# BIP-39 Wallet Implementation

**Kevin Chau, Jordan Mai**

**CS 168: Cryptocurrencies and Security on the Blockchain**

**San Jose State University**

## Introduction

The goal of our project was the implementation of BIP 39 into SpartanGold, creating a deterministic wallet feature. This report will go into detail about the implementation, the results, challenges, and the potential future work on this project. To demonstrate the usefulness of BIP-39, we wanted to introduce a form of fund recovery to SpartanGold. Unlike a JBOK wallet, if we were to lose access to our funds, we should be able to regain access to them with BIP-39 implemented. In addition to BIP-39, numerous other features will need to be added to SpartanGold to reach the goal of our project.

We decided on the implementation of BIP 39 because of its use of mnemonics. The user would be able to recover their funds just by remembering a 24-word mnemonic that has been generated based on a set list of 2048 words. These words were selected to avoid the occurrence of similar words in the list. The mnemonic would then be converted to a binary seed for the generation of addresses. Because the mnemonic was converted to the binary seed used for addresses, we would be able to generate lost addresses just from the mnemonic.

## Implementation

### *UTXO Model*

To facilitate the implementation of BIP-39, we worked on several other features to make SpartanGold more Bitcoin-like. The first change we needed to make to SpartanGold was to implement Bitcoin's UTXO model. SpartanGold currently uses an account-based model, which doesn't work with the concept of wallets and BIP-39. To transition SpartanGold over to a UTXO model, we took much of the implementation

from HW2 and extended it into the proper areas of SpartanGold. For example, for how transactions are created by a client, instead of deducting from one address, we gather the number of keys necessary to fulfill the transaction and append it. Part of the UTXO model implementation included adding the wallet feature which we can extend with BIP-39.

***Testing Base***

To test our implementations we needed to decide between creating a driver or using an existing file in the SpartanGold repository. Upon taking a look at SpartanGold, we decided to extend tcpminer to support our use case. This included adding support for mnemonics in the constructor and adding a mnemonic field in the JSON configuration file. We chose tcpminer because it had existing implementations for reading a JSON configuration file which was a feature we wanted to extend on. We also liked how it was interactive and allowed you to play around with things like creating and sending transactions. This made it the perfect place to implement our fund recovery function, which will be discussed below. Besides extending tcpminer, we also had to fix various issues surrounding the script. This included updating various functions due to deprecation and fixing bugs to stabilize the script. These fixes will be further elaborated on in the “Challenges” section.

***Mnemonic Generation***

Now that SpartanGold is more Bitcoin-like we needed a way to be able to generate the mnemonics we’re going to use. This was done through a slightly modified version of mnemonic.js from lab 10. With the mnemonic generation implemented, we need to connect it with tcpminer to be able to generate mnemonics for new users. To

achieve this, we implemented a helper function in utils.js that connects with mnemonic.js. Utils.js was the best place to implement the generateMnemonic function because of how accessible it is throughout the whole project. This would allow mnemonics to be easily implemented in other files for testing without having to connect to mnemonic.js. To utilize this helper function, a file just needs to be connected to utils.js and call the generateMnemonic function which would return a 24-word mnemonic for use.

***Deterministic Address Generation***

The next step of our implementation was to generate the addresses for the clients and miners to use. Since we're implementing BIP-39, one of the requirements is that addresses are deterministically generated based on a seed derived from a mnemonic. As for how the keypairs/addresses are generated deterministically, we'll cover that more in-depth in the "Challenges" section of the report. Every time a client/miner object was created, it would read the mnemonic passed in through the constructor and create the appropriate seed value using the BIP39.js library. This seed would be passed into a pseudo-random-number generator instance to create a deterministic number generator. This PRNG instance would finally be passed into the generate keypair function from the PKI library to generate the keys to use in transactions. These keypairs are then converted to addresses and stored in each client/miner's wallet.

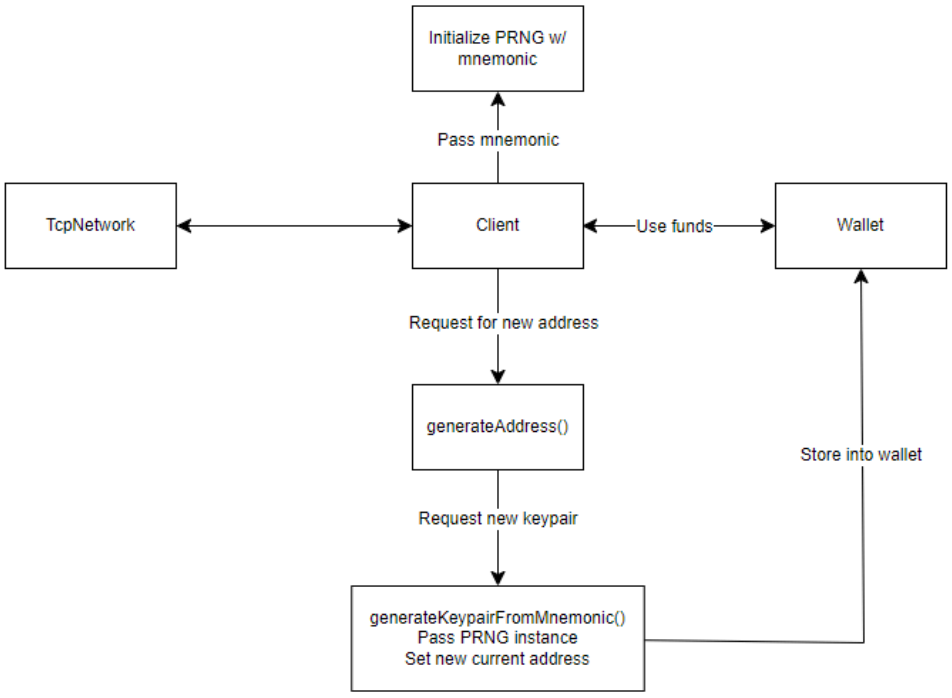


*Figure 2.4.1 How deterministic key generation fits into our project*

Figure 2.4.1, gives you an overview of what our main goal for implementation is for our project. With the mnemonic, we can pass it into any object that is or extends the client object. We use that mnemonic to generate deterministic keypairs for the client to use as addresses on the blockchain/TcpNetwork.

***Fund Recovery***

The last major feature that we implemented for this project was the fund recovery feature. As we mentioned above, one of the key benefits of BIP-39 is the ability to regenerate your keys using a mnemonic. Being able to regenerate the same keys means that you can recover your funds in the event that you lose access to your wallet.

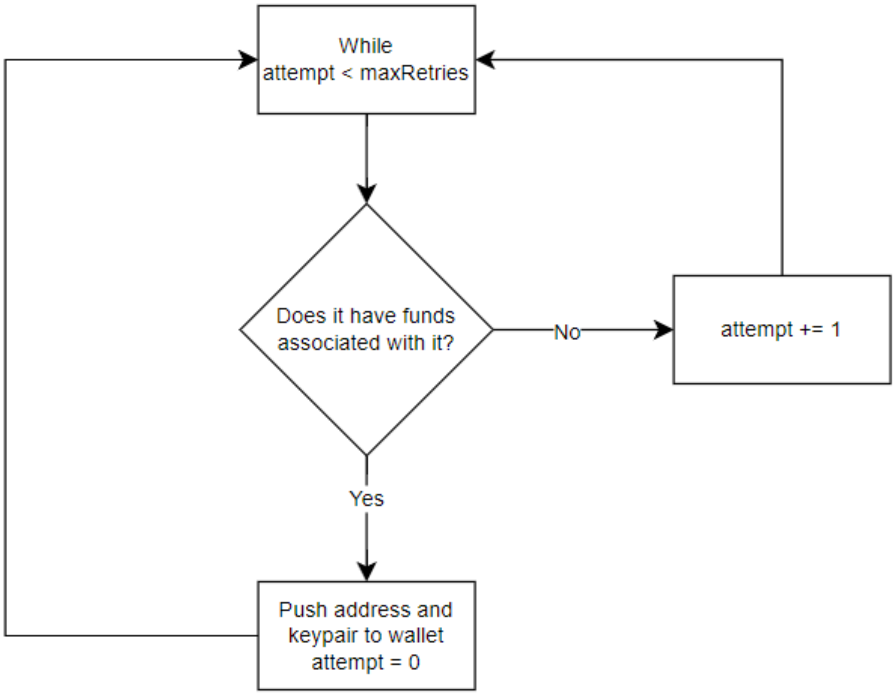


*Figure 2.5.1 Flow chart of fund recovery function*

To implement this function, we took the naive approach of brute-forcing each address to search for funds. The figure above illustrates our brute force methodology. To account for gaps between funded addresses in the address space, we add a max retry parameter that the user can specify. This max retry sets a limit on how many unfunded addresses it'll check before it stops trying to search for addresses with funds. If the program encounters an address with funds, this max retries counter is reset to account for another potential gap in the address space.

**Results**

By putting together all the features that we implemented, we are able to effectively complete what we set out to accomplish. The generation of the mnemonics worked as intended when passing in a config file that didn't have a mnemonic already included. This mnemonic was then used to deterministically generate addresses for the

client's wallet. Figure 3.1.1 shows how the addresses were generated and added to the client's wallet.

```
generateAddress(){
  this.keyPair = this.generateKeypairFromMnemonic();
  this.address = utils.calcAddress(this.keyPair.public);
  this.wallet.push({ address: this.address, keyPair: this.keyPair});

  return this.address;
}
```

*Figure 3.1.1 Generate address function using mnemonic*

With the addresses being deterministically generated and the use of a function called mnemonicToSeedSync from the BIP39 library, we were able to implement fund recovery into spartanGold. Figure 3.1.2 shows where the fund recovery function is when tcpminer is ran.

```
Starting Minnie

 Funds: 82
 Address: WHGUPctffqaWqXm4freSlZKmslozI+45EBVq3AoRbgE=
 Pending transactions:

 What would you like to do?
 *(c)onnect to miner?
 *(t)ransfer funds?
 *genereate new (a)ddress
 *(r)esend pending transactions?
 *show (b)alances?
 *show blocks for (d)ebugging and exit?
 *show all UTXO balances
 *(f)und recovery
 *(s)ave your state?
 *e(x)it without saving?

 Your choice:
```

*Figure 3.1.2 Fund Recovery*

To outline the fund recovery function, figure 3.1.3 highlights how there is an inaccessible wallet that belongs to the client. The address' funds aren't being reflected in the client's funds. To fix that, we would have to run the aforementioned fund recovery function which would prompt the user for a number. This number would correspond to the number of tries to attempt to recover the address.

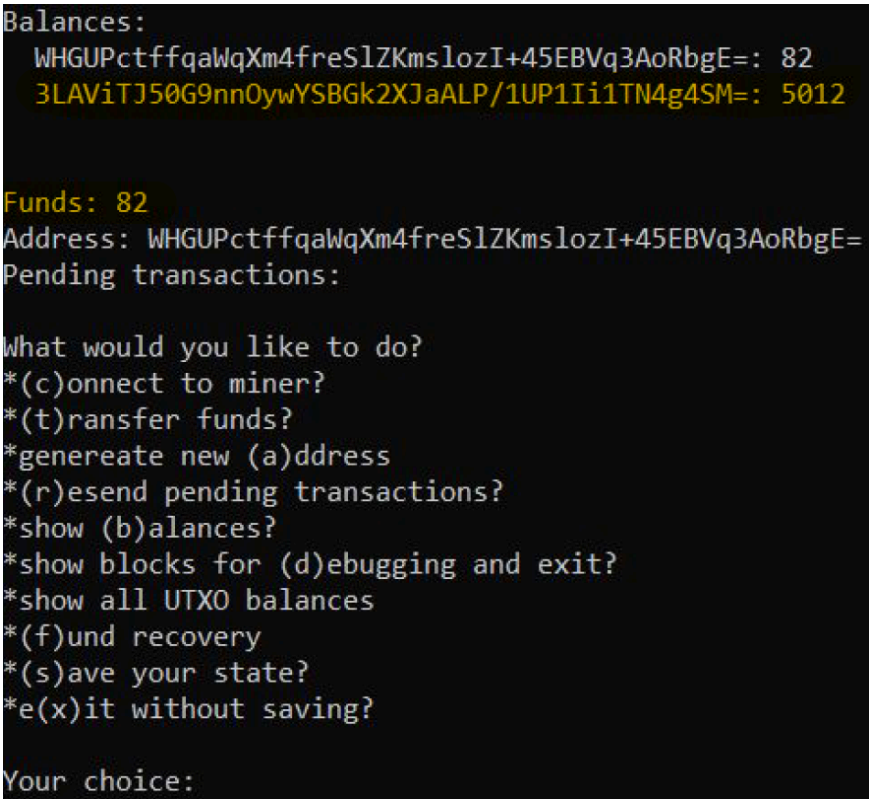


*Figure 3.1.3 Inaccessible Wallet*

The number of retries inputted into the recovery function would vary. There may be moments, where the amount of retries the user input wouldn't be sufficient to recover the inaccessible funds. However, as highlighted in figure 3.1.4, we were able to successfully recover our funds within 10 retries. The funds would then be reflected in the client's overall funds.

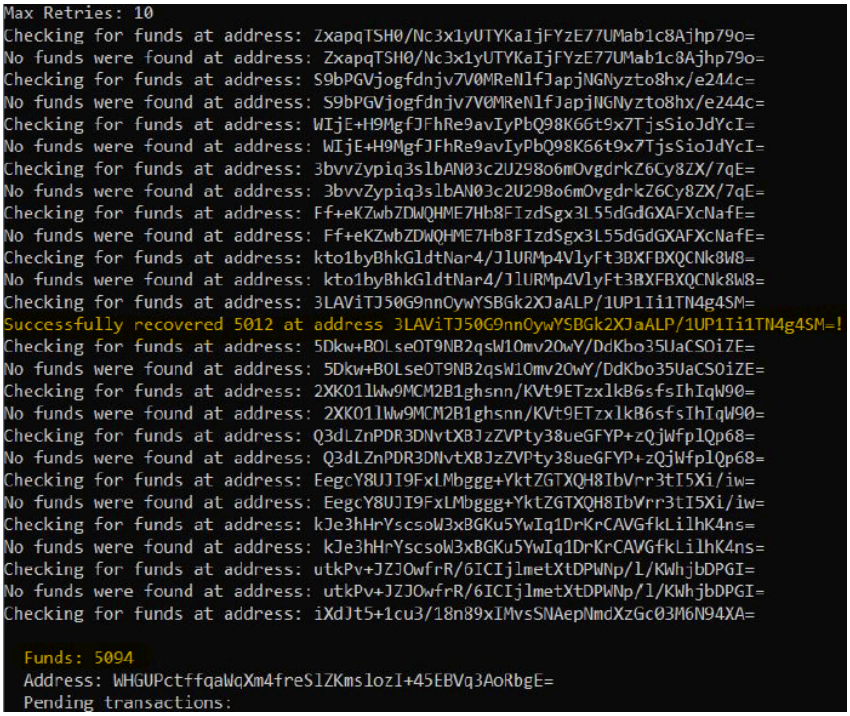


*Figure 3.1.4 Successfully Recovering Funds*

## Challenges

### *Deterministic Key Generation*

The most important challenge we had to overcome to make this project successful was to research and implement a way to generate keypairs deterministically. As mentioned before, we found that the three core functions that made it all work were, the mnemoncToSeedSync function from the BIP39.js library, the PRNG instance, and the PKI library function to generate the keypairs. In the beginning, we searched through the internet and the SpartanGold repo, examining how we might implement this feature. Luckily, we found that some work had already been done on the implementation of BIP-39 in the utils.js file in the SpartanGold repo. Now, we had to figure out how this function works and how we can make it work for the goal of this project. With a bit of debugging and reverse engineering, we figured out the three important functions that

we mentioned above. The mnemoncToSeedSync function was in charge of transforming the 24-word mnemonic into a binary seed. We could then take this and pass it into the PRNG instance. The PRNG instance is the key piece to the entire puzzle. With a seed, the PRNG instance can now deterministically generate the same data every time. We can now pass this instance into the PKI function that uses the PRNG instance to generate keypairs, giving us deterministic keypair generation.

***TcpMiner***

Part of the goal of the project was to get our BIP-39 implementation working with tcpminer. When we initially began the project, we started with trying to get tcpminer up and working. Upon the first run, we realized that it was broken and would not work at all. It was confusing since we've been using tcpminer in our homework and labs so we thought it would be a reliable testing base. After doing some research, we found out that the SpartanGold version we were using was completely different from the version of SpartanGold on the GitHub repository. This gave us a hint as to what was going on and where our bugs and issues could be. After breaking down each function that was being called in tcpminer, we found that the issue was with how the genesis block was being created for the blockchain. In the old version of SpartanGold, the makeGenesis() function was used to create the genesis block. But, in the latest version of SpartanGold, you use the makeInstance() method, passing in the same configurations as before.

Unfortunately, that was not the only issue with tcpminer. The next problem we had to tackle was how to get two miners/clients to connect with each other. Similar to the previous issue, they connected fine without any issues in our labs and homework. However, after making numerous changes to the codebase, tcpminer would not work

with multiple clients. The odd part with this problem was that it would work about 5% of the time. After hours of debugging, we narrowed it down to a race condition issue. To provide some context, each miner keeps track of other miners that have connected by getting their wallet's initial address and registering them as known. The problem is when a new miner connects, it begins mining for blocks before receiving the missing blocks from the initial miner. While mining, the new miner collects coinbase transactions for each proof and, as a result, generates a new address each time to collect it. Eventually, the new miner realizes that it's missing blocks from the blockchain, so it sends a request to the initial miner for the missing blocks. When the initial miner receives this request, they check their registered client list and don't see the new miner's address since it has changed due to mining. This results in a socket error, crashing the application. We hypothesized that tcpminer worked on rare occasions likely due to the new client being able to receive the missing blocks before their address changed from mining. To combat this race condition, we implemented a 5000 ms delay before a miner began so they could request and receive the missing blocks without any address issues.

**Future Work**

Our implementation of fund recovery wasn't the best approach reflecting back on it. Although it worked well enough, the method was both naive and inefficient. Given more time, we would've done more research into how fund recovery works in Bitcoin, specifically in regards to how it deals with edge cases like gaps in the address space. Our solution was to set max retries and stop attempting to recover after reaching it. It was also inefficient and wasted address space as well. Due to how fund recovery worked, if it searched an address that had no funds, it would discard the address. Due

to how the PRNG instance worked, the address could never be used again. Although it doesn't greatly impact the experience, it seems wasteful and opens up the potential for bugs.

The other improvement we would like to make is the structure of the code and how we implemented our features. Similar to the homework, any extension of the code was done through a separate file, utilizing the inheritance feature. This helped avoid clutter and improve the readability of the code. Although complexity is not inheritance's strong suit, having all of our new implementations in new files would have made it easier to debug and make any changes. This would have greatly sped up development and reduced constant jumping around. In its current state, we didn't want to do too much refactoring or restructuring out of fear that it may completely break our working project.

## References

bitcoinjs. (n.d.). *bitcoinjs/bip39: JavaScript implementation of Bitcoin BIP39: Mnemonic code for generating deterministic keys*. GitHub. Retrieved May 8, 2024, from https://github.com/bitcoinjs/bip39

Palatinus, M., Rusnak, P., Voisine, A., & Bowe, S. (2013, September 10). *bips/bip-0039.mediawiki at master · bitcoin/bips · GitHub*. GitHub. Retrieved May 8, 2024, from https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki