, but also integrity. Changing any part of a transmitted message would invalidate the message and make it undecryptable, thus ensuring the message's data integrity. Attacks centered on altering message contents or transmissions are rendered useless by using this method.

### Infrastructure-level encryption

In addition to the application-level encryption, Corvee also stores all data in AWS services encrypted at the infrastructure level. The data stored in AWS services — namely, in S3, RDS, or any related configuration — is encrypted using an AWS KMS key. These keys are kept secret as an added precaution at the infrastructure level. However, in line with our principle of "Security by Design," the security of the system does not depend on the secrecy of these keys because the data is encrypted at the application layer as well.

## Key Management

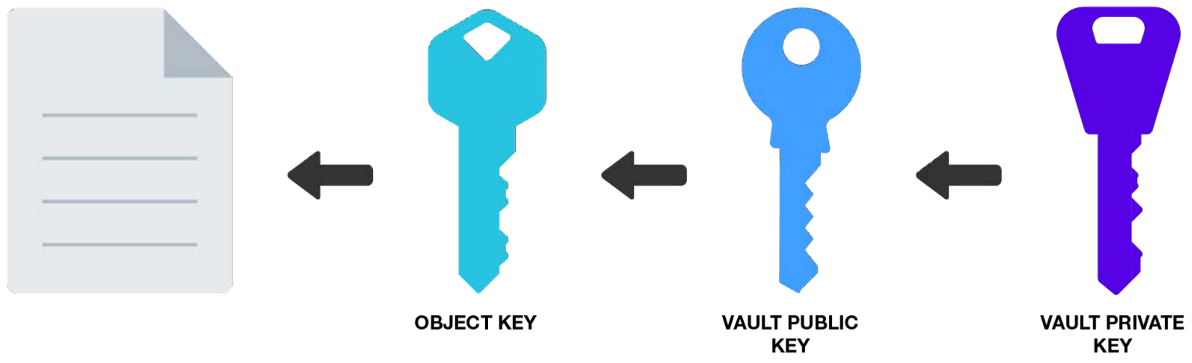
When a user uploads a document, it becomes visible to other authorized users but not to anyone at Corvee or other unauthorized users on the platform. The place where any user can upload an object, such as a document or questionnaire reply, is referred to as a "vault." A vault encompasses the location where clients, users, firms and entities may upload an object and indicates to whom an object belongs. Additionally, users can belong to an organization, which can be either a client or a firm, even though only users can take any action — organizations only serve as mechanisms by which to share keys. Because every uploaded object is encrypted using a key, there must be a way to share keys securely with other authorized users such that all users authorized to view a document can do so.

### Key creation

The user's key is generated as an RSA-4096 key pair. The user's private key is then wrapped using the key derived from the user's password. This wrapped key is saved to the database, where it can be retrieved and safely sent to the user, who may unwrap it and use it in the Corvee client-side application only through knowledge of the chosen password.

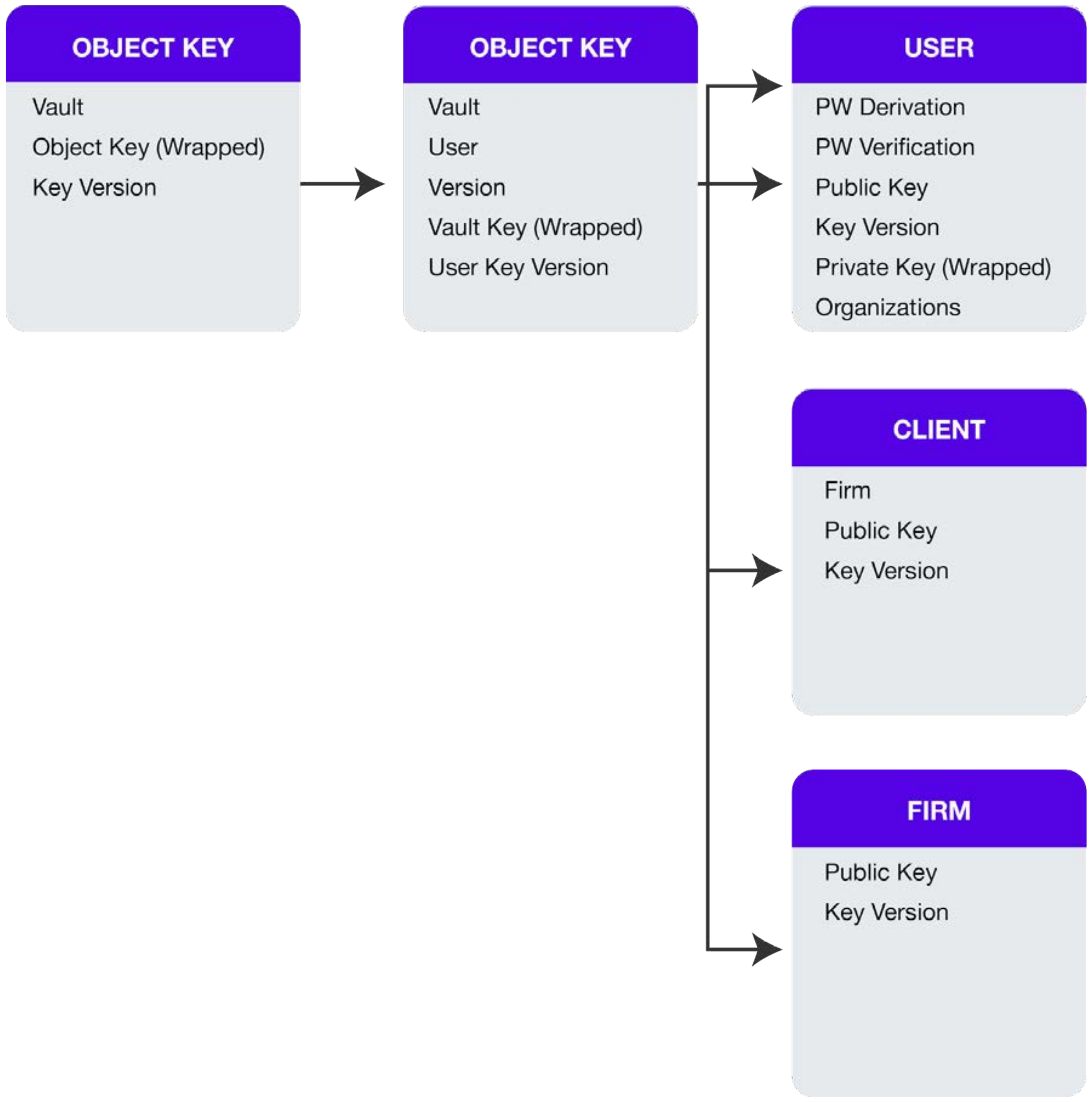
### Key exchange

Corvee solves the problem of key exchange among users, clients and firms by extracting the concept of a vault. A vault refers to the place an object was uploaded, and the vault key refers to the key pair used to wrap the key that encrypted the object within that vault.



When an object is saved, be it a document, questionnaire or any other such material, an object key is generated and used to encrypt the object. This object key is then wrapped using the vault key. It is this wrapped vault key that enables decryption of the object to people who are authorized. For instance, if the object belongs to a firm vault, then the firm key is used to wrap the object key. Every user that belongs to that firm will have a wrapped version of the firm private key, so that they each can unwrap their wrapped firm key and use it to decrypt the object.

In order to fully understand the abstraction of a vault key, however, a deeper explanation into the structure of the key exchange in the database is required. A summary of table relationships regarding keys is presented below to illustrate how a user may traverse the key chain to unwrap a document.



Visual aids always help! When requesting a document from the database, the API fetches the document metadata that contains the following data:

## Document:

-Vault: What vault the document was uploaded to
-Object Key (Encrypted): The wrapped object key that, when unwrapped, can decrypt the document
-Key Version: The version of the vault key used to wrap the object key

The vault field links the document to its uploading vault, which may be a user, a client or a firm. This reference contains the following data:

**Vault Private Keys:**
-Vault: The vault ID
-User: User ID with access to vault
-Version: Version of user, firm or client key used to encrypt the vault key
-Vault key (encrypted): The private vault key, wrapped using the user public key (This can refer to the user key itself, the client key or the firm key.)
-User Key Version: The version of the user key used to wrap the vault key

If the Vault belongs to a user, the vault key is equivalent to the user key. If the vault belongs to a client or a firm (an “organization”), this vault key must further be unwrapped using the user key.

At the final level of encryption lies the user with their wrapped (encrypted) private key. This key is wrapped using the key derived from their password. Unwrapping this key allows the user to unwrap the vault key and traverse the key chain up to the document.

Because the first step in traversing the keychain is to unwrap the user's private key using the user’s derived key, it is impossible for an unauthorized user to unwrap the object key and decrypt any object. However, using this key exchange mechanism, it is possible for multiple users belonging to the same organization to receive their own version of the object key to unwrap organization documents.

Faythe operates a very special sort of mail center:  mailboxes for love letters. Each one of her mailboxes needs to be available to two people, so they can share their letters with each other without anyone else being able to read them.

She keeps all the keys to the mailboxes at her office (because lovers don’t always remember their keys) but as her business grows, it’s getting harder and harder for her to verify every lover before giving them their key to check their mail. She needs to update her records after every breakup, and some of her clients are real heartbreakers. But worst of all, with all of the keys in her possession, her clients have to trust her not to read their mail!

Faythe then has an idea - what if she could just have them get their key themselves?  That way the lovers can decide who they want to share their mail with, and they know she can never snoop on their correspondence.  She decides to lock each key in its own small, PIN-protected locker. Each locker can open for every person who knows the PIN to the key, and the lovers can decide who they want to share the mailbox key with. But most importantly, Faythe doesn’t know the PINs, so her clients know she won’t read their mail!

Her business is booming, and Faythe has never had it easier.

## Data Anonymization

Whenever Corvee needs to access sensitive data, it is always done after anonymization. This need arises during form scanning and identification. The contents of all scanned documents are uploaded to S3 and then scanned through Textract to find all potential textual markers on a specific form, including the form contents. Because this process scrapes sensitive data, it is necessary to anonymize the markers before passing them on to the API where they will be processed. The goal of this anonymization is to be able to filter text that contains personal information from the markers Corvee uses to identify forms. The process is described in more detail in section §DOCUMENT IDENTIFICATION.

## Access Controls

Access controls are fundamental to protecting sensitive data. Corvee employs cryptographic security protocols, server-side security protocols, client-side security protocols and proactive security measures.

### Cryptographic security protocols

Corvee data is always encrypted under two or three layers of encryption. Everything that is encrypted is encrypted using keys appropriate to the level of access that should be granted to each datum. Additionally, these encrypted data are only transmitted through secure and encrypted channels. For instance, when uploading a document, it is:

1. Encrypted on the Corvee App Client using the unique object key
2. Transmitted over an HTTPS channel directly to S3
3. Can only be retrieved indirectly by unwrapping the associated object key with the vault key, which must in turn be unwrapped using the associated user key (unless the vault key is the user key)

**User, Firm, Client Keys**

Every document in Corvee is encrypted symmetrically using its object key, which is in turn wrapped using the vault key. A vault can refer to a variety of parties: a user, a client or a firm. Additionally, each user can belong to an organization: a client or a firm. The process by which a specific user can access organization keys securely is described under §Key Exchange. For the purposes of the following section, we will no longer distinguish between individual user, firm or client keys, but rather refer to them collectively as “vault keys.”

When a user wishes to access a document through the Corvee client portal, the user will request it through the Corvee API. The API fetches the document metadata, which contains the associated vault, the key encryption algorithm and the key version. This information is then used to fetch the associated wrapped vault key from the database that can be unwrapped to unwrap the object key used to decrypt the object.

With the information in hand, the Corvee app client can then fetch the encrypted document from the S3 vault, unwrap the vault key using the user’s own private key, unwrap the object key and then decrypt the document to display, analyze and process within the Corvee app client.

### Server-side security protocols

The Corvee API is hosted on Fargate instances, which limits the interactions anyone can have with it to only those allowed by the API routes. In fact, there is no SSH access available to the Corvee API.

Access to all authorized routes on the API is governed by the access token, generated with every new session. The API validates the access token for every request, barring any user who is not authorized to perform an action from doing so. This, combined with the inaccessibility of the API other than the available routes, secures the Corvee API from unauthorized access.

Additionally, a user without a valid request signature would be unable to unwrap the vault keys necessary to access, read or modify any encrypted content.

## Client-side security protocols

**HSTS (HTTP Strict Transport Security)**
The Corvee client side app enforces the HSTS specification. This restricts all traffic to and from the client side to HTTPS connections. Enforcing this restriction prevents man-in-the-middle downgrade attacks when accessing Corvee.

## Computing infrastructure

### Client

#### Static Hosting

The Corvee App Client is hosted as a static site, with all computations and processes taking place locally on the client. Because it is served as a static site, we can ensure that the same application code is delivered consistently to each person who accesses the app site. This provides us the ability to deploy updates easily and to make sure every user is on the same version, avoiding compatibility issues.

A statically hosted app comes with numerous other advantages: It reduces the risk for code injection, permits code verification, loads more quickly and enables higher availability and uptime.

#### WebCrypto

Corvee uses WebCrypto to calculate hashes, generate electronic signatures, encrypt data and wrap keys. Corvee therefore can be trusted to generate its keys using the best random number generation provided by its host environment.

Additionally, WebCrypto runs natively on host environments and utilizes hardware acceleration when possible. It is therefore highly efficient while encrypting and decrypting, allowing for the use of stronger keys.

### API

The Corvee API runs on serverless infrastructure by AWS Fargate instances in two different US availability zones. Likewise, the database is replicated over the two availability zones. Finally, being located at two different facilities, internet connection and electricity are also provided separately. Separated facilities in this manner provide more stability to the system and more resilience against risks.

Traffic to these instances is controlled first by a load balancer as well as Amazon's firewall, stabilizing the system and preventing any instance from becoming overwhelmed.

As with any Fargate instance, the API only accepts requests and writes to the log. This isolation keeps anyone at Corvee from gaining SSH root access into the system and changing the expected behavior without user knowledge.

## Serverless functions

### Lambda

The best way to ensure the user is in control of their data is to perform as much computation as possible on the client side, and most data processing is performed directly on the client. Some functions, however, do not allow for client-side processing. These have been offloaded to Lambda to permit cloud computation that is out of reach to Corvee but auditable to external parties and users.

Corvee does not share any data from these services with Amazon. The functions that take place in Lambda are kept private by AWS Lambda and Amazon, and they lie out of reach to Corvee once deployed. This means that the user only needs to trust Amazon's long track record for stability and its wide array of customers handling sensitive data. The user has full transparency to the data being sent to Corvee's AWS Lambda functions and can see exactly what computations the Lambda functions do with user data through reproducible binaries and code verification.

### Code verification and reproducible binaries

Our code that handles any data unencrypted is available for any user to inspect and verify. In this way, the user can see that Corvee at no point can access sensitive data for other purposes. Raw data is only ever processed within AWS Lambda, other AWS services isolated from Corvee or on the client machine. The user can verify the code themselves and even run the binaries themselves to verify that Corvee never saves sensitive data anywhere other than their S3 repository and never saves any sensitive metadata without user managed encryption.

### API authentication and access control

**The API has five types of authentication routes:**

1. Unauthenticated routes
    a. Unauthenticated public routes
    b. Unauthenticated routes requiring internal key
2. Authenticated Routes
    a. Authenticated routes
    b. Authenticated routes with non-scoped token
    c. Authenticated routes requiring internal key

These different authentication routes control access to the exhaustive list of Corvee's capabilities. Because the API is run on Fargate, the only interaction possible is via the API endpoints, which are all sorted into these five categories. These all require authentication except for unauthenticated public routes. Any unauthenticated public route is limited to functions that are exclusively related to signing in, account activation or fetching system version.

Authenticated routes require the use of an access token, which is generated as described in § Password Authenticated Key Exchange (PAKE).

Routes requiring an internal token are limited to internal use and require a token that is only available to the Corvee system.

## Proactive Security Measures

### Algorithm agnostic

A key feature that allows us to future-proof Corvee is to make the design as algorithm agnostic as possible. Our system is built in such a manner that we can replace algorithms as needed with absolute ease.

Additionally, Corvee does not hardcode the encryption/decryption methods in the code. Instead, it maintains metadata pertinent to encryption/decryption of every document and datum in the system. This allows us to build a fully backwards-compatible system to account for noncritical upgrades. The system will always be able to decrypt a datum previously stored on the system. This provides us with great flexibility to remain at the forefront of cybersecurity while supporting old document uploads.

### NIST recommendations comparison

Corvee either meets or exceeds every recommendation set forth by NIST for cryptographic algorithms and key lengths:[5]

5 https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf

| Use | NIST Recommendation | Corvee |
|---|---|---|
| **Symetric Encryption algorythms** | AES-128 Encryption and Decryption | AES-256 Encryption and Decryption |
| **Key Agreement** | Diffie-Hellman (DH) with len >= 112 bits security (e.g., SRP 2048) | SRP-4096 |
| **Key Agreement and Key Transport Using RSA** | RSA-2048 | RSA-4096 |
| **Key Wrapping** | AES-128 | AES-256 |
| **Key Derivation** | HMAC - based KDF using approved hash function (SHA-1, SHA-224 for digital signatures) | HMAC - based PBKDF2 using SHA-256 |
| **HMAC Generation and verification** | Key length>=112 bits | Key length = 256 bits |

**RSA-2048 vs. RSA-4096**

Corvee implements RSA-4096, which uses keys twice the length as RSA-2048.

RSA-4096 has an equivalent of 150 bits of security compared to RSA-2048's 112[6], resulting in an encryption 2^38 times — that is, hundreds of billions of times more secure than RSA-2048. While asymmetric encryption schemes with 112 bits of security are approved for use up until 2030,[7] Corvee's implementation future proofs security far into the future!

## Account Recovery

Corvee does not have a way to recover lost accounts. Even if this were implemented, all previously uploaded documents would be rendered illegible because they were all encrypted using keys wrapped with the forgotten password. Given that part of data security is not only preventing unauthorized access, but also ensuring continued access in the case an account becomes inaccessible, Corvee has implemented a few strategies to do so without permitting for crackable systems.

6 https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/fips140-2/fips1402ig.pdf

7 https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

### Emergency Recovery Kit

Upon signing up, Corvee will provide the user with an emergency recovery kit that contains the recovery key for the account. Once a new password has been set, a new key can be derived to grant access back to the account.

### Group Member Recovery

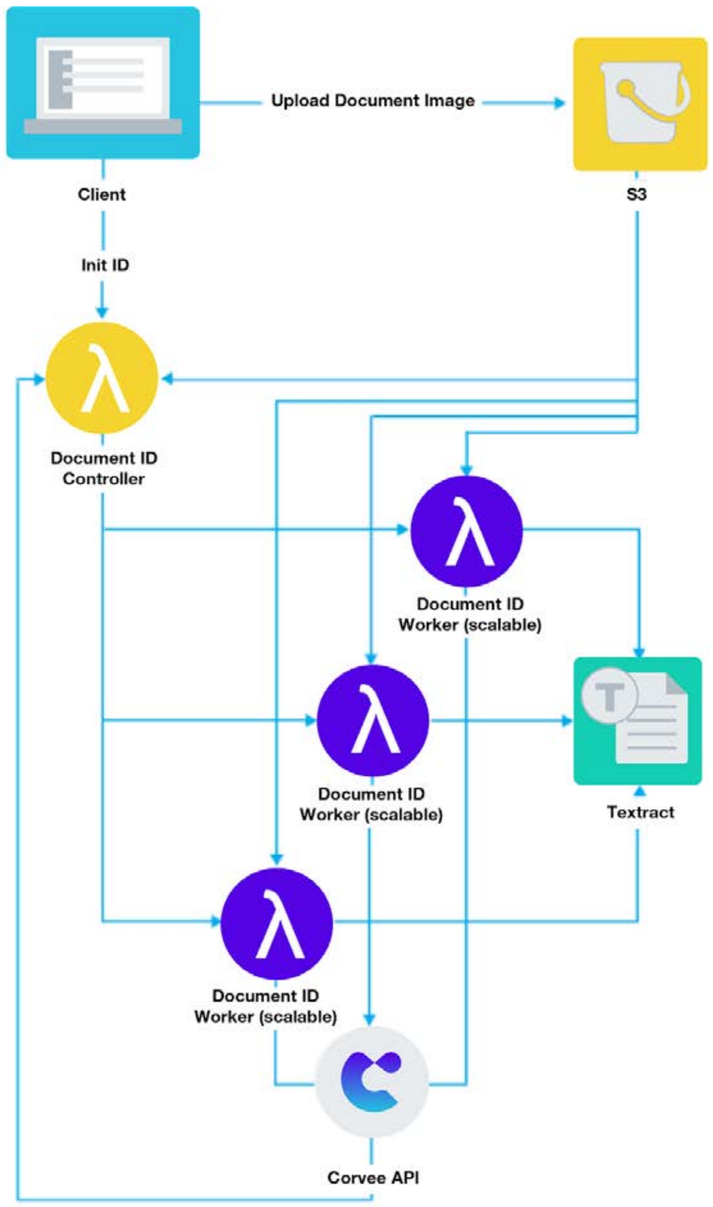
If the emergency kit is not available, the password for the account in question will be unrecoverable. The key sharing protocol implemented to grant access to all authorized users, however, permits another user with group access to still see all the documents and to invite the user whose account was lost back to the group. In this way, the lost account can be recreated with a new account and the same access to the documents it had lost.

### Local Recovery

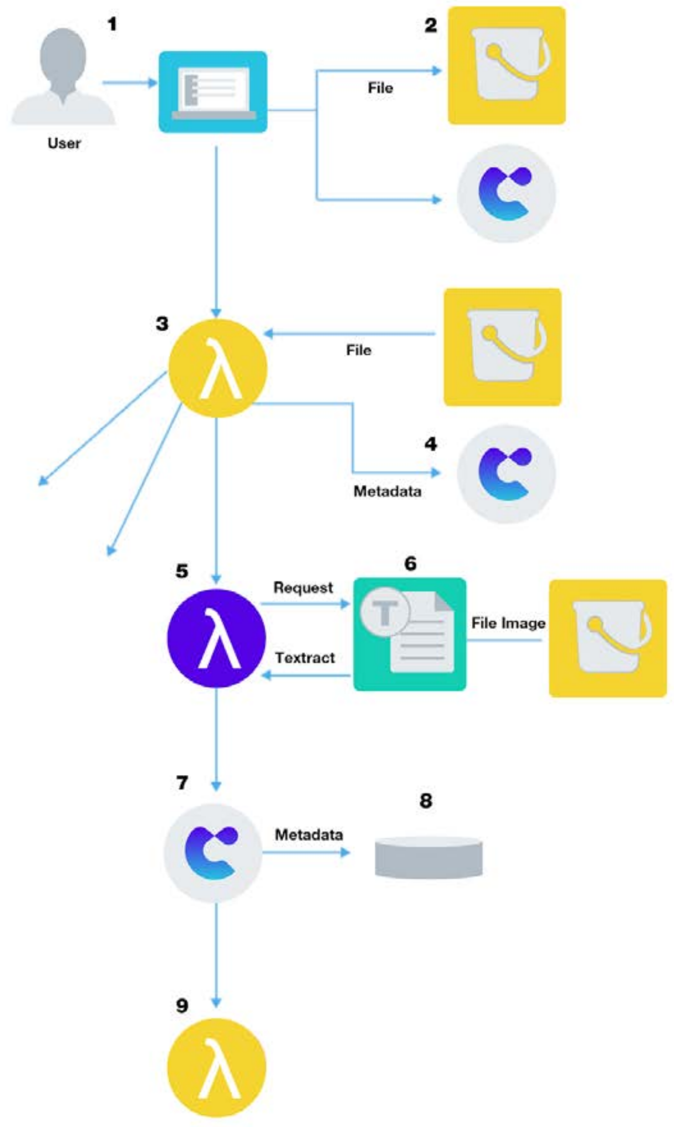
This is not ready yet but will be secret based.

## Document Identification

The document identification mechanism in Corvee is set up as a multistep process that keeps user documents secure.


Upload Document Image
Client
S3
Init ID
Document ID Controller
Document ID Worker (scalable)
Document ID Worker (scalable)
Textract
Document ID Worker (scalable)
Corvee API

This graphic shows the system layout and the scalable components of the system. It is useful, however, to visualize the flow of a single document's journey through this system.



1. A user uploads a document.
2. The client encrypts and uploads image to S3 and saves metadata about the image to API.
3. The user client invoke the lambda function controller with the authentication and document request.
4. The controller fetches the file from S3 to begin processing and write further metadata to the API.
5. The controller divides tke work to identify each page in an uploaded document and spreads it among Lambda Function Workers.
6. Each worker processes the task from the controller and invokes Textract synchronously.

   Textract fetches the file from S3 and extracts all content from each page.
7. The lambda worker anonymizes and queries the API to filter out the form inputs.
8. The API identifies the form and writes the result and other metadata to the database.
9. The controller is notified.

## Textract

Corvee uses Amazon Textract as an OCR engine to extract information from uploaded documents. These extracted words are used to identify the form and year, referred to as markers. Additionally, Textract also extracts the data input on the form.

Corvee has opted out from allowing Amazon or AWS visibility into the extracted text with the goal of protecting sensitive user data.

## Privacy of Sensitive Data

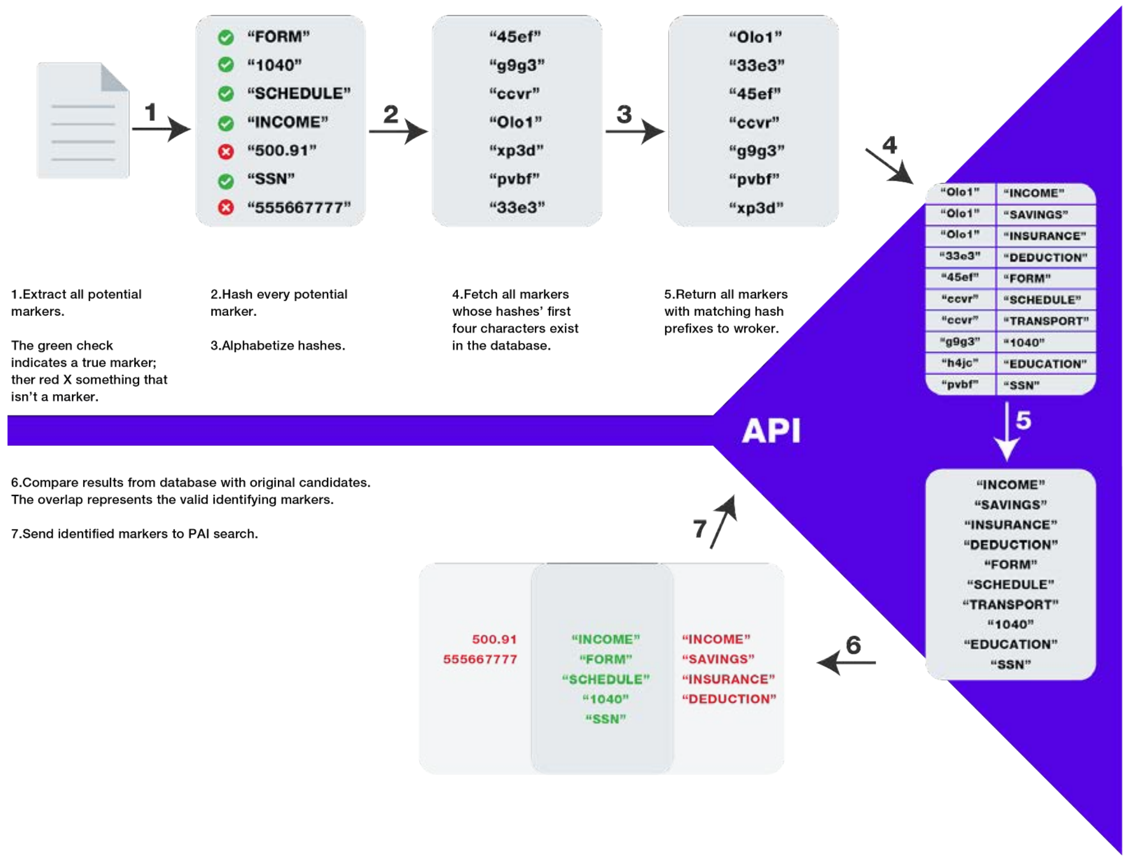
Because sensitive data is impossible to distinguish from non-sensitive data prior to scanning, Corvee keeps the entire process away from Corvee servers. A document scan is encrypted and uploaded from the client-side directly to S3, which triggers a Lambda function to retrieve the encrypted document and decrypt it. The image is then uploaded to Textract, where the text is scanned and sent back to the Lambda function. At this point, the Lambda function must distinguish between sensitive data and non-sensitive data, which it does by anonymizing each potential marker received from Textract and filtering them through the Corvee system to identify which markers contain sensitive data and which markers do not.

### Marker hashing

During form identification, it is inevitable that when the document is scanned, all information on a document will be scanned. Corvee implements marker hashing and anonymization to prevent sensitive data from being picked up and processed in an insecure manner.

In order to filter out that sensitive input information from the information Corvee uses to identify the form name and revision, Corvee anonymizes and changes the order of the scan results before any sort of transmission or processing takes place. Every element in the scan results is preprocessed by hashing using SHA-256 and cropping the hash to a fixed number of characters. This step ensures that none of the elements' original content can be deduced from the hashed, cropped scan elements. Additionally, by cropping the hashes, we increase the number of potential collisions, further masking the true identity of the hashed markers.

Then Corvee blocks meta-analysis of these preprocessed scan elements from betraying the contents by reordering them. The preprocessed scan elements are ordered alphabetically, so that their original order is completely obscured. Only these preprocessed scan elements are transmitted to the Corvee form identification system that keeps a record of the hashes for elements that are useful in form identification. The form identification can then compare the preprocessed elements to the elements useful for form identification and distinguish which hashes belong to useful elements and which hashes belong to sensitive data.



Once the preprocessed elements have been filtered in this manner, Corvee returns the list of nonsensitive element hashes to the user system, which will only then transmit the set of unhashed elements scanned from the form that are not sensitive data. This collection of non-sensitive data is then sent back to the form identification system over a secure channel, where it can be processed to identify the scanned form without revealing any sensitive data.

### No storage of any data other than form name and year

No personal data from uploaded forms is stored in the corvee database. The only extracted data that is stored in the database is the form name, page number, and revision year. However, all data is anonymized and filtered during the form identification process, discarding all input data and leaving it only within the image of the document uploaded by the accountant. That means that the only person with access to the private information on the document will be those with access to the uploaded document itself.

The year is ten years from now. Senator Grace Goode chairs her state's technology panel. She hasn't been late since 1992, always makes sure to cross her i's and dot her t's, and runs the latest and most secure operating system available to government employees (Doors EX-P).

Lately, a new threat has emerged. Proficient accountants have saved their clients millions of dollars, and she needs to analyze their clients' tax returns in order to find out how to raise more money. She thinks she can get it directly from the accountants' electronic storage, because last year she passed a law allowing the government to access all electronic storage providers data. What a dangerous little loophole!

Little does she know, however, that her law can't get her the information she needs. The electronic storage provider can't access the documents themselves! They can only access the name and year of the forms. The actual documents are kept fully under the accountants' control, and only the accountant can access the document uploads or data therein.

Grace will have to find another way to raise money, because no legal loophole will allow her to get tax returns from the electronic storage data providers without the accountant's consent.

## E-Signature

### Data Integrity

Data Integrity is one of the key components of NIST-DSS. It is important that data cannot be tampered with or changed or that any changes are logged and kept visible. Amazon's QLDB is an ideal solution for this challenge as a transparent, immutable and cryptographically verifiable ledger.

### Audit Trail

Every action in Corvee's E-Signature component is logged and kept in an immutable ledger, which makes auditing Corvee Agreements simple, verifiable and always available. No one at Corvee has the capacity to edit or change any occurence on a Corvee Agreement.

#### QLDB

QLDB is Amazon AWS's transparent, immutable and cryptographically verifiable ledger database. It tracks and stores every change in an application such that it can be later recalled and verified. Because it does not permit deletions or changes, it is impossible for anyone to undo an action. Any amendments to an agreement are appended to the ledger, but the original action remains verifiable.

This functionality protects all users from potentially abusive contracts or data loss, so no matter what happens, a record remains of the signed contract, as well as any subsequent changes.

Judy the Judge (her preferred title) is presiding over an interesting case. Alice and Bob both bought the same virtual timeshare in a Simulated Reality game from Oscar last night. They both signed the electronic document using an E-Signature, so it is binding and valid - one of them now legally owns a virtual timeshare on the beach in the Dune World of Qryptonia. However, they both claim they signed it first, so it should belong to them. What a pickle!

Oscar, being a clever software engineer, implemented his contract using an immutable, verifiable database. Pulling out his phone, he queries the database based on timestamps and proves that Alice signed at 2:12AM, whereas Bob signed at 2:14AM, meaning that by the time Bob signed, the timeshare already belonged to Alice. He happily pulls up his bank account to see his money while he waits for Judy to tell Alice the good news.

That night, Alice serves herself a happy mai-tai in front of her computer, waiting for next week when she could sip it in her virtual timeshare on the beaches of Qryptonia.

## Appendix A: RSA

RSA is a public domain **asymmetric cryptosystem** used mostly to share keys. It works by generating public/private key pairs for each party involved in a transaction. In this pair, the public key can be used to encrypt data, but cannot be used to decrypt it. Conversely, the private key can be used to decrypt data encrypted by the public key but cannot encrypt the data for the public key to decrypt. This contrasts with symmetric encryption, where the same key is used to both encrypt and decrypt data.

### Asymmetric Encryption

In an RSA encrypted transaction, both parties send their public keys to the other. These can be shared freely because they can only be used to encrypt data but not to decrypt it. Once the client receives the server's public key, the client can encrypt its request to the server using the server's public key and make the request. The server receives the request that the client encrypted using the server's public key and uses the server private key to decrypt the request. The server can then encrypt the response using the client's public key and respond to the client with the encrypted response. The client then uses the client's private key to decrypt the response from the server, which the server encrypted using the client's public key.

### Key Generation

Each party participating in an RSA cryptosystem needs to generate their public/private key pair.

1. Select two integers p and q such that $p \neq q$ and p,q are both randomly generated prime numbers of similar magnitude and both are kept secret.

2. Find the modulus for both the public and private keys n, where $n = pq$. n is also referred to as the key length.

3. Find $\lambda(n)$, where $\lambda$ refers to Carmichael's totient function[8]. The original statement of RSA used Euler's totient function[9], which can be substituted here with some simple arithmetic.
   Because p and q are prime, n is definitionally the lowest common multiple of p and q. It follows that
    a. a. $\lambda(n) = \lambda(pq) = \lambda(\mathrm{lcm}(p, q)) = \mathrm{lcm}(\lambda(p), \lambda(q))$
    b. b. Because p and q are prime, $\lambda(p) = p-1$ and $\lambda(q) = q-1$.
    c. c. $\lambda(n) = (p-1) * (q-1)$

4. Find an integer $1 < e < \lambda(n)$ that is coprime to $\lambda(n)$.

5. Find the private key exponent d where $d = e-1 \bmod \lambda(n)$, where d is also known as the modular multiplicative inverse of e mod $\lambda(n)$.

6. The public key now consists of the modulus n and the public encryption exponent e.

7. The private key now consists of the private decryption exponent d.

8. p, q, and $\lambda(n)$ must be discarded.

### Encryption

Assume Bob and Alice have created their key pairs and traded their public keys. If Alice wants to send Bob a message, she will encrypt it using Bob's public key, which consists of modulus $n_{Bob}$ and encryption exponent $e_{Bob}$.

First, Alice must turn her message MAlice->Bob into an integer $m_{Alice\text{->}Bob}$ where $0 \leq m < n$ by using a padding scheme. Alice creates the wrapped message using:

$$c_{Alice\text{->}Bob} = m_{Alice\text{->}Bob}^{(e\_Bob)} \bmod n$$

## Decryption

Assume Bob and Alice already created their key pairs and traded their public keys. Alice just sent Bob a wrapped message $c_{Alice\text{->}Bob}$ using his public key. Bob can reverse the calculation to discover the original unencrypted message $m_{Alice\text{->}Bob}$ by using his decryption exponent d_Bob:

$$(c_{Alice\text{->}Bob})^{d\_Bob} = (m^{e\_Bob})^{d\_Bob} = m \bmod n$$

Now given m, Bob can find the original message M by reversing the agreed upon padding scheme.

[8] Carmichael's totient function associates to every positive integer n another positive integer $\lambda(n) = m$ such that $a^m = 1 \bmod n$ for every integer a between 1 and n that is coprime to n.

[9] Euler's totient function refers to $\Phi(n) = m$ where $1 \leq m \leq n$ and m is the count of positive integers that are relatively prime to n.

# Appendix B: AES-256-GCM

The AES (Advanced Encryption Standard) is a subset of the Rijndael block cipher. The number refers to its strength (with 256 being the strongest and the one in use for Corvee). Because AES describes a block encryption cipher, it must implement a cipher mode of operation in encrypt and decrypt streams of data.

## Symmetric Block Encryption

Symmetric encryption contrasts with asymmetric encryption in that symmetric encryption uses the same key to both encrypt and decrypt a message; asymmetric encryption uses a public and private key pair to respectively encrypt and decrypt a message.

AES is also a block cypher, meaning that it can only encrypt a fixed block size of 128-bit blocks. That means that messages of different sizes must be encoded with either padding or by stream encryption through a mode of operation.

The state of the message prior during its encryption is referred to as the "state."

**The block encryption is described here at a high level:**

1. Derive sufficient round keys for each round, depending on the strength of the algorithm. For Corvee's AES-256 implementation, we need to generate 15 round keys - one for each round, plus one to initialize the encryption.
   Each round key is derived using Rijndael's key schedule[10].

2. Initialize the encryption:
   a. a. AddRoundKey

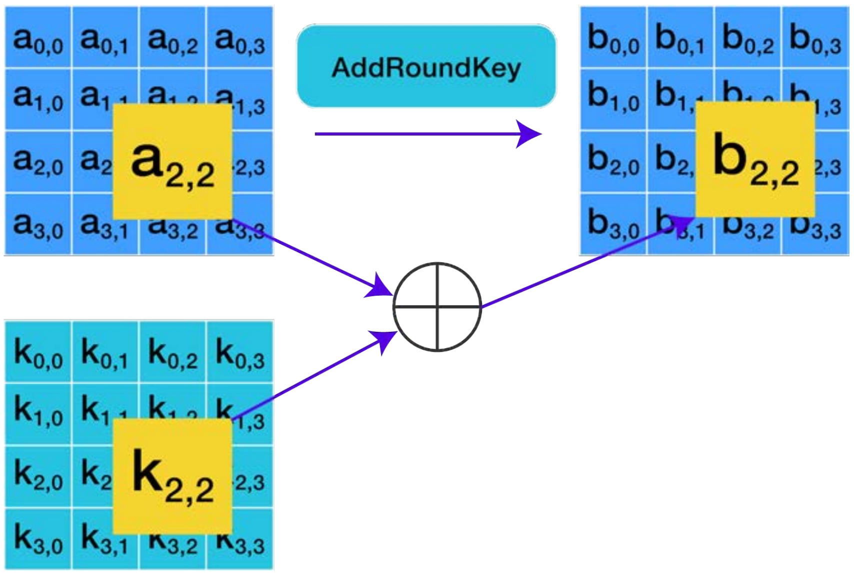
3. Repeat 13 times:
   a. a. SubBytes
   b. b. ShiftRows
   c. c. MixColumns
   d. d. AddRoundKey

4. Final Round:
    a. a. SubBytes
    b. b. ShiftRows
    c. c. AddRoundKey

[10] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf#page=23

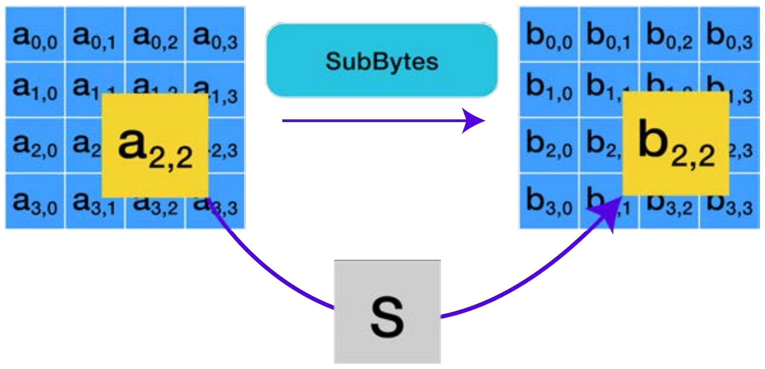
### AddRoundKey

Bitwise XOR addition of the round key to the current state of the message.
Each round key is the same size as the message.



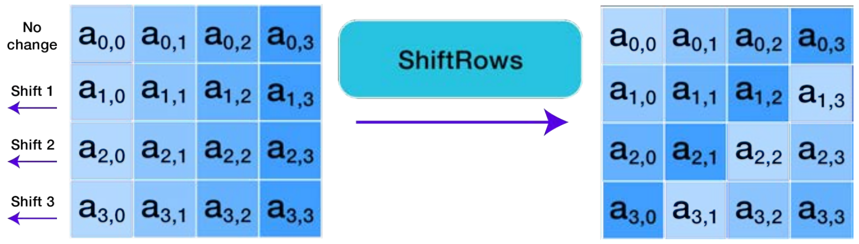
### SubBytes

Non-linear mixing of the bytes in the message according to the Rijndael S-box.[11]



### ShiftRows

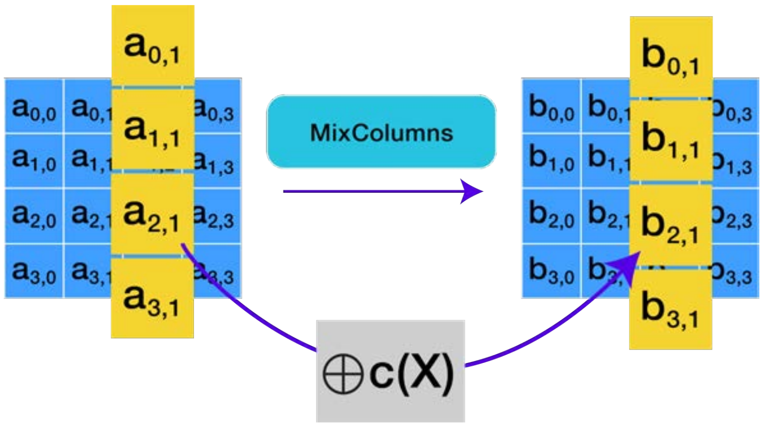
Transposition of the state's last three rows shifted left n steps, where n refers to the row index.



### MixColumns

Linear mixing of four bytes in each column. Each column is transformed to its polynomial form, where the coefficients of the cubic polynomial are defined by the values in the column. This polynomial is then multiplied (mod $x^4+1$) by the polynomial $a(x) = 3x^3 + x^2 + x + 2$.

The inverse of this polynomial, used during decryption, is $a^{-1}(x) = 11x^3 + 13x^2 + 9x + 14$.

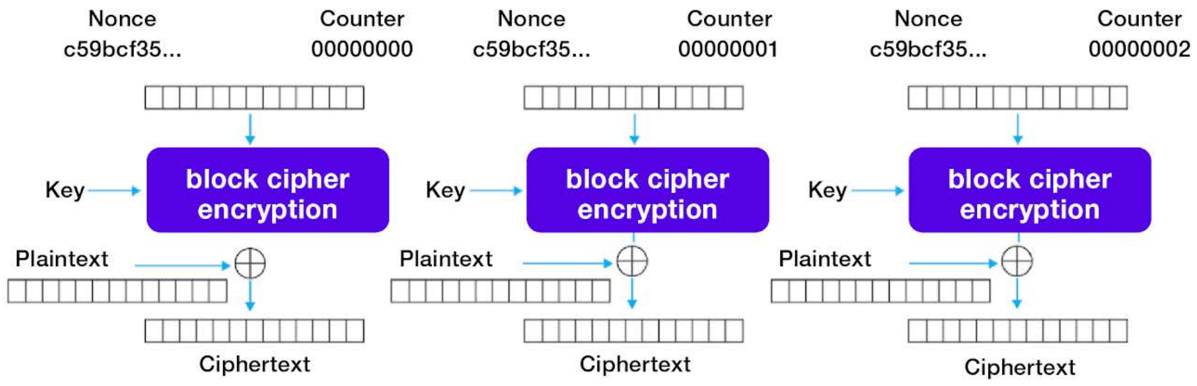


Decryption occurs by reversing the steps involved in encrypting the block.

## GCM Mode of Operation

Now that we know how to encrypt each block, we can explore how an encryption mode can convert the blocks into a stream encryption.

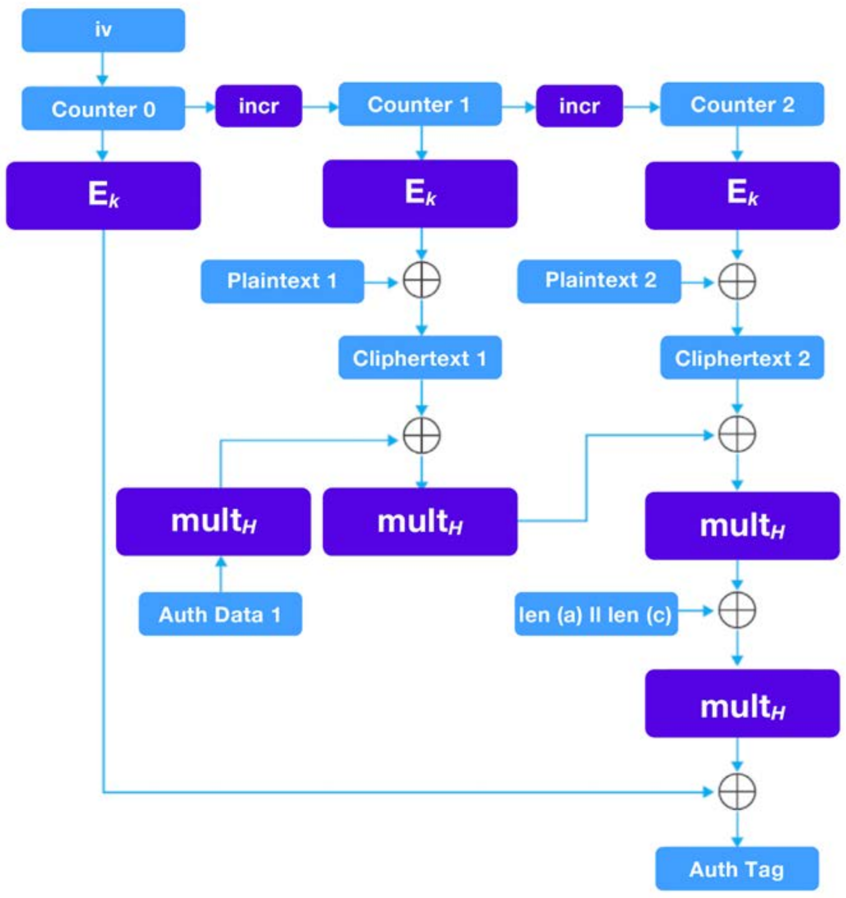
A counter mode turns a block cipher to a stream cipher. It functions by taking successive blocks of the input and encrypting them combined with a counter. The only requirement is that the Initialization Vector (Here called the Nonce) be unique. Using a non-unique IV reduces this encryption to a substitution cypher, introducing a major vulnerability. This is handled by Web Crypto in the implementation of Corvee because it employs the best random number generation available on the environment.



**Counter (CTR) mode encryption**

The difference between Galois Counter Mode formalizes this stream cipher by taking a random initialization vector (here iv), a block number (e.g., Counter 0) and encrypting them with a block cipher E (here, AES). The result of this encryption is XOR'd with the block of plaintext to output the ciphertext.

The ciphertext blocks can then be viewed as coefficients of a polynomial to evaluate at the key-dependent point H. The result of this evaluation can then be again encrypted and used as an authentication tag.

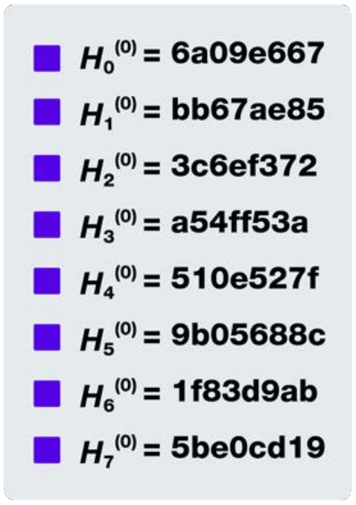


## Appendix C: SHA-256

SHA-256 refers to a set of cryptographic functions collectively termed "SHA-2," where the number in the name refers to the number of bits in the digest. SHA-256, for instance, takes in 32-bit blocks and outputs a 256-bit block.

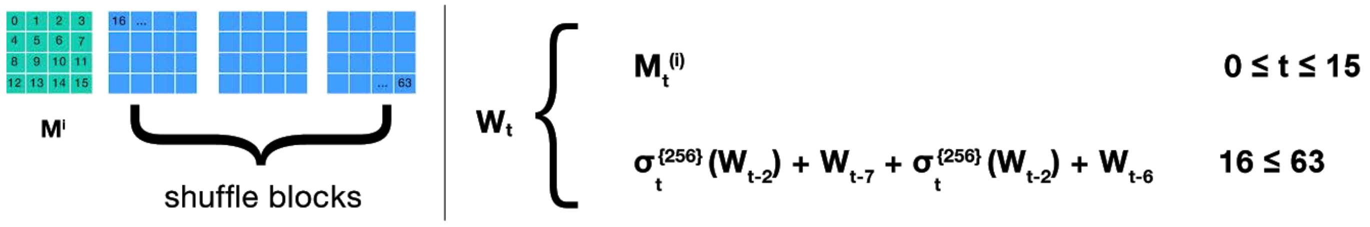
The first step[12] in calculating a SHA-256 hash is to first pad the input message such that the length of the input message is a multiple of 512, with a total length of n*512. Then, the padded input is split up into n*16 blocks of 32 bits each. The final step before hashing the message is to initialize the hash by extracting the first 32 bits of the fractional parts of the square roots of the first eight prime numbers.

[12] https://medium.com/biffures/part-5-hashing-with-sha-256-4c2afc191c40



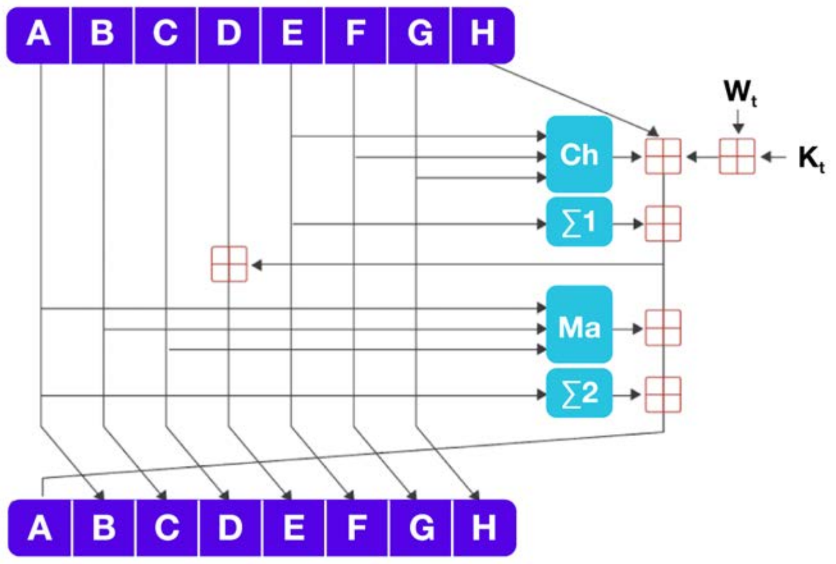
- $H_0^{(0)} = \text{6a09e667}$
- $H_1^{(0)} = \text{bb67ae85}$
- $H_2^{(0)} = \text{3c6ef372}$
- $H_3^{(0)} = \text{a54ff53a}$
- $H_4^{(0)} = \text{510e527f}$
- $H_5^{(0)} = \text{9b05688c}$
- $H_6^{(0)} = \text{1f83d9ab}$
- $H_7^{(0)} = \text{5be0cd19}$

A message schedule is then created by splitting up the padded input message into 512-bit input blocks (each made of 16 32-bit integers). For each block, we generate three additional rotations of the block:



$$W_t \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \sigma_t^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_t^{\{256\}}(W_{t-2}) + W_{t-6} & 16 \leq 63 \end{cases}$$

Each of the 64 resulting integers in the block schedule is then processed as summarized by this graphic to produce the hash of the block.



[13] https://en.wikipedia.org/wiki/SHA-2

$$Ch(E, F, G) = (E \wedge F) \oplus (\neg E \wedge G)$$

$$Ma(A, B, C) = (A \wedge B) \oplus (A \wedge B) \oplus (B \wedge C)$$

$$\Sigma_0(A) = (A \ggg 2) \oplus (A \ggg 13) \oplus (A \ggg 22)$$

$$\Sigma_0(E) = (E \ggg 6) \oplus (E \ggg 11) \oplus (E \ggg 25)$$

The red ⊞ character refers to addition mod 2.$^{32}$

After each of the 64 integers has been processed, the result is added to the previous hash and modulo 2.$^{32}$

The result after passing through all the blocks is the SHA-256 hash of the message.

## Appendix D: HMAC

HMAC is a message authentication code that uses both a hash function and key to calculate a hash signature that will verify a message's authenticity and integrity. A MAC will provide a verification of its progenitor message's authenticity and integrity by calculating a signature based on the contents of the message, which can then be verified by the receiving party to prove the message has not been tampered with during transit and that the sender is indeed who they say they are.

While attaching a MAC to a message serves to prove authenticity and integrity, it is not by itself enough to stop data interception, and some vulnerabilities mean that additional steps must be taken to protect against data tampering or interception. An HMAC does NOT encrypt the message. This must be done separately. The HMAC only serves to verify the message that has been sent, encrypted or not. This implies that HMAC is only meant as a complement to security, not additional data security.

The HMAC addresses this issue by using two passes of a hash computation. A secret key is first used to derive two keys: an inner and an outer key. The inner and outer keys are padded to match the block size of the message. The inner key is then concatenated to the message and the result is hashed to produce the inner hash. The outer hash is then concatenated to the inner hash and then again hashed to produce the actual HMAC.

This double hashing makes the execution of a length-extension attack — where the attacker appends data to the message without affecting the MAC — as well as a hash collision less likely. Using HMAC is effective enough that there are no extension attacks that can spoof a valid MAC.