Splitcoin – Manual Seed Phrase Encryption

Freddie Ranieri
May 31, 2023

"All men dream, but not equally. Those who dream by night in the dusty recesses of their minds, wake in the day to find that it was vanity: but the dreamers of the day are dangerous men, for they may act on their dreams with open eyes, to make them possible."
– T.E. Lawrence

Abstract

We present Splitcoin, the safest way to protect your seed phrase. The three pillars of self-custody are described in what we call the Self-Custody Trilemma: Security, Distribution, and Connectivity. Splitcoin uses our proprietary SPLIT39 protocol, which allows the user to manually encrypt and decrypt their seed phrase using an auto scrolling codebook that is securely displayed in an iOS/Android mobile application. The app communicates with a set of data storage devices such as NFC tags or QR codes that are used to store and retrieve an encrypted key. These are called coins. The key securely stored on your coins unlocks the complete two-way codebook, which contains pages (unique codebooks) for each seed word. The key to the codebook is protected by a secure mode of authenticated encryption, AES-256-GCM, which is virtually impenetrable by brute-force. Each coin stores an encrypted key shard. To reconstruct the key, all encrypted key shards must first be decrypted with the user's password. Splitcoin was designed to comply with seed phrase storage best practices: never share your seed phrase with anyone, never store your seed phrase online, and never enter your seed phrase into a device that is not a hardware wallet. There are two main processes in the software: Create Vault and Open Vault.

Create Vault performs NFC write functions via the mobile application to store an encrypted, key shard on each physical Splitcoin within the coin set. These key shards are combined into a key that unlocks a unique codebook of word/code pairs that the user will use to encrypt their seed phrase by hand. Open Vault retrieves these key shards by performing NFC or QR read functions via the mobile application. These key shards are combined into a key that unlocks a unique codebook of code/word pairs that the user will use to decrypt their encrypted seed phrase by hand. The major claim is that without all Splitcoins and the user password, the manually encrypted seed phrase cannot be decrypted. The app will never have any knowledge of the user's seed phrase or ask for it.

A basic understanding of cryptocurrency and self-custody is a prerequisite to understanding the rest of this paper. Understanding concepts such as private keys, seed phrases, key derivation functions, and encryption are critical to recognizing the value behind Splitcoin. Familiarity with old-fashioned solutions for storing private keys will further enhance this understanding. Cryptocurrency is transforming the world as we know it. It is up to us to continue to safely venture forth into this new frontier, while concurrently protecting our financial privacy and independence. Be a dreamer of the day knowing that your seed phrases are secure because of Splitcoin.

TABLE OF CONTENTS

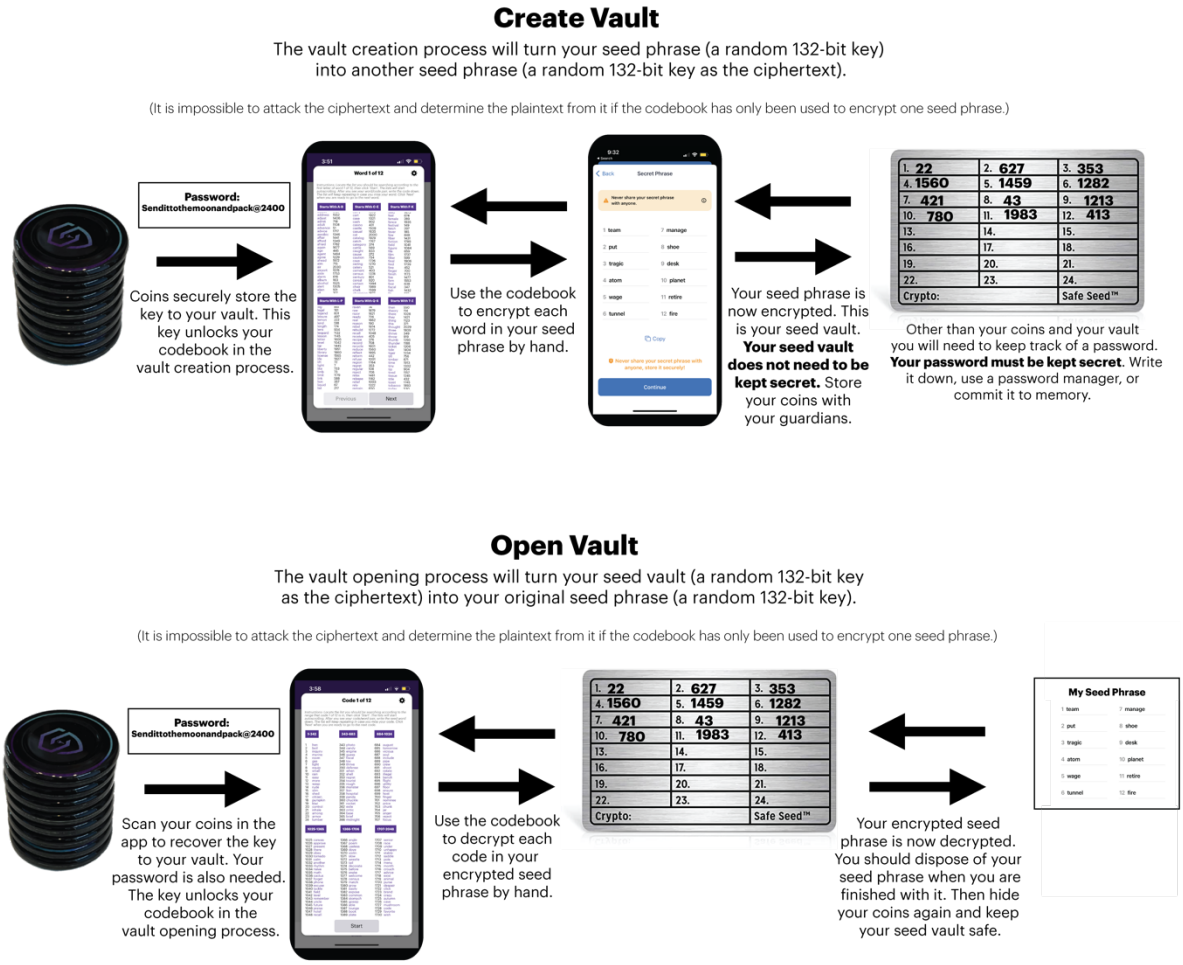
1. Introduction

A private key is like a password that grants access to your cryptocurrency funds. It is a key that consists of random letters and numbers. The generation of private keys is made possible by the usage of cryptographic algorithms based on mathematical problems to produce a one-way function. A seed phrase, also referred to as a "recovery phrase", is a 12-to-24-word code that is used as a backup access mechanism to a cryptocurrency wallet or associated private key. The seed phrase matches information stored inside the corresponding wallet that can unlock the private key needed to regain access. Bitcoin Improvement Proposal, BIP39, allows for the generation of a 12-to-24-word seed phrase from a dictionary of 2048 words [20]. With the recovery phrase, users can regenerate a wallet that has become lost or damaged. Because they hold such access, BIP39-enabled recovery phrases should be kept secret and stored securely. So, how do we do this?

Securely storing these seed phrases can be difficult, especially for people who are not technologically savvy. Cryptocurrency is growing in popularity, and new wallets are being created every day. Whether these are wallets for a single blockchain or DeFi wallets to store multiple cryptocurrencies and NFTs, users need a simple and secure way to store and back up their seed phrases. Traditional self-custody involves simply keeping a 12–24-word seed phrase secure. Best practices recommend that this seed phrase be stored offline, never be shared with anyone, and never get entered into a device that is not a hardware wallet. However, storing a word list on a piece of paper or a piece of metal is not secure. If that seed phrase is found, then your funds can easily be stolen. Backups stored as plaintext can create more risk of your seed phrase getting stolen.

Splitcoin is an offline vault for your seed phrase that does not rely on any trusted third parties. The key to your vault is stored securely across a set of physical, electronic coins or QR codes. This key unlocks a unique codebook, and the mobile application shows you how to create your vault by transforming your 12–24-word seed phrase into a new seed phrase of the same length, which is called your seed vault. Consumer behavior feels familiar with Splitcoin. This seed vault can still be stored with traditional metal or paper solutions, but it does not need to be kept secret. It reveals nothing about your seed phrase if it is found. The only way to transform your seed vault back into your seed phrase (to open your vault) is to scan your coin set and enter your password into the app. The app will unlock your codebook again and show you how to transform your seed vault back into your seed phrase by hand.

The process of setting up your coin set and transforming your seed phrase into your seed vault is called "Create Vault" in the mobile application. The process of scanning in your coin set and transforming your seed vault back into your original seed phrase is called "Open Vault" in the mobile application. You can also use Splitcoin Tools in the app to make backups of your coins. You can make copies of your coins, export them as QR codes, or export your codebook as a PDF. The diagrams below illustrate how the two main processes work when using the Splitcoin mobile application.



2. The Self-Custody Trilemma

"The Winklevosses came up with an elaborate system to store and secure their private keys. They cut up printouts of their private keys into pieces and then distributed them in envelopes to safe deposit boxes around the country, so if one envelope were stolen the thief would not have the entire key."

"How the Winklevoss Twins Found Vindication in a Bitcoin Fortune" [23]

-Nathaniel Popper, New York Times, December 19, 2017

2.1 Security

The first pillar of the Self-Custody Trilemma is security. There are three things that need to be stored with Splitcoin: your seed vault, your password, and your coin set. We define a seed vault as anything that stores your encrypted seed phrase. Your encrypted seed phrase can be stored as a list of 12-24 numbers or 12-24 words. Essentially, it mimics the characteristics of a traditional BIP39

seed phrase. This means that you can use any traditional seed phrase storage medium as a seed vault, such as writing it down on paper or using popular metal solutions. Metal solutions such as Cryptotag, Billfodl, and Hodlr One are highly recommended. As an affordable solution that is a clear improvement to a paper solution, Splitcoin includes a free magnet vault, which can be found on the bottom of your Splitcoin tin or in the packaging for Splitcoin Tags. However, third-party metal solutions are highly recommended.

The second thing that needs to be stored is your coin set, which stores the encrypted key that accesses your codebook used to encrypt and decrypt your seed phrase by hand. This is referred to as your vault key. A single set of Splitcoins (two to eight coins) are NFC tags made of a durable, acrylic material that is seventeen times stronger than glass. The coins are water resistant and will even scan underwater. These coins are dedicated storage devices and can never be mistaken for an external hard drive that can easily be written over or erased. Data is locked to your Splitcoins (comparable to burning data to a CD) so the data cannot be overwritten. This makes your coins read-only. Your vault key stored across these coins is protected by 256-bit encryption that can defend against the strongest of malicious attacks.

The third thing that needs to be stored is your password. This is the only thing that must be kept secret. This password will be used to encrypt the vault key that is stored across your coin set. It is recommended to either store this password separate from your seed vault, use a password manager, or commit it to memory. If this password is lost, the Splitcoin app cannot help you recover it. The Splitcoin app does not store any user data. The primary purpose of this password is to provide a secure communication channel between the mobile app and the Splitcoins. However, the system is designed so that if every coin and the seed vault were stolen, they would be useless without the password.

It is critical that the physical and digital security of any secret storage device be maintained without compromising the other two pillars. Should the storage device fail physically, the secret is unrecoverable (unless you have backups). Potential physical failures could be due to materials of poor durability, losing the storage device, or outside forces such as weather, fires, burglary, etc. Should the storage device fail digitally, there is also an unrecoverable data loss. An example of this would be writing over a flash drive storing your seed phrase or deleting it from your cloud storage.

In the event of a lost or stolen coin, Splitcoin provides a few different backup options. The first option is to simply use a backup set of physical coins. The Copy tool makes it easy to create multiple backup sets of your coins, which are interchangeable across sets. Additionally, Splitcoin Tags can provide an easy way to maintain backup sets. They use the same NFC technology as the acrylic Splitcoins, but they look like stickers with a 25mm diameter. You can peel and stick these tags to any non-metal real-world object such as books, picture frames, furniture, collectibles, etc. The possibilities are endless for real-world objects to store the tags with. They use military-grade encryption and can be stored practically anywhere. Digitally, the ICODE SLIX2 RFID chip boasts a data retention period of 50 years, meaning the data is never erased (in a reasonable lifetime). In addition to physical backups of your coins, Splitcoin also lets you export your coin set as QR codes in a PDF. Each coin has its own QR code. These can remain physical backups by printing the QR codes, or you can store these QR codes digitally on cloud storage, flash drives, phones, etc. The data

on the QR codes is the exact same data on your coins and is protected by AES-256-GCM encryption. The last way you can back up your coin set is to simply use no coin set at all and export your codebook as a PDF. Exporting your codebook as a PDF eliminates all dependency on the Splitcoin app. If you have this PDF, you can decrypt your encrypted seed phrase. You will never need to use the Splitcoin app again. This is an important option to have because it is your codebook that protects your seed phrase. Having this option aligns with the mission of self-custody. When exporting the PDF from the Splitcoin app, it can be encrypted with a password or remain unencrypted. The user is made aware of the security issues surrounding PDF 2.0 and how the encryption used affects the security of the codebook. PDF 2.0 uses AES-256-CBC encryption, which is an unauthenticated form of encryption. Though CBC mode will still be sufficient, it has known vulnerabilities. Therefore, we recommend not storing this file online if you choose to export your PDF, whether it is encrypted or not.

2.2 Distribution

The second pillar of the Self-Custody Trilemma is distribution. A Splitcoin contains six data records: a coin set ID prepended with a coin number, an algorithm version number, a salt shard, a nonce, a tag, and a ciphertext. These data records work with the Create Vault and Open Vault algorithms to protect the vault key used to generate a unique codebook. Secret splitting is used to make sure that n encrypted 640-bit keys shards and n 128-bit salt shards are distributed across a set of n coins. The keys shards must be decrypted, and then the process will XOR them together to get the vault key (also called the master key) [26]. To decrypt the key shards, the key encryption keys must be derived using Scrypt [22] with the user's password and the master salt. To get this master salt, the process will XOR the salt shards together. The salt shards are not encrypted because it is not a requirement for salts to remain secret. However, it adds a layer of security that still should be noted. The strong secret splitting scheme lies in the AES-256-GCM encrypted master key, as these master key shards are encrypted before being transmitted over NFC or QR codes.

If a thief were to gain access to one coin or three of four coins, then zero percent of the secret key would be able to be revealed because n coins are required to get the master salt needed to derive the key encryption keys used to decrypt the n key shards that XOR together to assemble the master key. Therefore, the master key would be impossible to reconstruct. In the Winklevoss strategy described, access to three of four pieces of paper would give the thief seventy-five percent of the key (assuming even distribution). The Splitcoin ideology is that a simple secret sharing scheme can be a preferred method over other solutions like Shamir's Secret Sharing, which could create trust issues if not executed properly.

> *"It is used to secure a secret in a distributed way, most often to secure other encryption keys. The secret is split into multiple parts, called shares. These shares are used to reconstruct the original secret. To unlock the secret via Shamir's secret sharing, a minimum number of shares are needed. This is called the threshold and is used to denote the minimum number of shares needed to unlock the secret. An adversary who discovers any number of shares less than the threshold will not have any additional information about the secured secret."*
>
> *-Explanation of Shamir's Secret Sharing [27]*

Shamir's Secret Sharing is a common algorithm for secret sharing. However, having a threshold or weighted shares overcomplicates and creates unnecessary risk for a simple task. Deciding who is assigned these shares, what the weight of each will be, and having to manage this share threshold can be too difficult. If your trusted shareholders are compromised when using SSS, then that threshold can work against you. For this reason, we recommend making multiple sets of Splitcoins and storing them with different entities to have minimal risk of loss. These entities are commonly referred to as "guardians" when referring to social recovery techniques. It is also wise to make sure that the guardians either do not know the other guardians or at least do not know that the other guardians are also guardians. At least one guardian should be locational (such as a safe deposit box at a bank or hidden inside a picture frame in your house).

There would certainly be an advantage of using Shamir's Secret Sharing, should a single Splitcoin be lost. However, the benefits of all or nothing far outweigh the risks of t-out-of-n schemes when data redundancy (multiple sets) could solve the problem of loss or theft. The backup options previously covered can be run in the Splitcoin Tools section of the app, and it allows for these sets to be interchangeable. For example, if you lose $coin_2$ in set_1, you can just use $coin_2$ in set_2 as a replacement when recovering from set_1. Requiring a complete set of coins to scan is simply the safest way to ensure an attacker or rogue guardian(s) do not gain access to your secret.

2.3 Connectivity

The third pillar of the Self-Custody Trilemma is connectivity. For cold storage to be truly cold, the data in transit and at rest must take place on devices disconnected from the internet. It does not matter how strong the encryption used is if you enter your seed phrase and somehow your plaintext is visible to a digital intruder. A threat could be something as simple as accidentally screen mirroring to a television while entering your seed phrase! Another possible threat could be leaving your keys on cloud storage, which could create an attack vector reliant on a centralized server or weak account credentials. Not only does the Splitcoin app never see the plaintext or ciphertext, but you can keep everything cold if preferred. You can run the app on an air gapped phone, which interacts with your air gapped NFC tags or QR codes. Staying offline vastly reduces the attack surface for a bad actor to exploit.

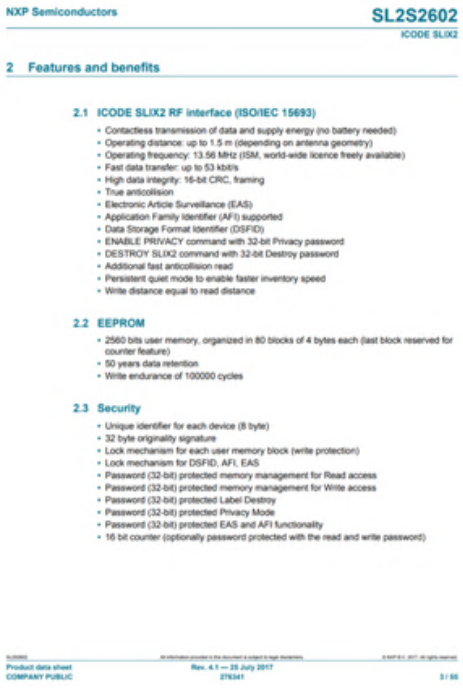
There is a level of trust that must be required when performing encryption using any piece of modern software. At some point, the plaintext must be typed in, processed by an encryption algorithm, and then the ciphertext is outputted. The same goes for decryption, where the ciphertext must be processed to output the plaintext. So how can a user be sure that the application is not malicious or that the developers of the software are not stealing their plaintext? Aside from trusting third party code reviews, how can the layman or everyday user get proof of this? Does this trust need to be reestablished with every software update? This highlights an important aspect of the Splitcoin manual encryption system. It utilizes modern day cryptography with military grade encryption and a strong key derivation function, combined with classical cryptography in the form of codebook encryption, to create the user experience of a pen and paper cipher. This eliminates all

risk of the plaintext (the seed phrase) being leaked or stolen by malware, developers, or hackers, because the mobile application never sees the seed phrase. Also, the Splitcoin application also does not store any of your data on the smartphone itself or in a remote database.

We refer to this entire system design as SPLIT39. We identify SPLIT39 as a cold storage solution because the plaintext (the seed phrase), and the ciphertext (the encrypted seed phrase) can remain permanently offline. The seed phrase (as plaintext and ciphertext) is never exposed to an electronic device at any point during the SPLIT39 protocol. It is possible for the coin set to be kept online if desired because the key shards stored on each coin are protected by AES-256-GCM encryption. SPLIT39 was designed for the coins to be stored offline or online. **The critical requirement is that the user's password must remain secret - especially if the coins are stored online**! The coins could be stored on any data storage medium. These could be stored on online databases or smart contracts to give our users more options.

3. Physical Coins

3.1 ICODE SLIX2


NXP Semiconductors

SL2S2602

ICODE SLIX2

2 Features and benefits

2.1 ICODE SLIX2 RF interface (ISO/IEC 15693)

- Contactless transmission of data and supply energy (no battery needed)
- Operating distance: up to 1.5 m (depending on antenna geometry)
- Operating frequency: 13.56 MHz (ISM, world-wide licence freely available)
- Fast data transfer: up to 53 kbit/s
- High data integrity: 16-bit CRC, framing
- True anticollision
- Electronic Article Surveillance (EAS)
- Application Family Identifier (AFI) supported
- Data Storage Format Identifier (DSFID)
- ENABLE PRIVACY command with 32-bit Privacy password
- DESTROY SLIX2 command with 32-bit Destroy password
- Additional fast anticollision read
- Persistent quiet mode to enable faster inventory speed
- Write distance equal to read distance

2.2 EEPROM

- 2560 bits user memory, organized in 80 blocks of 4 bytes each (last block reserved for counter feature)
- 50 years data retention
- Write endurance of 100000 cycles

2.3 Security

- Unique identifier for each device (8 byte)
- 32 byte originality signature
- Lock mechanism for each user memory block (write protection)
- Lock mechanism for DSFID, AFI, EAS
- Password (32-bit) protected memory management for Read access
- Password (32-bit) protected memory management for Write access
- Password (32-bit) protected Label Destroy
- Password (32-bit) protected Privacy Mode
- Password (32-bit) protected EAS and AFI functionality
- 16 bit counter (optionally password protected with the read and write password)

Product data sheet
COMPANY PUBLIC

Rev. 4.1 — 25 July 2017
278341

3 / 55

Each Splitcoin is made of a durable acrylic material and is 40mm in diameter. Our secondary product, Splitcoin Tags, are stickers that are 25mm in diameter. Inside each Splitcoin is an NFC tag, which is the ICODE SLIX2, one of the most advanced chips on the market for mobile compatibility with iOS and Android.

The ICODE SLIX2 has a 50-year data retention period, which is important to make sure that the data is retained for a reasonable lifetime [17]. It should be able to reliably store data for many years or even decades if it is used within its specified operating conditions and is not subjected to extreme environmental stress. However, as with any electronic device, there is always a risk of data loss or corruption due to factors such as physical damage or manufacturing defects. Therefore, it is important to have backup copies of coins in multiple locations. By using the NFC tags with a long data retention period and a sizeable amount of storage, Splitcoins can store data securely and efficiently.

We run an NFC clear function on each coin in our factory upon receiving them from our manufacturer. The only data that resides on the coin at the time of delivery is the unique ID of the tag, which is programmed at the factory level from NXP. This unique ID is authenticated in the app to make sure that only Splitcoin NFC tags can run the vault opening process in the mobile application. This eliminates the use of third-party tags for seed phrase recovery.

3.2 Coin Sets

Each pack of Splitcoins is sold with eight coins. When creating a set of Splitcoins in the mobile application, you can use two to eight Splitcoins. With eight coins, you could also easily make multiple sets. You could make four sets of two, two sets of three and one set of two, one set of five and one set of three, etc. Splitcoins and Splitcoin Tags can even be used in the same set. The mobile application makes it easy to make copies of those sets that can work interchangeably with one another.

4. Cryptography

4.1 AES-256-GCM

For securing the secret key used to generate the unique codebook, Splitcoin uses AES-256-GCM, an authenticated encryption mode that protects message integrity and confidentiality, to encrypt the key shard on every coin. It provides security against different attacks such as chosen-ciphertext attacks. Standard modes like CBC and CTR might be insecure if the ciphertext is intercepted by an attacker who is trying such attacks [9]. GCM is widely adopted for its performance, as throughput rates for state-of-the-art, high-speed communication channels can be achieved with inexpensive hardware resources [8]. It also uses less memory by not having extra padding on the ciphertext, which is advantageous when using NFC tags that are limited in memory such as the ICODE SLIX2. This makes it the perfect solution for our iOS/Android based application. AES-256-GCM is semantically secure, military-grade encryption that is virtually impenetrable by brute force. It is the strongest encryption standard in the world. It is used by governments, financial institutions, and the armed forces [29].

AES-256 is even believed to be quantum-resistant, unlike asymmetric encryption algorithms such as RSA [19]. The most powerful quantum search algorithm, Grover's Algorithm, could reduce the security of a brute force attack to its square root. This would still give AES-256 post-quantum security of 2^{128} (128-bit security). It is estimated that it would take 2.29×10^{32} years to crack one key in a post-quantum world with the right quantum computer [28]. This is more than one billion times larger than the age of the universe. AES-256 is the gold standard of symmetric key encryption. However, any AES-256 based scheme could fail due to poor implementation or poor key management.

Splitcoin uses a technically sound implementation of AES-256-GCM to encrypt the 640-bit key shard on each coin. Each IV is only ever used once (making it secure against chosen plaintext attacks), 96 bits in length, and is generated by a new Cryptographic Random Number Generator (CSPRNG). Each key encryption key, kek_i, is only ever used once, 256 bits in length, and is derived using Scrypt with the following parameters: N = 1048576, r = 8, p = 1, password = minimum 20 characters with complexity requirements, salt = 128 uniformly random bits generated by a new

Cryptographic Random Number Generator (CSPRNG). One nonce (IV) and one tag are stored on each coin with one encrypted key shard. Secrecy for the IVs and tags are not required. The ciphertext for each key shard can be public knowledge, as secrecy for the ciphertext is not required. If a single $coin_i$ were to be missing in a set, there would be a missing $salt\ shard_i$ (needed to reconstruct the master salt used in the KDF), a missing $nonce_i$, a missing tag_i, and a missing $ciphertext_i$ (the encrypted vault key shard) for $coin_i$.

As a .NET application, Splitcoin utilizes Bouncy Castle [2], which provides us with C# libraries used in cryptography with proven reliability since 2004. It has a strong emphasis on standards, compliance, and adaptability. Public support facilities include an issue tracker, developer mailing list, and a Wiki, which are available on the website. The Splitcoin app uses release 1.9.0 of the C# libraries released on October 17, 2021 [4]. All source code, examples, tests, and documentation for these libraries are made available on the Bouncy Castle website, and they will be available on our Github as well. Our usage of these libraries for AES-256-GCM [3] will also be available on our Github and can be found in this paper.

Below is the source code for the AES-256-GCM encryption method used in the Splitcoin application:

```
33          public List<byte[]> Encrypt(byte[] plaintext, byte[] key)
34          {
35              const int nonceLength = 12; // in bytes
36              const int tagLength = 16; // in bytes
37
38              //Generate nonce.
39              RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
40              byte[] nonce = new byte[nonceLength];
41              provider.GetBytes(nonce);
42
43              //Create destination array.
44              byte[] bouncyCiphertext = new byte[plaintext.Length + tagLength];
45
46              //Perform encryption.
47              GcmBlockCipher cipher = new GcmBlockCipher(new AesEngine());
48              cipher.Init(true, new AeadParameters(new KeyParameter(key), tagLength * 8, nonce));
49              int offset = cipher.ProcessBytes(plaintext, 0, plaintext.Length, bouncyCiphertext, 0);
50              cipher.DoFinal(bouncyCiphertext, offset);
51
52              //Split ciphertext and authentication tag.
53              byte[] ciphertext = new byte[plaintext.Length];
54              byte[] tag = new byte[tagLength];
55              Buffer.BlockCopy(bouncyCiphertext, 0, ciphertext, 0, plaintext.Length);
56              Buffer.BlockCopy(bouncyCiphertext, plaintext.Length, tag, 0, tagLength);
57
58              //Return nonce, tag, and ciphertext.
59              List<byte[]> toReturn = new List<byte[]>();
60              toReturn.Add(nonce);
61              toReturn.Add(tag);
62              toReturn.Add(ciphertext);
63
64              //Zeroize sensitive data.
65              Array.Clear(key, 0, key.Length);
66              cipher.Reset();
67              Array.Clear(bouncyCiphertext, 0, bouncyCiphertext.Length);
68
69              return toReturn;
70          }
```

To decrypt, we also use Bouncy Castle and do everything very similarly. Below is the source code for the AES-256-GCM decryption method used in the Splitcoin application:

```
72          public byte[] Decrypt(byte[] ciphertext, List<byte[]> data)
73          {
74              //Data: data[0] = nonce(length 12), data[1] = tag(length 16), data[2] = key(length 32)
75
76              //Create destination array.
77              byte[] plaintext = new byte[ciphertext.Length];
78
79              //Perform decryption.
80              GcmBlockCipher cipher = new GcmBlockCipher(new AesEngine());
81              cipher.Init(false, new AeadParameters(new KeyParameter(data[2]), data[1].Length * 8, data[0]));
82              byte[] bouncyCiphertext = ciphertext.Concat(data[1]).ToArray();
83              int offset = cipher.ProcessBytes(bouncyCiphertext, 0, bouncyCiphertext.Length, plaintext, 0);
84              cipher.DoFinal(plaintext, offset);
85
86              //Zeroize sensitive data.
87              Array.Clear(data[0], 0, data[0].Length);
88              Array.Clear(data[1], 0, data[1].Length);
89              Array.Clear(data[2], 0, data[2].Length);
90              cipher.Reset();
91              Array.Clear(bouncyCiphertext, 0, bouncyCiphertext.Length);
92              Array.Clear(ciphertext, 0, ciphertext.Length);
93
94              //Return plaintext as byte array.
95              return plaintext;
96          }
```

4.2 Scrypt

Scrypt is a key derivation function (KDF) that was first introduced in 2009 by Colin Percival, a computer security specialist. It was designed to be resistant to hardware acceleration, which means that it is difficult for attackers to use specialized hardware to crack Scrypt keys. This makes it particularly useful for password hashing and other applications where security is of the utmost importance. Scrypt is used in a variety of different contexts, including password hashing, file encryption, and the generation of random numbers [22].

It is often used as a replacement for the older and less secure key derivation functions, such as PBKDF2 (Password-Based Key Derivation Function 2) and Bcrypt. One of the key features of Scrypt is its ability to consume large amounts of memory, which makes it much more difficult to attack than other key derivation functions. For example, PBKDF2 has a major weakness: it is not GPU-resistant and not ASIC-resistant, because it uses relatively small amount of RAM and can be efficiently implemented on GPU (graphics cards) or ASIC (specialized hardware). Although Argon2id is widely considered a better KDF [5], we chose to use Scrypt to limit our dependencies. The only software package that we rely on for SPLIT39 to remain deterministic is Bouncy Castle 1.9.0 [4], which does not have an Argon2 implementation. Scrypt lets us use a 1GB output length, has been around for a longer time, and the Bouncy Castle implementation does not have the memory leaking that we experienced with some Argon2id .NET libraries.

To crack a Scrypt derived key, an attacker would need to use a significant amount of memory, which is both expensive and time-consuming. This makes it much more secure than other key derivation functions that do not require the use of large amounts of memory. In addition to its resistance to hardware acceleration, Scrypt is also designed to be resistant to parallelization. This means that it is difficult for attackers to use multiple computers or processors to crack Scrypt keys at the same time. This further increases the security of Scrypt and makes it an attractive choice for password hashing and other security-critical applications. One of the main reasons that Scrypt is considered one of the best key derivation functions available is its ability to strike a balance between security and performance. It is secure enough to protect against most attacks, but it is also fast enough to be used in a wide range of applications. The goal of the modern KDF functions such as Scrypt is to make it practically infeasible to perform a brute-force attack to reverse a password from

its hash. By following the guide listed in Practical Cryptography for Developers [16], we configure our Scrypt parameters to be extremely memory intensive. The parameters are as follows: N=1048576 (RAM = 1 GB), r=8, p=1, salt = 128 uniformly random bits. This takes approximately 45-60 seconds to run in our mobile application during the Create Vault and Open Vault processes.

We use Scrypt in two different ways throughout the Splitcoin application. The first use is usage as a key derivation function, which is to derive keys for n coins from a single password. The output length is set to n*32 bytes. We use the previously stated parameters with a password that is considered strong using the Pesto Password Strength Estimator. I developed this open-source project for anyone who needs a strong password strength estimator that only uses char arrays and can zeroize memory easily in .NET applications. It blends different features found in Azure Active Directory Password Protection and the popular zxcvbn, a "crappy password estimator" [25]. It works on a points-based system on the same scale as zxcvbn, uses the same banned word lists as zxcvbn, and includes additional lists such as popular cryptocurrency terms and 100,000 common passwords. It first normalizes an inputted password that may use leetspeak. Then it performs fuzzy matching using an edit distance of one comparison. For each bad password found, one point is assigned. For each character not found in a bad password, one point is assigned. The number of points required are configurable, but Splitcoin uses 5 points which is very complex. Additional parameters that are configurable include minimum password length, requiring an uppercase letter, requiring a lowercase letter, requiring a symbol, and requiring a number. The configuration we have chosen is: 5 points, 20 characters, uses at least one uppercase character, uses at least one lowercase character, uses at least one symbol, and uses at least one number. After running tests to directly compare the strength of zxcvbn to Pesto using these parameters, we have found that strength estimation with these parameters is even stricter than zxcvbn. This benchmark testing and more about the Pesto project is available on Github [24]. A strong password is critical to use when deriving keys from the Scrypt KDF. Even if we used strong Scrypt parameters, the strength of the keys is only as strong as the password being used. Below is the source code for Scrypt being used in the Splitcoin application to perform key derivation:

```
17          public async Task<byte[]> DeriveKeys(byte[] password, byte[] salt, int numKeys)
18          {
19              await Task.Delay(10);
20
21              //Set the number of bytes for the output length.
22              int numBytes = 32 * numKeys;
23
24              //Run Scrypt.
25              byte[] arr = Org.BouncyCastle.Crypto.Generators.SCrypt.Generate(password, salt, 1048576, 8, 1, numBytes);
26
27              //Zeroize sensitive data.
28              Array.Clear(password, 0, password.Length);
29              Array.Clear(salt, 0, salt.Length);
30
31              return arr;
32          }
```

The second way we use Scrypt is for key stretching. The secret key shards stored on each coin XOR together to create a 640-bit secret key. The first 512 bits are used as the password, and the last 128 bits are used as the salt in a Scrypt call. The use of Scrypt when we already have a uniformly random key is probably unnecessary in terms of "extra security" since we are already using a 640-bit secret key. Use of an HKDF would probably suffice. 128-bit security is the minimum-bit security to be considered acceptable until 2030 by the National Institute of Standards and

Technology (NIST) [1]. In 2031 and beyond, 128-bit security and above will be considered acceptable. Regardless, the practicality of brute forcing a 640-bit key is infeasible due to the size of its keyspace, which is 2^{640}. We chose this key size because a 640-bit key is the largest key that we can fit on the ICODE SLIX2. Any key size of 128 bits or greater would suffice. We have chosen to use Scrypt to limit the usage of third-party code while still improving the security, even if its added security is unnecessary. The most important property of our code outside of security is that it remains deterministic. More usage of third-party code only increases risk against this. The fact that it takes 45-60 seconds provides the user experience with a loading time titled "Unlocking Codebook". Using the same parameters: N=1048576 (RAM = 1 GB), r=8, p=1, and an output length of 1,048,576 bytes, we now have a byte array that we can use to create a unique codebook from when running the SPLIT39 protocol. We could reduce the value of N to speed up the function, but we do not see the time as a hinderance. Below is the source code for Scrypt being used in the Splitcoin application to perform key stretching:

```
35          //Derive 1MB key to fill the dictionary.
36          public async Task<byte[]> StretchKey(byte[] key, byte[] salt)
37          {
38              await Task.Delay(10);
39
40              //Set the number of bytes for the output length.
41              int numBytes = 1048576; //1GB
42
43              //Run Scrypt.
44              byte[] arr = Org.BouncyCastle.Crypto.Generators.SCrypt.Generate(key, salt, 1048576, 8, 1, numBytes);
45
46              //Zeroize sensitive data.
47              Array.Clear(key, 0, key.Length);
48              Array.Clear(salt, 0, salt.Length);
49
50              return arr;
51          }
```

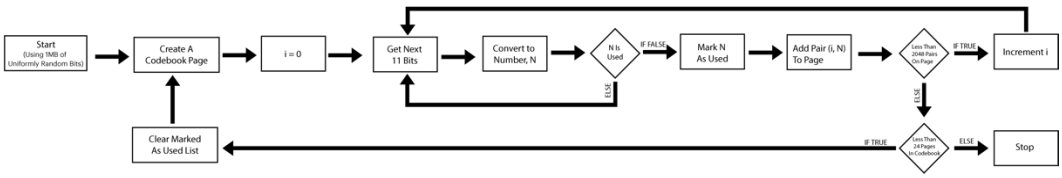
4.3 SPLIT39

SPLIT39 refers to the complete seed phrase manual encryption protocol used by the Splitcoin application, which includes codebook encryption [6] for seed phrases generated using the BIP39 standard and the use of a distributed secret key used to generate the codebook. The novelty of SPLIT39 is the manual seed phrase encryption system using an auto scrolling codebook generated from a secret key, where other elements of the system can be customized. SPLIT39 can use any secure encryption algorithm to protect the key (such as AES-256-GCM or ChaCha20-Poly1305), any secret splitting scheme (such as XOR, SSS, or none), any KDF to derive and stretch keys (such as Argon2 or Scrypt), any secure way of generating the codebook using random numbers (such as a secure random number generator or a stretched key), any storage mediums for the coins (such as NFC, QR, or cloud databases), and any word list (such as BIP39). Most of this paper covers the Splitcoin customization of SPLIT39, which is SPLIT39's namesake. BIP39 (Bitcoin Improvement Proposal 39) is a standard for representing a randomly generated mnemonic phrase, or seed, as a series of easy-to-remember words. The BIP39 standard defines a specific list of 2048 words that can be used in a seed phrase that can be 12, 15, 18, 21, or 24 words, and that each word is chosen from the list of 2048 words. Each word can be stored using the actual word or the index of the word in the BIP39 word list. This means that this index will be in the range of 1 to 2048.

The secret key shards on each coin XOR together to form the secret key. This secret key is stretched into a 1,048,576-byte array (1 MB) using Scrypt. This byte array is used to generate the

codebook. The codebook has 24 pages, and each page is essentially its own codebook. This can be compared to a polyalphabetic substitution cipher using a different alphabet for each letter in a message. In the same way, the codebook uses a different page for each seed word. Each page of the codebook will have a list of 2048 word/code pairs. By using different word code pairs, it eliminates the greatest weakness of codebook encryption (such as ECB mode), which is the patterns in the ciphertext. Because ECB encrypts identical plaintext blocks into identical ciphertext blocks, it does not hide data patterns well. The encryption of the image below illustrates this weakness:

The most important part of generating the codebook is assigning the codes to each word/code pair. Since there are 2048 word/code pairs and 24 pages, there are a total of 49,152 codes that need to be generated. The words in the pairs on each page are in alphabetical order. So, the word in Word/Code #1 will always be the same as Word #1 in the BIP39 word list, which is "abandon". The word in Word/Code #2048 will always be the same as Word #2048 in the BIP39 word list, which is "zoo". Each code will be a random value between 1 and 2048. The codes are assigned to each word/code pair by using the 1,048,576-byte array. This byte array is converted to an array of bits which will contain 8,388,608 uniformly random bits. To assign a random code to each pair, it traverses the bit array starting at index 0. Next, we get the next 11 bits and convert it to a number. This number will have a value between 0 and 2047. To get a number between 1 and 2048, simply add 1 to the result. Starting at the first word/code pair on the first page, this number becomes the code for the first the word/code pair. This process continues to traverse the array to get the next 11 bits to get a number and assigns it as the code to the next pair, repeating for each word/code pair on the page. However, if a code has already been used on the page, then the 11 bits get thrown out and it keeps getting the next 11 bits until an unused code is found. After Word/Code #2048 is assigned a code, the list of used codes is cleared. This process repeats for each page until all 24 pages have codes assigned to each of their 2048 word/code pairs. Below is a flowchart showing the algorithm used to create the codebook:



The result will be a unique codebook with 24 pages, where each page essentially acts as its own codebook used to independently encrypt each seed word. Each page of the codebook has 2048! combinations for word/code pairs. In theory, there are a total of $2048!^{24}$ possible codebooks that

could be generated from any secret key, assuming keyspace of the key provided as the Scrypt input is unlimited. However, this will be limited by the keyspace of the secret key Splitcoin uses in our application, which is 2^{640}. Is this key large enough? Something to compare this against would be a Bitcoin private key, which has a keyspace of 2^{256}. Underneath the fancy CSPRNG being used to generate the key, a simplification of what is happening is this: "pick a number between 1 and 2^{256}" or "flip a coin 256 times" [14]. We can see that in practice there is no concern with a collision of keys for a securely generated 256-bit key, so a collision of 640-bit keys is not a concern at all. Because of this, every codebook generated in the Splitcoin application will be unique.

So how do we know that 8,388,608 bits are enough to produce this codebook? During development, we ran codebook generation tests to track how many numbers (11 bits) needed to be generated to fill these codebooks. After running thousands of simulations, our results showed that it took approximately 400,000-500,000 numbers between 0 and 2047 to fill the codebook with word/code pairs. Therefore, (500,000*11)/8 = 687,500 bytes would be needed. For good measure we round it up to 1MB and set the Scrypt output length to 1,048,576 bytes. This ensures that we will have enough uniformly random bits to create the codebook from the original secret key. We could just use the secret key as a seed for a secure random number generator, but we prefer to use Scrypt to stretch the key for two reasons. The first reason is that a secure random number generator is more third-party code that the Splitcoin app would have to rely on (even if there is one in Bouncy Castle 1.9.0). The second reason is that a slow hashing function like Scrypt further defends against brute force attacks on the codebook.

After the codebook is unlocked during the vault creation and opening processes, it displays the codebook in the app. For each page in the codebook, a popup window appears and shows six different lists on the screen. Because this is a two-way codebook, Create Vault will display the codebook as word/code pairs (ordered alphabetically by the word in the pair) and Open Vault will display the codebook as code/word pairs (ordered numerically by the code in the pair, which is a number between 1 and 2048).

For creating the vault, word/code pairs appear in six different lists based on the letter that word starts with. The user locates the list they should be searching according to the first letter of their word, and then they click 'Start'. The lists will start auto scrolling. When they see their word/code pair, they write the code down. The list will keep repeating in case they miss their code. When they are ready, they can click 'Next' to go to the next word.

For opening the vault, code/word pairs appear in six different lists based on the range of numbers the code is in. The user locates the list they should be searching according to the numerical value of their code, and then they click 'Start'. The lists will start auto scrolling. When they see their code/word pair, they write the word down. The list will keep repeating in case they miss their word. When they are ready, they can click 'Next' to go to the next code.

The lists are auto scrolling so that there is no suspicion of the app tracking patterns in the vertical scroll movement inside the scroll view (scroll view y-axis) to determine the user's seed word. The Splitcoin app requires no typing, manual scrolling, or any other action that could potentially reveal information about a user's seed phrase. The scrolling speed has five different speed settings,

and it can be tailored to the user's preference. On average it takes one minute to auto scroll through an entire page.

While moving through each page in the codebook, the user can encrypt and decrypt their seed phrase by the process previously described. This is codebook encryption using a two-way codebook. Codebook encryption is a type of symmetric-key encryption, which means that the same key is used to encrypt and decrypt the message [6]. The key is the codebook, which contains the pre-agreed upon codes that correspond to each word in the seed phrase. The seed phrase is 12, 15, 18, 21, or 24 words. These could also be viewed as 12, 15, 18, 21, or 24 numbers from 1 to 2048.

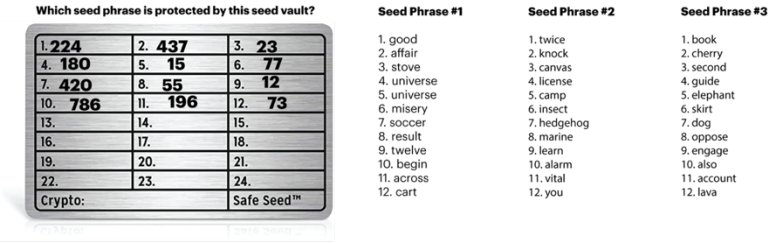
There are several weaknesses to codebook encryption. However, due to the unique design of SPLIT39, it is not susceptible to them. The first weakness is codebook compromise. If the codebook falls into the wrong hands, the confidentiality of the messages can be compromised. This is not a concern because SPLIT39 secures the codebook using AES-256-GCM encryption and by splitting the unique key used to generate the unique codebook across a coin set (distributed cold storage). The second weakness is key management. Codebooks can be large and cumbersome to manage and distribute securely. The Splitcoin codebook is digital and secured by military grade encryption. It is easy to store, transport, and back up using NFC tags, QR codes, and PDFs. This weakness is concerning for physical codebooks. The third weakness is limited keyspace. The number of possible codes in a codebook are typically limited, which means that the keyspace (the total number of possible keys) is also limited. This is simply not the case for the SPLIT39 codebook, which we have already shown has a keyspace of 2^{640} and uses a uniformly random key. This makes it computationally infeasible to brute force the key. Note that to brute force the codebook, you would also have to stretch the key with Scrypt because the actual key to the codebook encryption is the codebook.

Codebook encryption shares a prominent weakness with substitution ciphers: frequency analysis. Frequency analysis consists of counting the occurrence of each letter in a text. In a plaintext, certain combinations of letters occur with varying frequencies. For instance, given a section of English language, letters E, T, A and O are the most common, while letters Z, Q and X are not as frequently used [21]. Frequency analysis is not possible with SPLIT39 because it is designed for only encrypting seed phrases. Seed phrases are already uniformly random keys and the resulting ciphertext is also a uniformly random key. For example, let us look at a 12-word seed phrase, keeping in mind that the math scales for 15-, 18-, 21-, and 24-word seed phrases.

A 12-word seed phrase is basically a list of 12 numbers between 1 and 2048. This means that there are 2048^{12} (2^{132}) possible combinations. However, this is actually 2^{128} due to some of the data in a BIP39 phrase not being random. Regardless, SPLIT39 will work for any of the 2^{132} possible combinations (even if only 2^{128} are valid seed phrases), and it will transform any seed phrase into 1 of 2^{132} possible ciphertexts. Of course, there is always the possibility of transforming a valid seed phrase into the same seed phrase, but the odds of this would be infeasibly low (1 in 2^{128}). These are the same odds as generating two identical seed phrases.

SPLIT39 is similar to a one-time pad in theory (if the key remains secret), because seeing a ciphertext doesn't give you any extra information about the plaintext. However, it cannot be considered perfect security like a one-time pad because the key is stored and used again in

decryption. The probability of determining the plaintext after seeing the ciphertext is the same as the probability of determining the plaintext without seeing the ciphertext. The ciphertext does not give any information about the plaintext because it is equally likely for all 2^{132} possible plaintexts (only 2^{128} are valid seed phrases) to be the plaintext of any given ciphertext (any of the 2^{132} possible ciphertexts). In simple terms, SPLIT39 converts any seed phrase into what looks like another seed phrase. There is only a 6.25% chance that the ciphertext produced would be another valid seed phrase because $(2^{128})/(2^{132}) = .0625$. Storage of the ciphertext is still compatible with all commercial seed phrase storage mediums because the ciphertext will always be 12-24 numbers between 1 and 2048. It can also be converted to 12 to 24 BIP39 words. The important thing to recognize is that without having access to the codebook, **all seed phrases are equally likely to be the plaintext of any ciphertext**. The following ciphertext and three possible plaintexts illustrate this:



Which seed phrase is protected by this seed vault?

1. 224	2. 437	3. 23
4. 180	5. 15	6. 77
7. 420	8. 55	9. 12
10. 786	11. 196	12. 73
13.	14.	15.
16.	17.	18.
19.	20.	21.
22.	23.	24.
Crypto:		Safe Seed™

Seed Phrase #1	Seed Phrase #2	Seed Phrase #3
1. good	1. twice	1. book
2. affair	2. knock	2. cherry
3. stove	3. canvas	3. second
4. universe	4. license	4. guide
5. universe	5. camp	5. elephant
6. misery	6. insect	6. skirt
7. soccer	7. hedgehog	7. dog
8. result	8. marine	8. oppose
9. twelve	9. learn	9. engage
10. begin	10. alarm	10. also
11. across	11. vital	11. account
12. cart	12. you	12. lava

Practically, SPLIT39 provides computational security. The only known cipher to provide perfect security is the one-time-pad because the key is never reused in whole or in part. The security is reduced to the same level of security as AES-256-GCM because this is the layer of encryption protecting the vault key at rest and in transit. A cipher can only be as secure as the key distribution, and it does not provide perfect security if the key is secured by a modern-day block cipher like AES. However, reducing the security of the cipher to that of AES-256-GCM is more than secure enough, especially with the secret splitting schemes interwoven into SPLIT39. The strength of AES is only as good as the strength of the derived keys. So, assuming the most dangerous threat model where an attacker manages to obtain the seed vault and every coin in your coin set, the password is what keeps everything secure. By using a strong password and memory intensive Scrypt parameters, we can rest assured that this password would take an infeasible amount of time to brute force.

Lastly, like the one-time-pad, SPLIT39 requires that a codebook must only be used once to protect a single seed phrase. This is made very clear in the app before running the vault creation process. Codebook encryption shares similarities with substitution ciphers. Substitution ciphers, even those that are polyalphabetic, are not considered to be CPA (Chosen Plaintext Attack) secure. A chosen plaintext attack is a type of attack in which the attacker could get encrypted plaintexts and analyze the resulting ciphertexts. Substitution ciphers are not considered to be CCA (Chosen Ciphertext Attack) secure either. A chosen ciphertext attack is a type of attack in which the attacker could get specific ciphertexts decrypted and analyze the resulting plaintexts. The vulnerability is that analysis of plaintext/ciphertext pairs will start to reveal the codebook, which is basically the key. This is dangerous if a plaintext/ciphertext pair reveals a part of the codebook used to encrypt another plaintext.

In conclusion, SPLIT39 provides many benefits that work perfectly for seed phrase encryption. A digital, portable, secure codebook encryption system built into a mobile application that allows the user to easily encrypt and decrypt their seed phrase by hand is advantageous when dealing with random keys as the plaintext, which is the case with seed phrases. It eliminates the exposure of the plaintext to the software while making sure that the resulting hand-recorded ciphertext cannot be attacked. The ciphertext is not subject to the weakness of frequency analysis because the plaintext is always a random key. It is not subject to the weakness of pattern analysis because each word is encrypted independently with its own key (page in the codebook). It is impossible to determine the original seed phrase without the key if the codebook is only ever used to encrypt one seed phrase. The security of the secret key used to create the codebook is where the security challenge lies. By using a strong password and strong parameters with the Scrypt key derivation function, we can secure the secret key on a set of Splitcoins with a simple secret splitting scheme and encrypting the resulting shards with AES-256-GCM.

4.4 Threat Modeling

SPLIT39 is designed to be a secure encryption system that abides by Kerckhoffs's principle. The principle states that a cryptographic system should be secure even if everything about the system, except the key, is public knowledge [12]. For threat modeling purposes, we must note that if the password falls into the wrong hands, all your coins and encrypted seed phrase would still be needed for an attacker to gain access to your seed phrase. If the attacker was missing even one coin, the seed phrase is secure. Additionally, keeping your encrypted seed phrase offline is recommended because it makes a remote digital attack impossible to succeed if the plaintext or ciphertext are never digitally exposed. However, this is not a requirement of the system design.

Specifically looking at the threat of the Splitcoin app, we already established that it never sees the encrypted seed phrase or the original seed phrase. Therefore, it is impossible for the app to ever gain access to your seed phrase. The greatest threat is an attacker who can physically obtain your encrypted seed phrase and your coins. If an attacker were to find all your coins and your encrypted seed phrase, your seed phrase would remain secure unless they could obtain the key encryption keys used in your coin set. All key encryption keys are derived from the same password using a single Scrypt function call. So, the entire security of the system is reliant on the password remaining secret.

Passwords form the cornerstone of modern digital security. The Splitcoin app requires users to use a 20-character password employing a mix of lowercase, uppercase, symbols, and numbers from an 89-character ASCII set. We use Pesto, which is an open-source, secure password strength estimator used for sensitive applications that require mutable data structures. I developed Pesto and performed benchmark testing against zxcvbn to prove its efficacy as a password strength estimator. From the results, we can see that a much more complex password is required to achieve a Pesto score of 4 than a zxcvbn score of 4 for the Pesto settings used by Splitcoin (5 match points and 20 characters with at least 1 lowercase letter, at least 1 uppercase letter, at least 1 number, and at least 1 special character). You can refer to the Pesto repository on GitHub for more information on how it works and to view benchmark testing results [24].

As previously discussed, this password is provided to a Scrypt KDF configured with parameters N=1048576, r=8, and p=1. The derived keys are 256-bit in length, and Scrypt utilizes a 128-bit randomly generated salt. Salt is a random data string that is unique per user and concatenated with the password prior to the hashing process. The presence of a unique salt drastically enhances the complexity of cracking hashes by neutralizing the utility of pre-computed tables (lookup tables, reverse lookup tables, and rainbow tables). We used GPT-4 to analyze and estimate the feasibility of a successful password cracking attempt on Splitcoin's system design using popular password cracking techniques discussed by CrackStation [11].

Lookup tables are pre-computed data structures that store hash values for potential passwords. Given a hash, an attacker can search this table for a match to recover the original password. However, the use of unique salts for each password renders lookup tables ineffective. If two users have identical passwords, the unique salt ensures they will still produce different hashes. An attacker cannot simply generate a hash for a guessed password; they must compute the hash for the guessed password and each unique salt. Considering the 128-bit salt in this scenario (i.e., 2^{128} unique possible salt values), an attacker would have to create 2^{128} lookup tables for every single password, a computational and storage feat that is beyond current and near-future technological capabilities.

A reverse lookup table, a variant of the lookup table, is indexed by hash values rather than the original password. Thus, an attacker can retrieve matches even if only a partial hash value is known. However, the application of a unique salt for each hash ensures that reverse lookup tables, like standard lookup tables, become infeasible. For every password guess, an attacker would have to compute hashes for each unique salt, leading to computational demands that are currently impossible to meet.

Rainbow tables represent a more storage-efficient version of lookup tables. They involve a trade-off between computation and storage, allowing a substantial reduction in the space needed to store hashes. However, they are also vulnerable to the presence of unique salts. A rainbow table built for a specific hash function becomes irrelevant when a unique salt is appended to every password. Given the 2^{128} possible unique salts in this scenario, an attacker would have to create 2^{128} rainbow tables for every potential password, an unachievable task with today's computational capabilities.

Despite the presence of salt, certain password-cracking techniques can still be theoretically pursued, although their practicality is a different question altogether. We will discuss two of the most common methods: brute-force and dictionary attacks.

Brute-force attacks involve systematically trying every possible combination of characters until the correct password is found. While this attack methodology is theoretically capable of cracking any password, in practice, the resources required for a brute-force attack against a complex password can be prohibitive.

Consider our password scenario: we have a 20-character password made up of a mix of lowercase letters, uppercase letters, symbols, and numbers from an 89-character ASCII set. The total number of possible combinations for this password would be 89^{20}, which equals approximately 1.95×10^{39}. To put this into perspective, let us consider an imaginary supercomputer capable of checking one quintillion (billion billion) passwords per second. Even with this absurdly fast computer, it

would take approximately 1.95x 10^{21} years to try all possible combinations according to GPT-4. That is about 14 billion times the current age of the universe (~1.38 x 10^{10} years). This illustrates the astronomical scale of the problem. However, for the sake of argument, let us say we do not have an imaginary supercomputer, but rather an Application-Specific Integrated Circuit (ASIC), a type of hardware specifically designed for hashing operations. Given the parameters specified for Scrypt, an ASIC could perform approximately 100,000 Scrypt hash operations per second. That is significantly slower than our hypothetical supercomputer, so you can already see the scale of the problem growing. Even if we deploy an array of these ASICs, the situation does not improve significantly. For example, if we had 5000 such ASICs running in parallel, we would be able to check 500 billion passwords per second. At this rate, it would still take ~1.23 x 10^{27} years to exhaust all password combinations. That is about 89 trillion times the current age of the universe.

Beyond time, there are other considerations that make a brute-force attack infeasible. The first of these is cost. Assuming that each ASIC costs $2000 and we are deploying 5000 of them, the hardware alone would cost $10 million. But the costs do not stop there. Let us assume each ASIC consumes power at about 500W per device. If the array of 5000 ASICs runs continuously, the total power consumption would be 2.5MW. In a year, that would amount to 21.9 million kWh. With an average cost of electricity in the U.S. being about $0.12 per kWh, the cost of running these ASICs for a year would be around $2.6 million. Over the time it would take to exhaust the password space (1.23 x 10^{27} years), the electricity costs would mount to ~$3.11 x 10^{21}, a number that is so astronomical that it dwarfs any kind of budget we can realistically conceive. These are just the direct costs. We also need to consider the costs of maintaining and cooling the ASIC array. ASICs produce heat, which must be dissipated to prevent damage. This requires a significant cooling infrastructure, which increases costs further. There is also the issue of replacing ASICs that fail, which is a virtual certainty over the time scales involved. Finally, there is the logistical issue of managing such a large array of devices. With 5000 ASICs, you are effectively running a small data center. This would require additional infrastructure and personnel, adding yet more costs.

In conclusion, while brute-force attacks are theoretically capable of cracking any password, the time and resources required to mount such an attack against a complex password are prohibitive. Given the size of the password space, the slow rate of hash operations, and the significant costs involved, a brute-force attack against a 20-character password hashed with Scrypt and salted is practically infeasible. It would take longer than the current age of the universe by many orders of magnitude and it would incur costs that are beyond astronomical.

The brute-force attack scenario outlined above highlights the importance of using strong, unique passwords and protecting them with robust hashing and salting techniques. Given the computational, financial, ethical, and legal hurdles involved, it is clear that brute-force attacks against strong passwords are infeasible with current and foreseeable technology. Instead of trying to crack passwords, our efforts would be better spent on educating our users about good password hygiene. I created Pesto to eliminate a user password being the weak link in a secure system.

Dictionary attacks, as compared to brute-force attacks, adopt a more 'intelligent' approach by leveraging known or likely password combinations. These attacks rely on lists of words or commonly used password structures, hence the term 'dictionary'. However, even though dictionary

attacks are more efficient than brute force attacks, their success greatly depends on the complexity and uniqueness of the user's password. Our password scenario involves a 20-character password with a high Pesto score of 4. A score of 4 indicates a very strong password that is likely to resist common dictionary attacks. The password uses a mix of lowercase letters, uppercase letters, symbols, and numbers from an 89-character ASCII set, further improving its resilience against dictionary attacks.

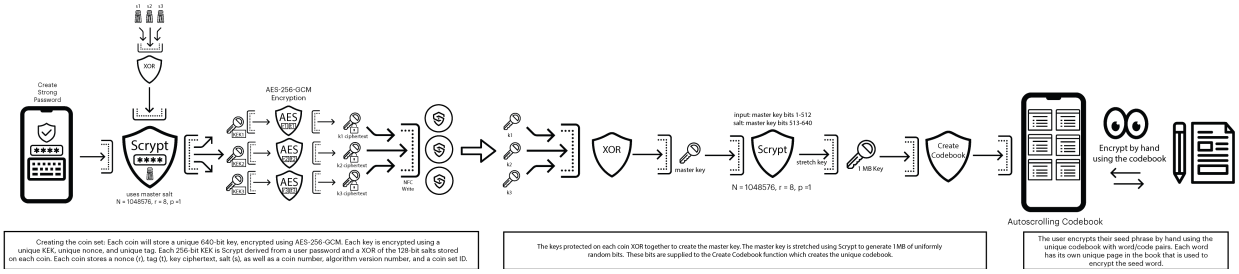
Let us hypothetically analyze a dictionary attack with a list of 1 billion passwords. For this attack, you must keep in mind the size and complexity of the password space. In our scenario, where a password is 20 characters long and composed of a possible 89 characters, the total number of possible passwords is 89^{20}. This is the total password space. Comparatively, a dictionary with 1 billion passwords covers only an infinitesimal fraction of this total password space. To put it into perspective, if you were to compare 1 billion to the total password space, it's about 5.12×10^{-31}%, an incredibly small portion. In practice, even if an attacker is using a sizeable dictionary of 1 billion passwords, the odds of our complex password being included in that dictionary are extremely low. Given the password's length, complexity, and Pesto's resistance to the use of common passwords, it is highly unlikely to be found in common dictionaries used by attackers. Therefore, even with a large dictionary, a dictionary attack would be largely ineffective against our password scenario. Additionally, as with the brute-force approach, the attacker would need to calculate the Scrypt hash for each password in the dictionary, a computationally intensive and time-consuming process, especially given Scrypt's design to resist attacks using custom hardware. So, even with a billion-password dictionary, the dictionary attack would still be impractical and unlikely to succeed. The strength of the password and the resilience of the Scrypt algorithm significantly deter such attempts, demonstrating the importance of using long, complex passwords and strong, secure password hashing algorithms. Furthermore, even if the dictionary was tailored towards more complex and longer passwords, the size of such a dictionary would be prohibitively large. For instance, a dictionary containing all combinations of 10 characters (out of our set of 89) would already contain about 3.43×10^{19} entries. Note that our password is 20 characters long, making a comprehensive dictionary vastly larger and unwieldy to use in practice. The success of a dictionary attack also heavily depends on the speed at which an attacker can hash and check each dictionary entry. Assuming the attacker uses the same ASIC setup mentioned in the previous section (capable of hashing at a rate of 100,000 Scrypt hashes per second), and assuming they deploy 5000 ASICs, they would be able to check about 500 billion passwords per second. If the attacker tries to run a dictionary attack with a dictionary of 3.43×10^{19} entries (10-character combinations), it would take them about 2.17×10^{8} years according to GPT-4. That is over fifteen times the current age of the universe. And remember, this is only for 10-character combinations. The actual password is 20 characters long, and the dictionary size for all 20-character combinations is exponentially larger.

On top of the time constraints, the cost associated with this kind of attack is also prohibitive. The initial hardware cost for 5000 ASICs, each priced at $2000, would be $10 million. The ongoing electricity cost would be about $2.6 million per year, assuming an average US electricity cost of $0.12 per kWh and a power consumption of 500W per ASIC. These costs do not even include additional expenses related to cooling, maintenance, and replacement of the ASICs.

ASICs typically have limited on-chip memory, as memory is expensive and takes up a lot of space on the chip. As a result, you will not be able to perform many parallel Scrypt operations on each ASIC, because you would run out of memory. Even if you could somehow fit 1GB of memory into each ASIC, the cost of the ASICs would be much higher, and the number of Scrypt operations you could perform per second would be much lower. On top of that, the Scrypt algorithm is not just memory-hard, it is also computationally intensive. This means that even if you had enough memory to perform many parallel Scrypt operations, each operation would still take a significant number of computational resources and time to complete. Therefore, these numbers are a best-case scenario, assuming that the only limit is how fast you can compute the Scrypt hash. This ignores the memory requirements and computational intensity of the Scrypt algorithm. In reality, the time and resources needed to perform a brute-force attack or dictionary attack on a Scrypt-hashed password would be orders of magnitude greater than these estimates. While the basic math is sound, it does not consider the full complexity of the problem when dealing with Scrypt. These additional factors make these attacks even less feasible than the calculations suggest.

In conclusion, a dictionary attack against a complex, 20-character password hashed with Scrypt and salted is, like a brute-force attack, practically infeasible. It would require an enormous amount of time and financial resources. The length and complexity of the password, combined with the hashing and salting techniques employed, make the dictionary attack as challenging as a brute-force attack. These calculations underscore the importance of strong, unique passwords, which can withstand even advanced cracking methodologies.

5. Create Vault



This process securely splits a secret key across a set of Splitcoins. This "vault key" unlocks a unique codebook used to encrypt your seed phrase by hand. The codebook will show you how to translate each of your seed words into a code, which is a random number between 1 and 2048. You can record these codes in your seed vault. A "seed vault" is anything that stores your encrypted seed phrase, such as metal, paper, or the included magnet vault.

First, the user is asked if they are creating a new coin set or loading an existing coin set. Creating a new coin set means that the user would like to generate and store a new codebook across their coin set. Then, they will be shown their codebook with word/code pairs used for encrypting their seed phrase by hand. Loading an existing coin set means that the user would like to recover their codebook from their coin set and be shown the word/code pairs used for encrypting their seed

phrase by hand. Creating a new set will always be done first because you must have a coin set to load if you would like to load an existing coin set. Loading an existing coin set will be less commonly used, but it gives the user the option to manually encrypt their seed phrase a second time or resume the process if they are interrupted during the initial vault creation process.

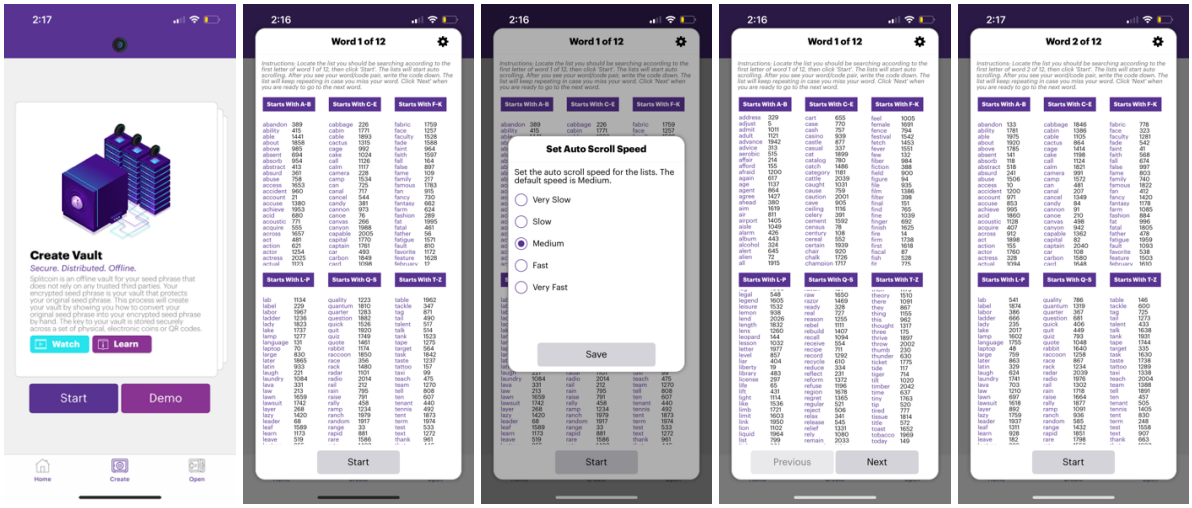
If the user decides to proceed with the vault creation process, they will enter how many coins will be used in their set (from two to eight). They will select that number when prompted. The user will then choose one of six seed phrase lengths so the app can know how many pages in the codebook to display. These options are 12 words, 15 words, 18 words, 21 words, or 24 words. One page (popup) will later be shown for each word.

If the user is creating a new coin set, choosing a password is the next step. This password will be used to derive AES-256-GCM encryption keys using Scrypt for this coin set. This means that simply scanning all the coins would not be enough to recover the secret key used to generate the codebook. This password would also be required. The user will enter this password, and then they will enter it again for confirmation. The requirement for this password is at least 20 characters in length, at least one uppercase letter, at least one lowercase letter, at least one number, and at least one symbol. It must also receive a Pesto score of 4. Pesto is scored on the same scale (0-4) as the popular zxcvbn password strength estimator. Using a password (to derive strong keys for AES-256-GCM encryption) adds a secure channel for data in transit via NFC and QR. This provides defense against eavesdropping attacks. However, due the nature of NFC requiring close-proximity communication, some of these attacks are unlikely since Splitcoin is most likely being used in a secure location like a home or office. AES-256-GCM encryption also provides encryption at rest on the NFC tags and QR codes.

If a new coin set is being created, the next step is to create the coin set and write the data to the physical Splitcoins via NFC. If a coin set is being loaded, this data will be read from the physical Splitcoins via NFC. The app can determine that a single Splitcoin is not read twice due to the unique ID of the NFC tag. If a coin is being read, it must be an authentic Splitcoin. Below is a chart of the generated data and how it is stored on each coin:

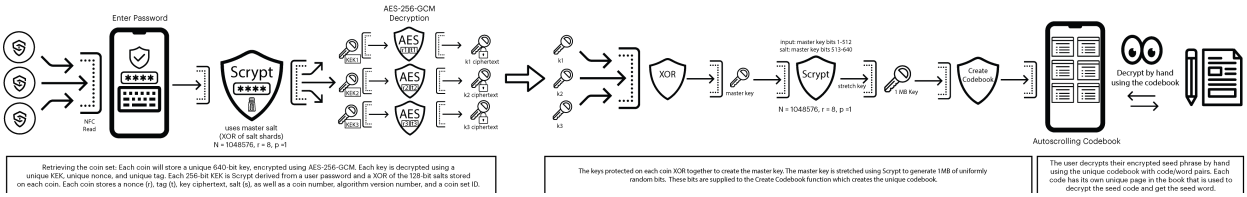
	Data Record #1	**Data Record #2**	**Data Record #3**	**Data Record #4**	**Data Record #5**	**Data Record #6**
What Data Is Stored?	Coin Index + Coin Set ID (Base64 String)	Algorithm Version Number (Base64 String)	Salt Shard (Base64 String)	Nonce (Base64 String)	Tag (Base64 String)	Ciphertext (Base64 String)
What Does It Do?	The coin index is the number of the coin in the set. This will be a value of 1-8. It is concatenated with a unique ID for the coin set, so that your coins belonging to the same set can be easily identified.	The algorithm version number is used to identify the algorithm that was used to create this coin set. This ensures that if a better algorithm is found and the app is updated, this coin set will always work with this algorithm.	This salt shard is 128 bits. The master salt is a XOR of every salt shard in your coin set. The master salt is supplied to the Scrypt KDF as the salt to derive your keys used to encrypt the key shard on each coin.	The nonce is 96 bits and used in AES-256-GCM. It needs to be stored to decrypt the ciphertext.	The tag is 128 bits and used in AES-256-GCM. It needs to be stored to decrypt the ciphertext.	The ciphertext is the encrypted 640-bit key shard. The 640-bit key shards XOR together to get the master key. This master key is used to generate the unique codebook used to encrypt/decrypt the seed phrase by hand.

The secret key that has now been generated (if creating) or recovered (if loading) will be provided to the codebook creation function as previously defined. The user will then step through the codebook as needed to encrypt their seed phrase by hand. For each word, the user will be shown a page of their codebook that shows all their possible seed words and their corresponding numeric codes. This will display six lists in a popup inside the app. First, the user locates the list they should be searching according to the first letter of word, and clicks 'Start'. The lists will start auto scrolling. After the user sees their word/code pair, they write the code down. The list will keep repeating in case the user misses their word. The user clicks 'Next' when they are ready to go to the next word. After this has been completed for every word, the user has an encrypted seed phrase as described by the SPLIT39 protocol. Below are screenshots of this process that can be seen in the mobile application:



At this point the vault creation process is complete. As the user, you should hide your coins in different hiding places or with trusted individuals. How they are stored is completely up to you! Remember, you will need your full set to recover your codebook that decrypts your seed phrase. Most importantly, do not lose your password! Splitcoin Inc. cannot help you recover your coins, password, or seed vault if anything is lost. We do not store or have access to any user data. With the understanding that Splitcoin is a self-custody tool only, we cannot stress the importance of making backups enough. Splitcoin Tools on the Home tab give you alternative ways to back up access to your codebook, such as making copies of your coins, QR backups, and PDF printouts. If you have backups, then losing a coin will never be problematic.

6. Open Vault



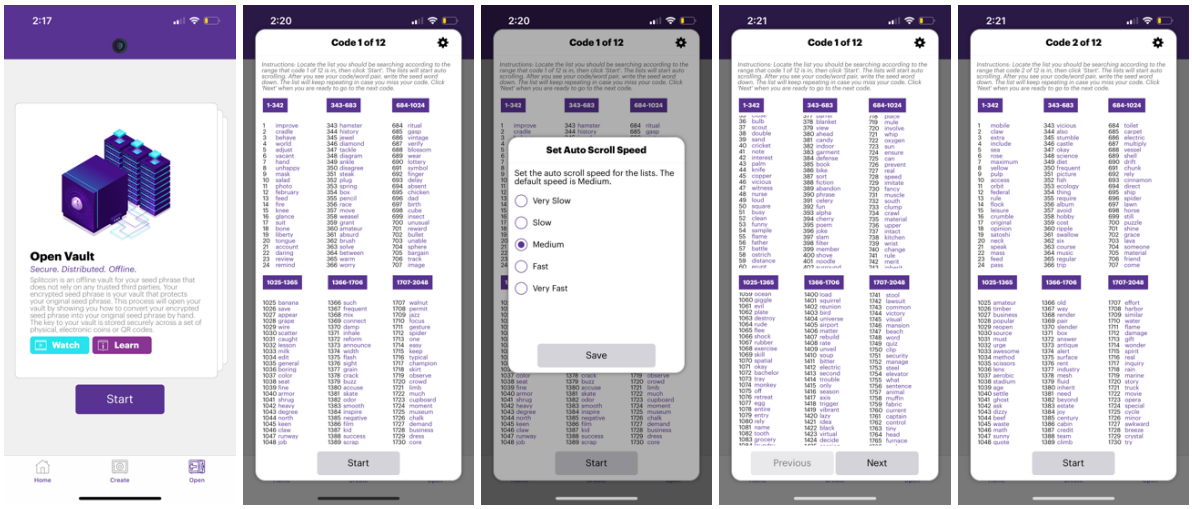
This process securely recovers a secret key from a set of Splitcoins. This "vault key" unlocks a unique codebook used to decrypt your seed phrase by hand. The codebook will show you how to translate your numeric codes into your original seed words. You can record this on a piece of paper that should be disposed of when you are finished with your seed phrase.

If the user decides to proceed with the vault opening process, they will enter how many coins will be used in their set (from two to eight) and then select that number when prompted. The user will then choose one of six seed phrase lengths so the app can know how many pages in the codebook to display. These options are 12 words, 15 words, 18 words, 21 words, or 24 words. One page (popup) will later be shown for each word.

The next step is to scan and read the data from each of the physical Splitcoins via NFC. The app can determine that a single Splitcoin is not read twice due to the unique ID of the NFC tag. If a coin is being read, it must be an authentic Splitcoin.

Entering the password is the next step. This password will be used to derive AES-256-GCM encryption keys using Scrypt for this coin set. This means that simply scanning all the coins would not be enough to recover the secret key used to generate the codebook, but this password would also be required. The salt shards on each coin XOR together and are used with password to derive the encryption keys using Scrypt. Due to the authentication of AES-256-GCM, we will be able to immediately tell if the password is valid, although we will first have to wait for the Scrypt KDF to complete. This adds an extra 45-60 seconds to the process each time a different password is attempted. If a password is the incorrect password for this coin set, it will allow the user to try again. If the correct password is entered, the secret key shards are decrypted and we XOR them together to recover the secret key.

The assembled secret key is provided to the codebook creation function as previously defined. The user will then step through the codebook as needed to decrypt their encrypted seed phrase by hand. For each numeric code, the user will be shown a page of their codebook that shows all their possible numeric codes and their corresponding seed words. This will display six lists in a popup inside the app. First, the user locates the list they should be searching according to the range that the code is in, and clicks 'Start'. The lists will start auto scrolling. After they see their code/word pair, the user writes the seed word down. The list will keep repeating in case they miss their code. The user clicks 'Next' when they are ready to go to the next code. After this has been completed for every code, the user has their original seed phrase as described by the SPLIT39 protocol. Below are screenshots of this process that can be seen in the mobile application:



At this point the vault opening process is complete. As the user, you should hide your coins in different hiding places or with trusted individuals. How they are stored is completely up to you! Remember, you will need your full set to recover your codebook that decrypts your encrypted seed phrase. Most importantly, do not lose your password! Splitcoin Inc. cannot help you recover your coins, password, or seed vault if anything is lost. We do not store or have access to any user data. Understanding that Splitcoin is a self-custody tool only, we cannot stress the importance of making backups enough. Splitcoin Tools on the Home tab give you alternative ways to back up access to your codebook, such as making copies of your coins, QR backups, and PDF printouts. If you have backups, then losing a coin will never be problematic.

7. Tools

7.1 Authenticate Coins

The authenticate tool is custom to the Splitcoin app. When this tool is executed, it will run an NFC read. Every Splitcoin comes preprogrammed from the semiconductor manufacturer (NXP) with an ICODE SLIX2 unique ID. To run the vault opening process on your device, your coins need to be authenticated. An internet connection is required to authenticate your coins. If you have an internet connection while running Open Vault, your coins will authenticate automatically. If you want to use the vault opening process with an offline smartphone, then you should run this tool while you have an internet connection available.

7.2 Read Coin

The read tool is a basic NFC function. When this tool is executed, it will run an NFC read. The user scans a Splitcoin with their device, and it will read the data records (also referred to as data

slots) on the NFC tag. By reading this data, you can see some useful data about each Splitcoin. The coin number shows the number of the coin in the set.

- *Coin Number*: This shows the number of the coin in the coin set. It will be a value from 1 to 8.
- *Coin Set ID*: This will show the unique ID of your coin set. This is helpful if coins from different coin sets get mixed up and you need the ability to sort them or determine which coin set a Splitcoin belongs too.
- *Data Hash*: This shows a SHA-256 hash of the ciphertext stored on this coin. This is helpful to show that the data on your coin differs from another coin in the same set.

7.3 Reset Coin

The reset tool is a basic NFC function. When this tool is executed, it will run an NFC clear. The user will scan a Splitcoin with their device, and it will clear every data record on the NFC tag. Once a Splitcoin is reset, the cleared data cannot be restored. This tool is only made available for the user in case they want to run the reset tool on a Splitcoin that they may have used for something else. It most likely will never be used. It can never destroy a coin or ruin a coin set because every NFC write performed by the Create Vault process will be locked. This means that the coins will be made to be read-only, and a reset (NFC Clear function) would fail if attempted.

7.4 Copy Coin

The copy tool is used to copy one NFC tag to another. When this tool is executed, it will first prompt the user to read one NFC tag. Next, it will prompt the user to scan a second NFC tag. It will then write the data read from each data record on the first tag to each data record on the second tag.

7.5 Import From QR Codes

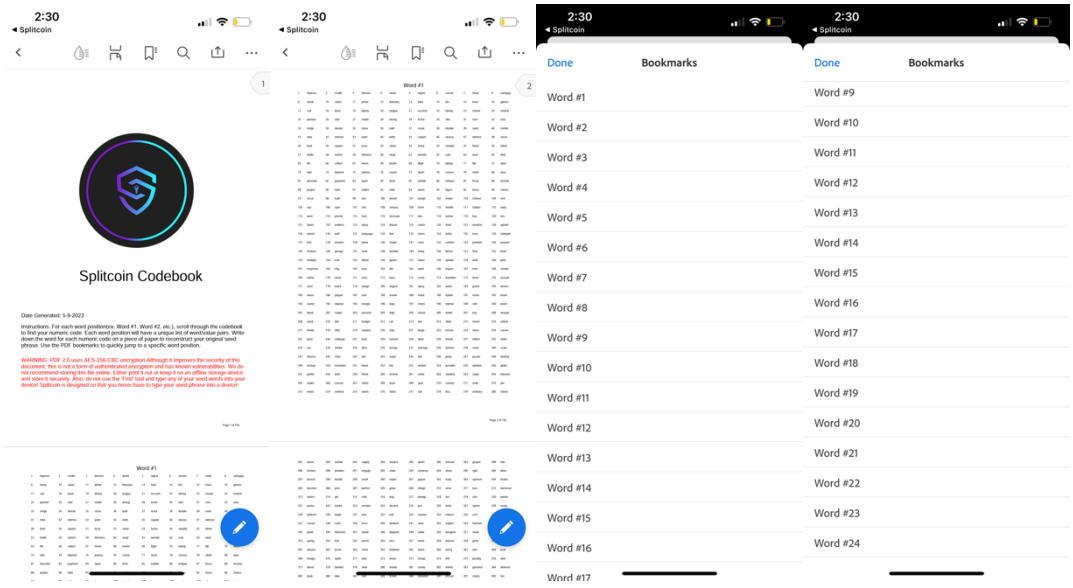
The import from QR codes tool is used to import your coin set from Splitcoin QR codes. Each QR is a coin in your coin set. Storing your coins in the form of QR coins gives you an alternative way to back up your coins: either by printing them out or storing them digitally.

7.6 Export To QR Codes

The export to QR codes tool is used to export your coin set to Splitcoin QR codes. These QR codes will be generated onto a PDF file which opens in the mobile application. Each QR is a coin in your coin set. Storing your coins in the form of QR codes gives you an alternative way to back up your coins: either by printing them out or storing them digitally.

7.7 Export To Codebook

This tool will export your coin set to a Splitcoin Codebook stored as a PDF using Syncfusion UI controls. Exporting your codebook as a PDF eliminates all dependency on the Splitcoin app. If you have this PDF, you can decrypt your encrypted seed phrase. You will never need to use the Splitcoin app again! We want to give you this option because it is **your codebook** that protects **your seed phrase**. Having this option aligns with the mission of self-custody. You can even encrypt this PDF from the Splitcoin application. However, you must be aware of the security issues surrounding PDF encryption and how that affects the security of your codebook. Encrypted PDF files have known vulnerabilities to skilled attackers! PDF 2.0 does not use authenticated encryption like the rest of the Splitcoin app. Therefore, we do not recommend storing this file online. Either print it out or keep it digitally on an offline storage device like a flash drive. Below are screenshots of a codebook exported to a PDF by the mobile application:



7.8 Convert To Words

Your encrypted seed phrase is a list of numbers. The value of each of number is between 1 and 2048. You can store your encrypted seed phrase as a list of numbers (numeric codes), or you can use this tool to convert each number into a seed word. This will make your encrypted seed phrase look like a seed phrase! This is helpful if you want to store your encrypted seed phrase using storage mediums such as Billfodl that require words to store a seed phrase. Alternatively, you can store your encrypted seed phrase using storage mediums such as the Splitcoin Magnet Vault, Hodlr One, or Cryptotag, which all use numbers to store a seed phrase.

7.9 Convert To Numbers

If you converted your encrypted seed phrase into a list of words, then you will need to use this tool to convert it back to a list of numbers before you can run the vault opening process and decrypt it.

8. Mobile App Security

8.1 Air Gapping and Authentication

The smartphone running the Splitcoin app never stores any encrypted data or encryption keys. All data generated from the vault creation process is only stored on your Splitcoins that are air gapped, meaning that they do not connect to the internet. Splitcoin gives you the ability to make the entire experience air gapped by using a secondary phone permanently in airplane mode. This makes the entire Splitcoin experience a malware-resistant process since no payload could ever reach the internet. Essentially, the phone placed permanently in airplane mode acts like a 100% offline, air gapped device, providing a high level of security against malware.

The optional, fully air gapped process works like this. First, the smartphone must be online to install the app from the App Store for iOS or the Google Play Store for Android. The user then starts up the app and goes through the launch process: reading all the explainers, watching videos, and exploring the mobile application. Before the phone can be turned offline permanently, the vault opening process requires that the coins must be authenticated. The user will run the Authenticate Coins tool to do this. This authentication process works by reading the unique ID of every coin and making sure that it is an authentic Splitcoin. Each coin has a unique ID on its NFC tag, which is permanently burned to the tag by NXP, the NFC tag manufacturer.

At this time, the smartphone is still connected to the internet and no secret keys have been generated by a CSPRNG yet. Authentication of your coins will be required whether you choose the fully air gapped experience or not. If your smartphone is online during the Open Vault process, your coins will be automatically authenticated during the process. The Authenticate Tool will scan your empty coins and call our API to authenticate them. Our SAS API is used to generate shared access signatures that grant read-only access to our Azure Cosmos DB, which is a database of all Splitcoin unique IDs. The Azure Cosmos DB is used to verify the unique ID of a Splitcoin as an authentic Splitcoin. The API call returns a short expiration, read-only SAS to the mobile application to allow it to run queries using a table service client from Azure SDKs. There is no sensitive data stored in the database, which is why these read-only calls do not require authentication. These unique IDs are never associated with any customer data. The unique IDs of the authenticated Splitcoins (non-sensitive data) are then saved to secure storage on the smartphone. It is important to make sure that all your Splitcoins have been authenticated if you choose the fully air gapped experience.

The smartphone can now be placed into airplane mode and never has to come back online. All your Splitcoins that have been authenticated can be used with the phone permanently in airplane mode. For users worried about malware, you can now run the vault creation and opening processes with your authenticated Splitcoins knowing that it is impossible for any payload to ever touch the internet!

8.2 App Hardening

The need for robust self-custody tools in the cryptocurrency space has been emphasized by recent events. The collapse of reputable exchanges and other centralized institutions has made the age-old adage "not your keys, not your coins" more crucial than ever. As users of cryptocurrencies take charge of their own keys and transition from exchanges to self-custody solutions such as hardware wallets, there is still a major challenge in securely storing the seed phrase offline. As Changpeng Zhao, CEO of Binance says, “Most people are not able to back up their security keys; they will lose the device. They will not have the proper encryption for their backup; they will write it on a piece of paper, someone else will see it, and they will steal those funds [13].”

Splitcoin was designed to be a solution to this seed phrase encryption challenge, and it still complies with the most extreme seed phrase storage best practices: Never share your seed phrase with anyone, never store your seed phrase online, and never enter your seed phrase into a device that is not a hardware wallet. The key shards generated in the mobile application are encrypted using AES-256-GCM with the Scrypt derived key encryption keys from your password before any NFC transmission or QR export takes place. During development of the mobile application, we used the MASVS [18] for various security design principles and app hardening techniques [10].

Zeroization of sensitive data is a critical element of mobile app security. All keys are created in memory using the RNGCryptoServiceProvider class [7] and stored in byte arrays in memory. After they are used and no longer needed, these byte arrays are immediately zeroized using the Array.Clear() method. This is true for anything that could be considered sensitive such secret key shards, secret keys, key encryption keys, passwords, and codebooks. Other things that we still zero out are ciphertexts and salts, even though these are not considered sensitive. An important thing to highlight is that the app never has information about your seed phrase or the encrypted seed phrase. The Splitcoin app will never retain access to your codebook, keys, or passwords after the processes complete.

Disposing of resources is another way we protect sensitive information generated in the Splitcoin app. We utilize “using” statements for all our uses of the RNGCryptoServiceProvider class [7] and our custom AES256GCM and Scrypt classes (both use Bouncy Castle), which both implement IDisposable. The IDisposable interface simply facilitates this release of unmanaged resources. When an object of a class implementing the IDisposable interface is used with a "using" statement, the instance is automatically disposed once it is no longer needed to prevent sensitive information generated by it from being leaked or accessed by unauthorized users. Whenever an instance of these classes is needed, a new instance of that class is created and only used once. The RNGCryptoServiceProvider class in the .NET framework is used to generate random numbers for

cryptographic purposes [7]. We use this class to generate our IVs, keys, and salts. Using statements and disposing of instances can help to prevent key leaks and ensure the secure management of cryptographic keys. When keys are created, they are typically stored in memory, which means that they could potentially be accessed by other applications or users with access to the system. If the code block within the "using" statement is exited, even if an exception occurs, the Dispose() method of the object is automatically called.

To protect the codebook, there are notable design features to explain. The first feature is securely getting user input for the password that the key encryption keys are derived from. To do this, we designed our own keyboard popup from scratch using secure stack layouts and char arrays that can be zeroized. We did not use a standard solution that takes input from an iOS or Android keyboard because these store the user input as a String and are susceptible to key loggers. Sensitive data is never stored as a String in the Splitcoin application because a String is an immutable data structure that cannot be zeroized in memory.

Another important feature is the use of Pesto, which was previously discussed. I designed an estimator that matches the performance of zxcvbn [25] but does not require the password being estimated to be stored as a String. It is being processed as a char array, and it is zeroized in memory following each estimation.

The app is carefully developed to make sure the Label class limits the exposure of the codebook and the user password on the mobile device screen and in memory. To display data on the screen we use the Label class, which is included in the Xamarin.Forms framework. The challenge here once again lies in using the String data structure. The sensitive data that needs to be shown on the screen are the password (when the user clicks the eyeball button to check what they have typed) and the display of the codebook itself. To solve this, we sequentially create labels in memory at the launch of the application for every individual character that exists in the keyboard, labels for every word in the codebook, and labels for every numerical code. These labels (and their text stored as Strings in memory) are all created at once in alphabetical/numerical order. For the temporary password display, the Labels of the corresponding characters in the char array are added as children to a secure stack layout inside of a horizontal scroll view, and then the children (a mutable collection) of the secure stack layout are cleared when the temporary display times out after 2000 milliseconds. While the temporary display is not showing, the children in the secure stack layout inside the horizontal scroll view are bullet points (ex: ••••••), equal to length of the char array storing the currently typed in password. A similar concept is implemented for the pages of the codebook. Each page displays six lists, which are secure stack layouts inside vertical scroll views. For the display of the codebook, int arrays are the only data structure passed into the popup views. The int arrays store the index of the BIP39 word labels and numeric code labels that need to be added to the six lists in the popup view. These int arrays are zeroized after they are used. The children (a mutable collection) in the secure stack layouts are cleared as well so a page of the codebook is not sitting in memory after it has been viewed.

Screen overlay attacks are important to protect against on Android. We use custom renderers for the StackLayout class and ContentPage class to create SecureStackLayout and SecureContentPage (secure views for our virtual keyboard and codebook displaying popups). By

setting ImportantForAccessibility.NoHideDescendants on a view indicates that the view's content should always be visible to accessibility services, even if the view is currently hidden or obscured by other views. By default, views are considered "descendants" of their parent views, which means that they may be hidden from accessibility services when the parent view is hidden or obscured. By setting ImportantForAccessibility.NoHideDescendants, we can ensure that the content of the view is always accessible to accessibility services, regardless of its visibility. The way we do this is by using the setFilterTouchesWhenObscured(boolean) function. When this function is called with a value of true, the system will discard touch events that are received by an app when it is not visible on the screen, such as when another app is overlaying it. This can prevent malicious apps from intercepting touch events and stealing sensitive data. Together, these settings can help protect sensitive information or content within a view by preventing it from being captured by accessibility services or other apps when the app is not in the foreground or when the window is partially or completely obscured [30].

The Window.SetFlags (WindowManagerFlags.Secure, WindowManagerFlags.Secure) method call on Android sets the SECURE flag on the window, which prevents screenshots that can compromise the security of the app. Setting the SECURE flag on the window prevents the window from being captured or recorded by other apps, which can help to prevent screen scraping attacks and other types of attacks that rely on capturing the contents of the screen. This is particularly important for the custom virtual keyboard in the app to take user input for the user password and for the popups displaying the codebook on the screen. On iOS we use NSNotificationCenter.DefaultCenter.AddObserver(UIScreen.CapturedDidChangeNotification, Callback). This adds an observer to the default NSNotificationCenter object to listen for changes to the captured state of the screen. By adding an observer for the CapturedDidChangeNotification notification, we are notified when the screen capture status changes in the app. This helps us protect sensitive information by preventing screen recording of the password being typed in. It also prevents screen recording of the auto scrolling codebook when it is displayed.

On iOS, we also use a full screen dark blur with a popup of the Splitcoin logo when overriding the OnResignActivation(UIApplication application) function in the AppDelegate class. This allows us to hide these sensitive screens when the app moves to the background, and it makes sure that no sensitive data is visible in the app switcher. On Android, we already use Window.SetFlags (WindowManagerFlags.Secure, WindowManagerFlags.Secure) to prevent screenshots, which keeps a secure view in the app switcher by showing a blank white screen [15].

When communication is secured, the risk of man-in-the-middle attacks is neutralized. Man-in-the-middle attacks occur when a malicious actor intercepts communication between two devices, allowing them to eavesdrop on or manipulate the conversation. By implementing AES-256-GCM encryption on the key shards before any data transmission over NFC or QR, we can ensure secure channels and defense against these attacks. This means that even if some malware were logging your NFC data or your Splitcoins fall into the wrong hands, your codebook cannot be recovered without your password. The Splitcoin app does not use any hardcoded keys or app keys. The only encryption keys used in the app are derived from your strong user password using Scrypt with strong parameters.

The static data across your coin set is another feature to highlight. The Splitcoin app makes sure that the data stored on the NFC tags are permanently locked to the tags. This is to ensure that the NFC tags are never overwritten by accident or by anyone with malicious intent. Making the Splitcoins read-only after they are written to for the first time eliminates this risk completely. QR codes also provide static data in the sense that they cannot be overwritten. They can also be digitally backed up.

As a self-custody tool first and foremost, we feel a duty to empower our users. Splitcoin gives our users multiple options for storing their codebook and encrypted seed phrase so that they can choose a strategy most suitable for them. For example, your encrypted seed phrase can be written on paper, written on a magnet, stored on flash drive, engraved in metal, etc. Consumer behavior does not change when it comes to writing down a seed phrase, but the encrypted seed phrase (which looks like a seed phrase) keeps your funds safe from prying eyes. The encrypted seed phrase can be stored as a list of numbers between 1 and 2048 or as a list of BIP39 words. It is compatible with all seed phrase storage solution such as Billfodl, Cryptotag, and Hodlr One. Some recommended strategies can be found on the Splitcoin blog.

Another way we empower our users is giving them the option for their seed phrase to be recovered without any dependency on Splitcoin Inc. or the Splitcoin app. The options for storing your codebook are storing it across a set of acrylic Splitcoins, Splitcoin Tags, QR codes, or on a PDF. After storing your codebook across a set of coins, you can use Splitcoin Tools to export your coin set to a set of QR code backups and a PDF backup. We plan on releasing another mobile application later this year called Splitcoin Lite. It is a lite version of the Splitcoin app that lets you run the vault opening process only using QR codes. It will be completely open source and available on the Google Play Store for Android only. Until then, users can still export their codebook as a PDF using Splitcoin Tools in the Splitcoin mobile application. The only thing you need to decrypt your seed phrase is the digital or printed PDF, your PDF password (if you used one), and your encrypted seed phrase. There is no reliance on the app or physical Splitcoins whatsoever. Splitcoin Inc. is on a mission to deliver the most reliable self-custody tools. Empowering our users means to give them options to cut out dependencies on third parties if they so choose.

8.3 Audits and Bounties

Claims are nothing without proof. Splitcoin Inc. will prove the safety of our mobile application using professional security audits for our software. Our code will be audited and peer reviewed in the future to ensure that that the application works as advertised. To further provide evidence of fund security, www.splitcoin.com will have a bounty page showing that our solution cannot be hacked. For example, one bounty may show the data on three out of four coins in a set. Bounty hunters can try to hack the funds inside the wallet protected by the coin set. We will have multiple bounties to prove that funds are always safe if the password or at least one Splitcoin in a set is inaccessible.

References

[1] Barker, Elaine. National Institute of Standards and Technology, 2020, *Recommendation for Key Management: Part 1 – General*, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf. Accessed 31 May 2023.

[2] "Bouncy Castle (Cryptography)." *Wikipedia*, 24 Jan. 2021, en.wikipedia.org/wiki/Bouncy_Castle_(cryptography).

[3] Brady, Scott. "Authenticated Encryption in .NET with AES-GCM." *Scott Brady*, 20 May 2021, www.scottbrady91.com/c-sharp/aes-gcm-dotnet.

[4] "C# Cryptography APIs." *The Legion of the Bouncy Castle* , www.bouncycastle.org/csharp/. Accessed 31 May 2023.

[5] CheatSheets Series Team. "Password Storage Cheat Sheet." *Password Storage - OWASP Cheat Sheet Series*, 2021, cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

[6] "Codebook." *Wikipedia*, 3 Aug. 2022, en.wikipedia.org/wiki/Codebook.

[7] Crozier, Jonathan. "How to Generate a Cryptographically Secure Random String in .NET with C#." *Jonathan Crozier*, 19 May 2022, jonathancrozier.com/blog/how-to-generate-a-cryptographically-secure-random-string-in-dot-net-with-c-sharp.

[8] "Galois/Counter Mode." *Wikipedia*, 17 Oct. 2021, en.wikipedia.org/wiki/Galois/Counter_Mode.

[9] Green, Matthew. "How to Choose an Authenticated Encryption Mode." *A Few Thoughts on Cryptographic Engineering*, 29 Aug. 2016, blog.cryptographyengineering.com/2012/05/19/how-to-choose-authenticated-encryption/.

[10] Haupert, Vincent, et al. *Honey, I Shrunk Your App Security: The State of Android App Hardening*, 2018, faui1-files.cs.fau.de/filepool/projects/nomorp/nomorp-paper-dimva2018.pdf.

[11] Hornby, Taylor. "Salted Password Hashing - Doing It Right." *CrackStation*, 28 Sept. 2021, crackstation.net/hashing-security.htm.

[12] "Kerckhoffs's Principle." *Wikipedia*, 15 May 2022, en.wikipedia.org/wiki/Kerckhoffs%27s_principle.

[13] Key, Alys. "'99% of People' Will Lose Crypto Storing in Self-Custody: Binance CEO Changpeng Zhao." *Yahoo! Money*, 14 Dec. 2022, money.yahoo.com/99-people-lose-crypto-storing-144442564.html.

[14] "Keys and Addresses." *Saylor.Org Academy*, learn.saylor.org/mod/book/view.php?id=36350&chapterid=18931. Accessed 31 May 2023.

[15] Lucio, Vicente Gerardo Guzmán. "Protecting Sensitive Data in the Background in Xamarin.Forms." *Syncfusion*, 4 Nov. 2020, www.syncfusion.com/blogs/post/protecting-sensitive-data-in-the-background-in-xamarin-forms.aspx.

[16] Nakov, Svetlin. "Scrypt." *Practical Cryptography for Developers*, 2022, cryptobook.nakov.com/mac-and-key-derivation/scrypt.

[17] NXP B.V. "SL2S2602 ICODE SLIX2 Rev. 4.2 — 1 December 2021 Product Data Sheet." *NXP*, 1 Dec. 2021, www.nxp.com/docs/en/data-sheet/SL2S2602.pdf.

[18] OWASP Foundation. "Owasp MASVS." *OWASP Mobile Application Security*, 2023, mas.owasp.org/MASVS/.

[19] O'Shea, Dan. "AES-256 Joins the Quantum Resistance." *Fierce Electronics*, 29 Apr. 2022, www.fierceelectronics.com/electronics/aes-256-joins-quantum-resistance.

[20] Palatinus, Marek, et al. "Mnemonic Code for Generating Deterministic Keys." *GitHub*, 26 July 2022, github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

[21] Pan, Yining. "The Scope of Application of Letter Frequency Analysis in Substitution Cipher." *Journal of Physics: Conference Series*, 2022, iopscience.iop.org/article/10.1088/1742-6596/2386/1/012015/pdf.

[22] Percival, Colin. *Stronger Key Derivation via Sequential Memory-Hard Functions*, www.tarsnap.com/scrypt/scrypt.pdf. Accessed 31 May 2023.

[23] Popper, Nathaniel. "How the Winklevoss Twins Found Vindication in a Bitcoin Fortune." *The New York Times*, 19 Dec. 2017, www.nytimes.com/2017/12/19/technology/bitcoin-winklevoss-twins.html?_r=0.

[24] Ranieri, Freddie. "Pesto 1.0.0." *GitHub*, 24 May 2023, github.com/aardev14/Pesto.

[25] Richards, Tony. "Zxcvbn C#/.NET." *GitHub*, 12 Feb. 2021, github.com/trichards57/zxcvbn-cs.

[26] Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C* . 20th Anniversary ed., Wiley, 2015.

[27] "Shamir's Secret Sharing." *Wikipedia*, 28 Nov. 2021, en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing.

[28] Tobias, Eric. "128 or 256 Bit Encryption: Which Should I Use?" *Ubiq*, 15 Feb. 2021, www.ubiqsecurity.com/128bit-or-256bit-encryption-which-to-use/.

[29] "What Is Military-Grade Encryption?" *PassCamp*, 7 Apr. 2022, www.passcamp.com/blog/what-is-military-grade-encryption/.

[30] Éliás, Tibor. "Mobile Overlay Attacks on Android." *IKARUS Security Software*, 12 Nov. 2020, www.ikarussecurity.com/en/mobile-device-management-en/mobile-overlay-attacks-on-android/#4.