Welcome to Expressive Cryptography

Expressive Cryptography ft. Brainbox is an interactive GUI where the user can implement and play around with different cryptographic primitives, using which various cryptographic protocols can be built. This extended GUI interface simulates primitives like hashing (eg., SHA256), HMAC, AES and Elliptical Curve. The following lessons will walk the user through different complex schemes that are pre-built, explaining the implementation and functionality of each of them. This GUI does not require any extensive prerequisite knowledge of cryptography or programming. It is designed in a way that allows the user to drag-and-drop different components, connect them and view the resulting output of their custom design.

In the following sections, we will also give you a brief background on the cryptographic functions/primitives that are implemented and how each component will affect the respective protocols that are built using these functions. The protocols implemented are:

1. Password-based registration and login protocol
2. Authenticated symmetric cryptography
3. Authenticated hybrid cryptography

Password-based registration and login protocol

The password-based registration and login protocol takes username, password pairs as input and stores them in the database. This protocol mainly uses a hashing function that hashes the passwords before storing.

Hashing

When a password has been “hashed” it means it has been turned into a scrambled representation of itself. A user’s password is taken and – using a key known to the site – the hash value is derived from the combination of both the password and the key, using a set algorithm. To verify a user’s password is correct it is hashed and the value compared with that stored on record each time they login.

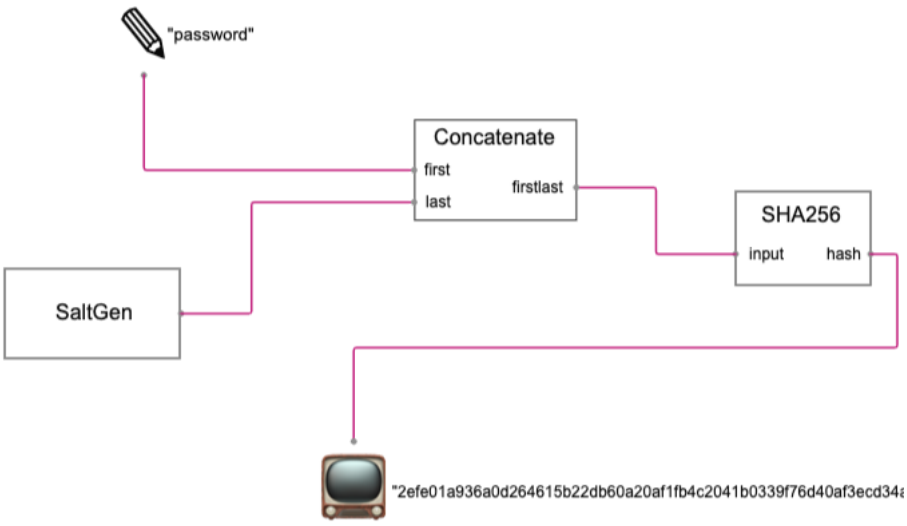
The algorithm used in our implementation is SHA256. Secure Hash Algorithm SHA256 (belonging to the family SHA-2) is cryptographic hashing algorithm originally design by the US National Security Agency. It generates 256-bit hash value that is typically rendered as a 64-digit hexadecimal number. This algorithm can essentially handle a data string of any size as input and renders the resulting hash impossible to predict, making it a solid means of securing said data. SHA256 is generally recommended for secure hashing.

In our implementation, the **SHA256** block takes a string, eg., "thisisapassword", which returns the hash of the respective input string. To increase the security of the passwords stored, we use the **Concatenate** block to concatenate the plaintext password with a random salt value generated from **SaltGen** before being passed to the Database for storing.

Salting

Passwords are often described as “hashed and salted”. Salting is simply the addition of a unique, random string of characters known only to the site to each password before it is hashed, typically this “salt” is placed in front of each password or after it. The salt value needs to be stored by the site, which means sometimes sites use the same salt for every password. This makes it less effective than if individual salts are used. This is the reason why our **SaltGen** block uses a random string in every iteration/ every password stored.

The SaltGen block is implemented with the help of a random number generator.

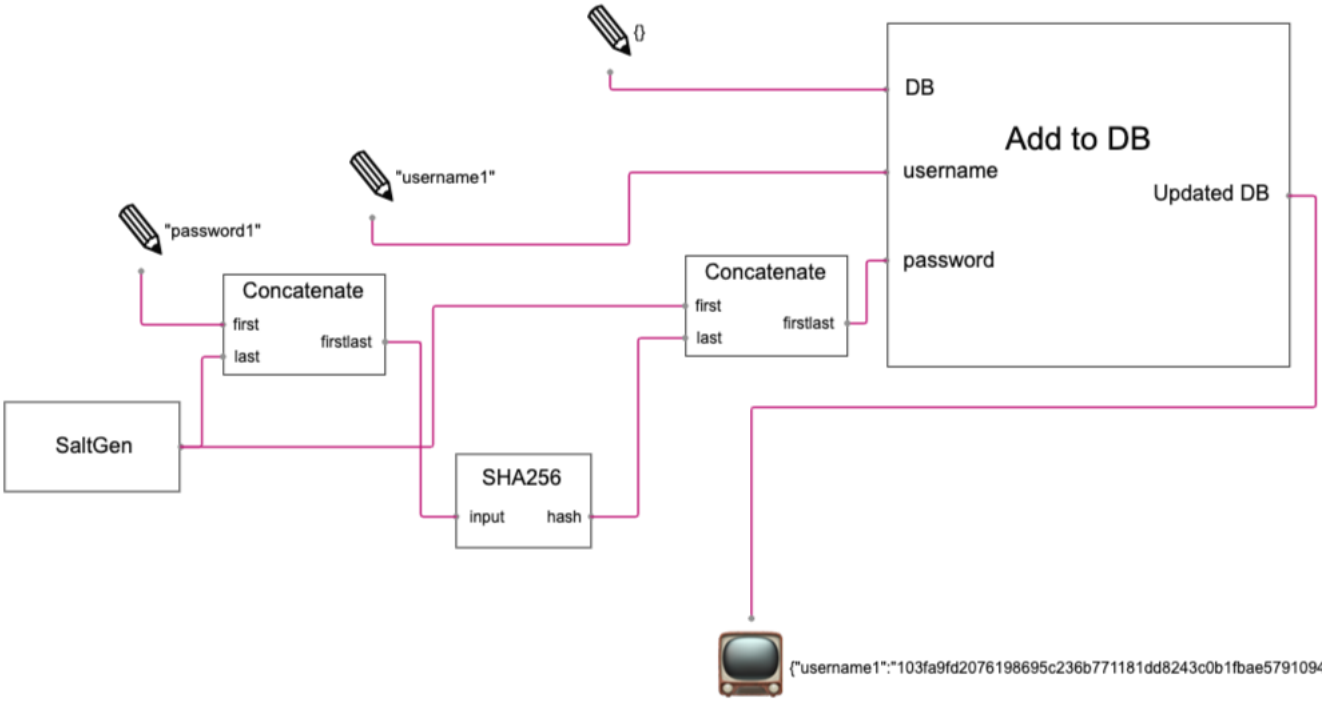


The above diagram shows a salt being concatenated with the password before being passed to the SHA256 block, which hashes the password+salt string. The resultant hexadecimal version result of the hash is showed as output. The following piece of code is what implements the hashing block:

```
var bitArray = sjcl.hash.sha256.hash(input);
var hash = sjcl.codec.hex.fromBits(bitArray);
```

Storage of {username, password} pairs in the Database

Once we have the hashing set up, as shown above, we can use the Database, which is the **Add to DB** block to store these values. The implementation of this is as shown below:-

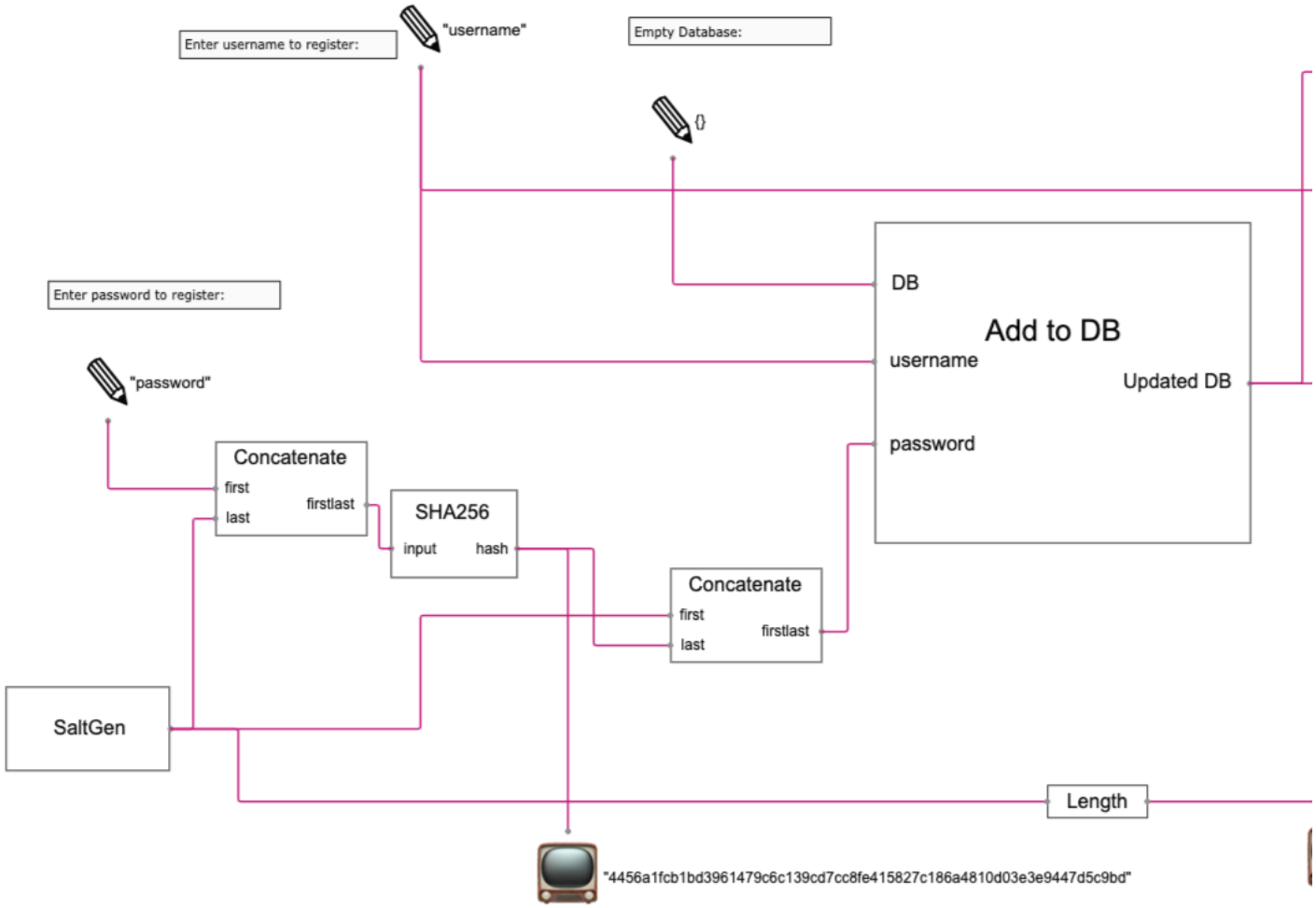


Here, as we can see, for the first user, the DB input is set to be empty. This is the basic registration step. The password is concatenated with the salt before hashing. To store the value, the salt is stored along with the hash in the DB. Moving forward, for every new user, the DB grows with new {username, password} pairs. The most important thing top make note of here is that the salting of passwords before hashing proves to be more secure for storage.

Verifying the login credentials for existing users

In case of existing users, the username is checked within the **Retrieve from DB** block before adding/storing in the DB. For returning users, login is allowed only in the case where the hash of respective password entered versus hash of password stored in the DB proves to be the same.

The following implementation of the login circuit demonstrates how the password-based login protocol works.



We start off with an empty database and add the {username, password} during the registration step, which is the first part of this implementation. Once we have the database with a non-zero number of entries, we move onto the login step. In this step, we retrieve the hashed password given the registered username. The hashed password is then compared to the password given as input by the user during login. This password is concatenated with the same salt before hashing to compare. If the two hashes are equal, the user is given access via login. Else, it returns false and login cannot be completed.

Authenticated Symmetric Cryptography

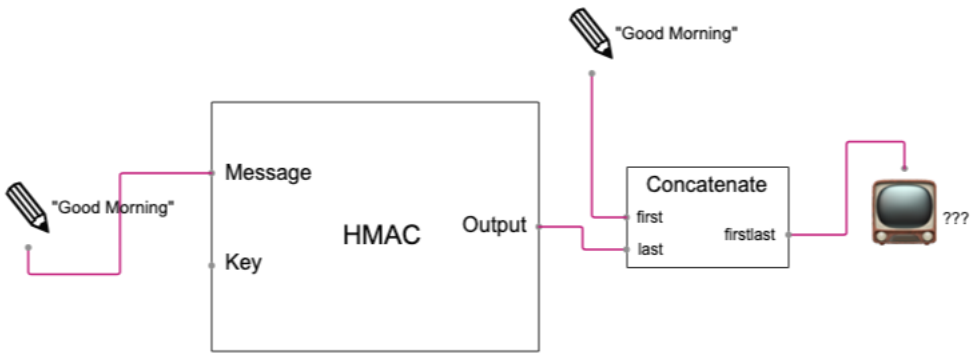
What is AES?

- AES is a type of symmetric encryption, meaning the two users, often called Alice and Bob, or the client and the server, share one key which is used for encryption and decryption. AES is a block cipher which is based on a design principle called a substitution-permutation network. This type of encryption scheme uses several rounds of different permutations and substitutions to create a cipher text, which is completely unidentifiable with regards to its plaintext.
- We chose to use AES in CCM (Counter with CBC-MAC) mode to add the extra benefit of authentication and integrity to each message. Two primitives are combined in this protocol, a MAC and Encryption. First a CBC MAC of the message is generated, which we attach to the message as a tag. The message and the tag are then encrypted using AES CBC mode, resulting in an authenticated encrypted message!

In this lesson, we will illustrate AES as well as how a MAC works separately to show the inner workings of CCM mode. The MAC we will be demonstrating is HMAC. Although HMAC isn't the exact type of MAC used in AES, its functionality is similar and will help demonstrate its use within the protocol.

All of the cryptography used in these blocks are done using the SJCL (The Stanford Javascript Cryptography Library).

Implementing HMAC



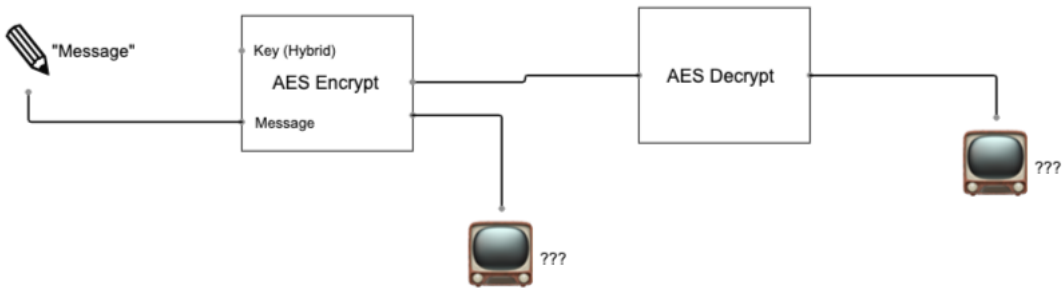
The HMAC block we built has the option to provide a key, or to generate one. In this demo, the HMAC block generates its own key. The user provides a message and the block, using the message input and generated key runs this line of code:

```
new sjcl.misc.hmac(key, sjcl.hash.sha256)).mac(message);
```

This returns a MAC, and in this particular case, it is created using a type of SHA256 that uses a key. The original message is then concatenated with the MAC and shown on the screen as an authenticated message.

AES in CCM mode

The inner workings of AES - CCM mode do something very similar, but with a CBC-MAC instead.



- As seen in the image above, the AES block takes in a message. Internally, a MAC of the message is generated and concatenated with the original message, then encrypted. This is done using the following line of code. The phrase “password” is used to generate a random key, internally within the SJCL library.

```
cipher_text = sjcl.encrypt("password",input_one);
```

- The display at the bottom shows the encrypted ciphertext.

- The AES decrypt block then decrypts the message, which is displayed on the far right.

- AES can also be used with a key generated with Diffie Hellman. If the user chooses to do so, “password” portion of the code above is replaced with the Key that is generated by Diffie Hellman, passed through the port on the upper portion of the left most block. This is explained further in the hybrid cryptography section.

Some challenges faced when creating this cryptographic primitive were understanding the library and mostly integrating it with the brainbox GUI which was originally made to only simulate digital circuits.

Asymmetric Cryptography

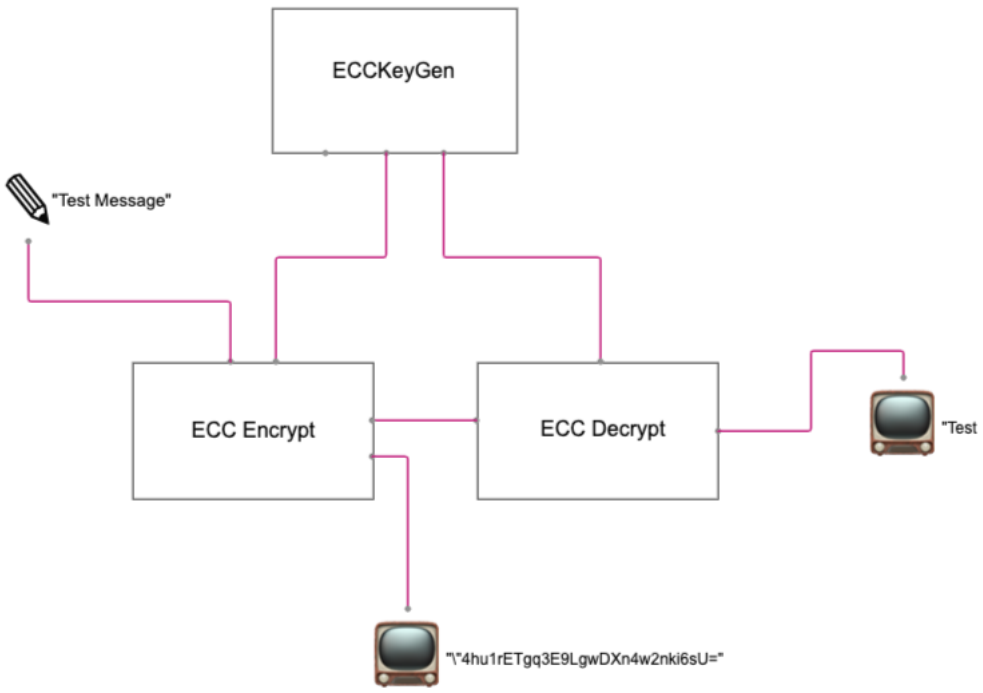
What is Asymmetric Cryptography?

- In comparison to Symmetric Cryptography that needs the same keys for encryption and decryption, this kind requires and generates a pair containing a public key and a private key. For proper security, the public key can be freely distributed while the private key should be kept secret.
- The way each of these algorithms work is that the plaintext message is encrypted using the public key, but can only be decrypted using the receivers' secret private key.
- One of the benefits of this type of cryptography is that it can be combined with a digital signature and the sender can "sign" the message with his private key before sending it. This step, however, is not a necessity.

Elliptical Curve Cryptography

In this lesson, we cover the basic structure of implementing the Elliptical Curve Cryptography (ECC) approach of Public-key (asymmetric) cryptography. This is done by using the ElGamal discrete logarithmic encryption and using elliptical curves for the same. The libraries used to implement the same were imported from SJCL (Stanford Javascript Cryptography Library). All of the code written to compose the building blocks of the GUI have been written in JavaScript as well.

Below is the screenshot of the entire circuit of ECC with a random input and the respective output. The basic communication structure thought of here is that the ECCKeyGen and ECC Decrypt blocks belong to "Alice" while the ECC Encrypt block belongs to Bob. Neither of them know each others' private keys.



- The first block that was made in this circuit was the ECCKeyGen block. This imported the ecc sub-library from SJCL and generated a pair of keys - private and public on a curve finite field composed of 384-bit primes. Three output ports are made here : One for the sending of the public key to the Encrypt function, one for the transferring of the self private key to the Decrypt block, and the third (unused here) is solely for use in the hybrid circuit which we shall see later.

- The second block is the ECC Encrypt block. This takes the input from the KeyGen block of the public key, and then takes the plaintext input given by the user here (Or by Bob, in the main case). This message is then encrypted using the received public key by ElGamal Encryption. This we decided to show as an output (the tv symbol) and display the ciphertext that is sent to the Decrypt Block as well.

- The third and final block is the ECC Decrypt block. This basically receives a private transfer from the KeyGen block of the previously generated secret key (belonging to Alice). It receives the ciphertext as input from the Encrypt block and decrypts the message. This decrypted message is then displayed and we note that the message is identical to the one sent.

Once joined using links, each of these blocks work in order to avoid errors in computation. On the first step, we see the key being generated and being sent to the encrypt and decrypt blocks respectively. The next step, the encryption is done, while the decrypt block waits to receive the ciphertext. This ciphertext is encrypted and then sent to the decrypt block where it is decrypted by (Alice's) private key and then displayed.

The library code used here was:

```
var pair = sjcl.ecc.elGamal.generateKeys(384);  //to generate key pair

var cipher = sjcl.encrypt(pair.pub, plaintext); //to encrypt the plaintext with the public key of the pair

var plaintext = sjcl.decrypt(pair.sec, c);  //to decrypt the message using secret key
```

The biggest challenges faced during this implementation were integration of the Javascript libraries and code with the Brainbox GUI. Although it is built on JS, integrating each input, output, and algorithm line with the GUI was a tough task. For debugging, the browser js console had to be used. A shared tunnel between all the members of the team had to be created to merge each change made on the components and the circuits. Several lines of library code often had to be read through for debugging.

Authenticated Hybrid Protocol

Hybrid Encryption

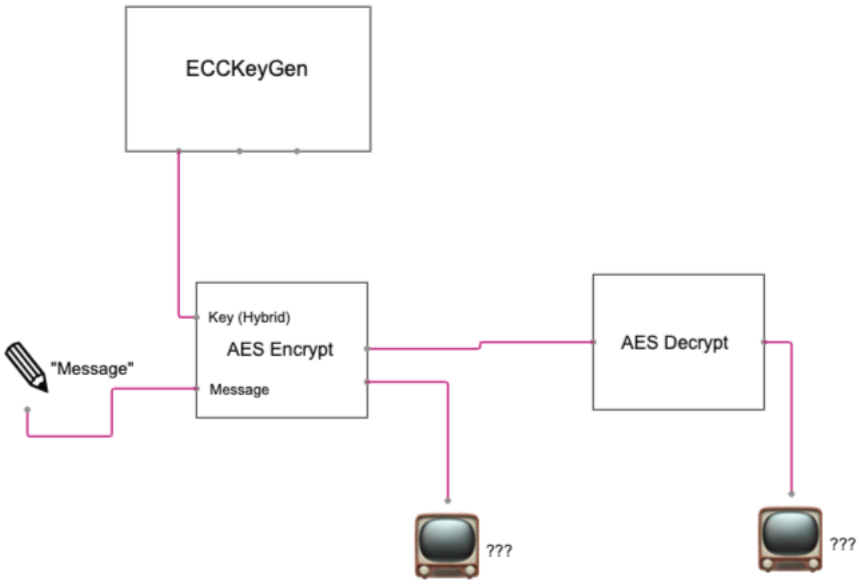
What is Hybrid Encryption?

- Hybrid Encryption is a mode of encryption that merges two or more encryption systems. It incorporates a combination of asymmetric and symmetric encryption to benefit from the strengths of each form of encryption. Asymmetric encryption having the strength of security and symmetric encryption having the strength of speed.

- The Hybrid Encryption System here uses a diffie-hellman exchange as the asymmetric cryptography to get the shared secret key and then goes to AES for the symmetric encryption.

This lesson will cover the Diffie-Hellman/AES Hybrid Encryption scheme that can be developed in this tool and how to get the appropriate outputs. All code for Diffie Hellman and AES which are the building blocks of this Hybrid scheme are implemented using the SJCL library.

Below is the screenshot of the entire circuit making up the hybrid encryption with a random input message of “Hey There”.



- The ECCKeyGen block is where the shared diffie-hellman key gets generated to be put into AES along with the message to be encrypted. Diffie-Hellman is done in SJCL with the following:

```
var pair = sjcl.ecc.elGamal.generateKeys(256);
var val = pair.sec.dh(pair.pub);
```

- The AES Encrypt block then does AES in CCM mode which is outputted using the lower right port. The ciphertext from the encryption can be seen by the TV. This ciphertext is also fed into the AES Decrypt function.

- The AES Decrypt block then decrypts the inputted ciphertext and outputs the original message which can be seen on the TV to the right.

The main challenges that were faced in the development of the Hybrid Encryption Scheme was actually getting diffie-hellman to work in the ECCKeyGen block. All other issues were related to integrating the transfer of these keys and ciphertexts with the Brainbox GUI. This is because originally Brainbox was made to simulate logical circuits.