# Airbitz Edge Security White Paper

The Airbitz mobile digital wallet implements a decentralized, secure, peer-to-peer synchronized, and backed up bitcoin wallet with a clean, easy to use interface.  Integrated with the wallet is a proximity enabled and searchable business directory to help consumers find merchants that accept bitcoin as payment.

The Airbitz wallet is a fully client-side wallet that creates, manages, and encrypts private keys, public keys, and transactions on the client device and uses server side assistance for backup and synchronization of encrypted data and for authenticating users. Airbitz users retain full control of their private keys at all times with neither Airbitz nor third parties having access to private keys or transaction data. Sending and detecting of received funds is done through publicly-accessible Libbitcoin or Electrum server nodes.

The Airbitz mobile wallet is implemented using a GUI written in the native language and platform of the device. iOS and Android implementations are written in Objective-C, Java, or React Native.  in Apple's NS framework with newer version written in Javascript React Native and Android devices are written in Java.

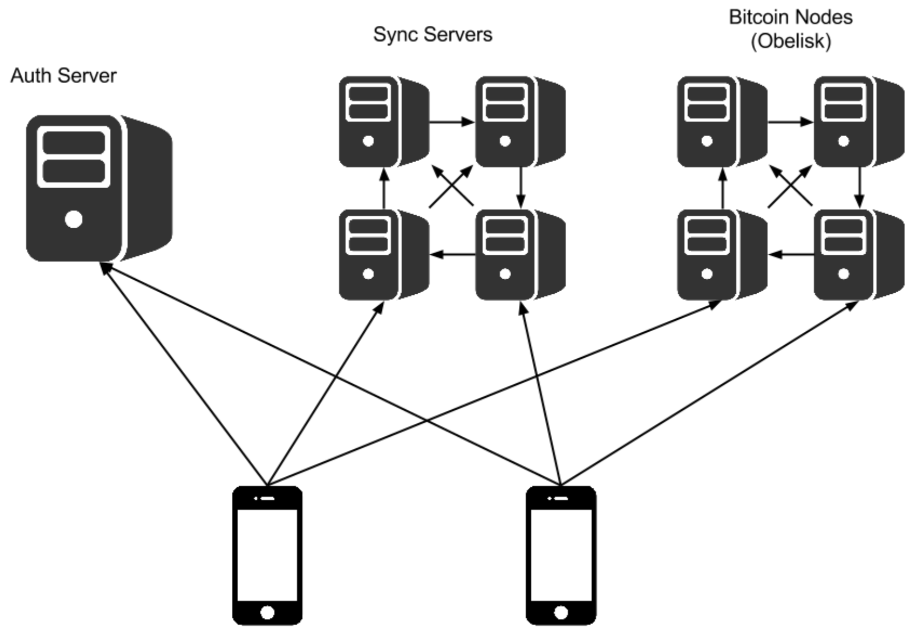
Data security and backup is implemented implemented in a cross platform C/C++ or Javascript library which is compiled into the platform and operating system of each device. This is referred to as the Airbitz Core or ABC. ABC implements all account and wallet creation, user authentication, management of transaction data, encryption, and synchronization across devices.

The ABC architecture is inherently "client-strong" meaning nearly all wallet logic exists and operates on the client device. Servers exists only to assist in authentication, for encrypted backups, and as nodes on the bitcoin network. ABC leaves control of the private keys entirely in the hands of the user or application which controls the login and password of the account. Airbitz servers have no capability of spending funds on users' behalf.

## Client-Server Architecture

ABC utilizes a "95% decentralized" client-server architecture which utilizes 3 separate types of servers.

1. The first is a conventional app server developed in Python/Django with a PostgresSQL database. Although this is auto-replicated and backed up, this is still a centralized server. It is utilized for creating new accounts and authenticating existing accounts. In ABC, it is referred to as the ‘auth server’. This server incorporates automatic dual server replication and encrypted hourly backups to an offsite server. Unavailability of the auth-server still allows Airbitz wallets to function for send, receive, and synchronization between devices which have accounts that were previously logged in.
2. ABC utilizes a semi-decentralized, peer-to-peer network of data-sync backup servers running the Git protocol and software. Referred to as ‘sync-servers’, this pool of Git servers only hosts data that is pre-encrypted by the client devices running ABC. These are true peer-to-peer servers in that there are no primary or secondary servers, however all sync-servers and the auth-server are operated by Airbitz. A client can connect to any of the sync servers to upload or download data. Any new or modified data sent to a sync-server is automatically replicated to other sync-servers, utilizing the Git protocol to resolve conflicts and provide modification history and rollback if needed. To scale this architecture, “pools” of sync-servers will be created in the future such that each pool will replicate data across the sync-servers in the pool, but not across other pools. If all sync-servers were unavailable, Airbitz wallets can still function to Send and Receive funds directly from public bitcoin nodes.
3. ABC relies on public bitcoin full nodes running Electrum/Stratrum for doing transaction detection and broadcast. These servers are publicly accessible nodes hosted by Airbitz and other community members. In this way, wallets can seamlessly send and receive funds even if all of the Airbitz server infrastructure were to be unavailable, a feature no other wallet API can provide.

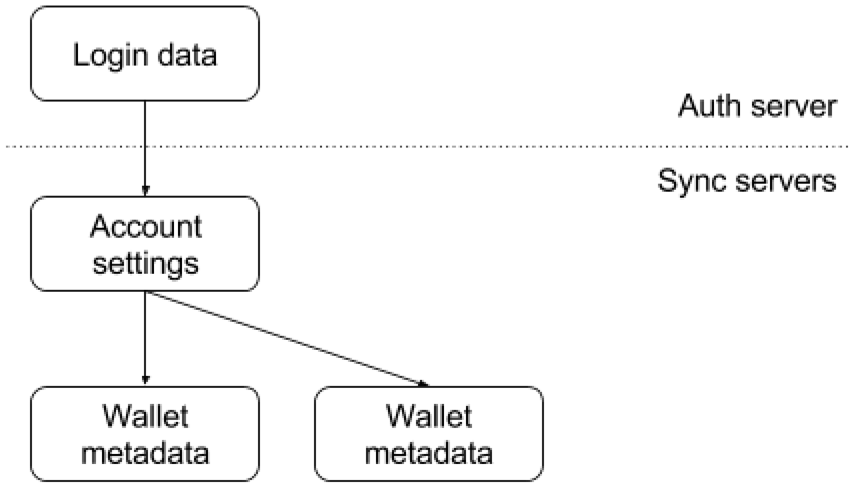


# Data Model

The Airbitz data model contains two types of data:

- Login data
- Synced data

The login data allows the user to gain access to their account data. It consists of encrypted keys to the account, as well as the hashed credentials, such as usernames and passwords, needed to access those keys. The Airbitz auth server maintains a copy of the login data and provides it to authenticated users upon request (subject to rate-limiting, 2-factor authentication, and other safeguards). Once users gain access to their login data, they can decrypt the keys to their account data:



The account and wallet data are both located on the Airbitz sync servers, which are general-purpose data stores. The account store typically holds user settings, a list of wallets, and the private keys for those wallets. The wallet store typically holds transaction and address metadata. Applications using the Airbitz SDK can write any data they like to these data stores. Accessing either type of data store requires two keys:

- syncKey - 160-bit shared secret used to authenticate with the sync server
- dataKey - 256-bit AES-CBC encryption key for the data store contents

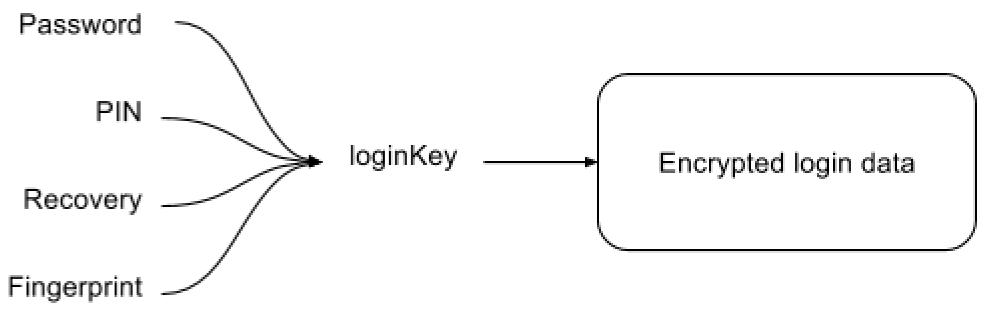
The dataKey is never revealed to the sync server, so Airbitz cannot read the contents of the synced data.

# Logging In

Since the login data controls access to the account data, “Logging in” is defined as obtaining the 256-bit loginKey that decrypts the login data. There are several ways to retrieve this key:

- Username and password
- Username and PIN
- Username and recovery answers
- Thumbprint to unlock the phone’s secure storage

By combining this loginKey with a copy of the login data, the user can gain access to their synced data (where the actual value is stored). As long as they lead to the loginKey, all methods are equivalent. This flexible architecture allows Airbitz to introduce new zero-knowledge login methods as new use cases arise.



## Password Login

To perform a password login, the client first needs a copy of the login data. If the client has a copy of the data from a previous login, it can just use that. This is known as an “offline” login, and allows users to access their accounts even if the Airbitz servers are unavailable. Otherwise, the client needs to obtain a copy of the login data from the auth server.

### Server Authentication

To perform an online password-based login, the client hashes the username and password as follows:

```
userId = scrypt(username, userIdSnrp)
passwordAuth = scrypt(username + password, userIdSnrp)
```

The parameters for these hashes, known collectively as `userIdSnrp`, are:

- `n=16384`
- `r=1`
- `p=1`
- `salt='b5865ffb9fa7b3bfe4b2384d47ce831ee22a4a9d5c34c7ef7d21467cc758f81b'`

The client sends these hashes to the auth server. If the userId matches a known account, the auth server hashes passwordAuth a second time using a variable salt. If the double-hashed passwordAuth matches the double-hashed passwordAuth stored in the database, the user has authenticated, and the server sends back the login data.

### Decryption

The login data includes the following password-related fields:

- `passwordBox`
- `passwordKeySnrp`

To log in, the client must decrypt passwordBox, which holds loginKey. The key to this box is called passwordKey, and is derived from the password and the passwordKeySnrp parameters:

```
passwordKey = scrypt(username + password, passwordKeySnrp)
loginKey = decrypt(passwordBox, passwordKey)
```

Although passwordAuth and passwordKey are both derived from the same password, the two values use completely different salts and difficulty parameters. Since the auth server never sees the plaintext username or password, it has no way to derive passwordKey and obtain loginKey. This is what makes the login “zero-knowledge”. The server may hold all the data, but cannot read any of it.

As time goes on, phones become increasingly powerful. To take advantage of this, the Airbitz core selects difficulty parameters for passwordKeySnrp based on the phone’s performance. The more powerful the phone, the stronger the password hashing, and the greater the brute-force resistance.

## PIN Login

PIN login is only available on devices that have previously logged in before. These devices save a 256-bit piece of data, called pin2Key, locally on the disk.

To perform a PIN login, the client first hashes the username and PIN using the pin2Key:

```
pin2Id = HMAC-SHA256(data=username, key=pin2Key)
pin2Auth = HMAC-SHA256(data=pin, key=pin2Key)
```

The client sends these two hashes to the auth server. Assuming these hashes match the values stored in the database, the auth server sends back the login data for that user.

The login data includes the following PIN-related fields:

- `pin2Box`
- `pin2KeyBox`

To log in, the client simply decrypts pin2Box with pin2Key, retrieving loginKey.

The pin2KeyBox is used to synchronize the pin2Key between devices. Each time a device logs in, regardless of the method, it decrypts pin2KeyBox to obtain pin2Key, and stores that key locally in plaintext.

Any time the pin2Key and pin2Box are brought together, the loginKey is readable. Thus, the security of this scheme rests on keeping pin2Key on the client and pin2Box on the server. The client does **not** store a copy of pin2Box on disk, since that defeats the purpose. This separation is necessary because 4-digit PIN's are easy to guess. By requiring the auth server on each PIN login, Airbitz can enforce rate-limiting in a way that would be impossible with purely client-side cryptography.

There is an older version of PIN login that uses weaker cryptography (scrypt instead of HMAC), but is based on the same principles.

## Recovery Login

To perform a recovery login, the user must first possess a 256-bit key called recovery2Key. The Airbitz application emails this key to the user when they first enable recovery questions, and each logged-in device also saves the key to disk in plaintext. Thus, either a copy of the recovery email or a previously logged-in device are sufficient to begin the process.

To retrieve the recovery questions, the client first hashes the username with the recovery2Key:

```
      recovery2Id = HMAC-SHA256(data=username, key=recovery2Key)
```

The client sends this hash to the auth server, and auth server replies with the encrypted recovery questions (and nothing else). To decrypt the recovery questions, the client uses recovery2Key.

Once the user answers the questions, the client hashes each of the answers using recovery2Key:

```
      recovery2Auth = [HMAC-SHA256(data=answer1, key=recovery2Key),
…]
```

It sends these answer hashes to the auth server along with the recovery2Id from earlier. Assuming everything matches, the auth server sends back the login data.

The login data includes the following recovery-related fields:

- `recovery2Box`
- `recovery2KeyBox`

To log in, the client simply decrypts recovery2Box with recovery2Key, retrieving loginKey.

The recovery2KeyBox is used to synchronize the recovery2Key between devices, in the same way as for PIN login.

The same security considerations that apply to PIN login apply to recovery login. The client does not store recovery2Box on disk, ensuring that the server is always required. This allows the server to rate-limit the login attempts, which is important if the answers are in any way guessable.

Note that recovery2Id is not the same as userId. Thus, knowing the username alone is not enough to retrieve the questions. An attacker must actually know the recovery key before they can see the questions.

## Two-Factor Security

To provide additional security against brute-force attempts, users can enable 2-factor security on their logins. When this is enabled, the auth server requires a six-digit TOTP token in addition to any other authentication. Without this token, the server will not reveal the login data.

To compute the TOTP token, both the client and server follow the algorithm specified in RFC 6238. Both the client and the server store the same 40-bit shared secret, called the “otpKey”, in plain text.

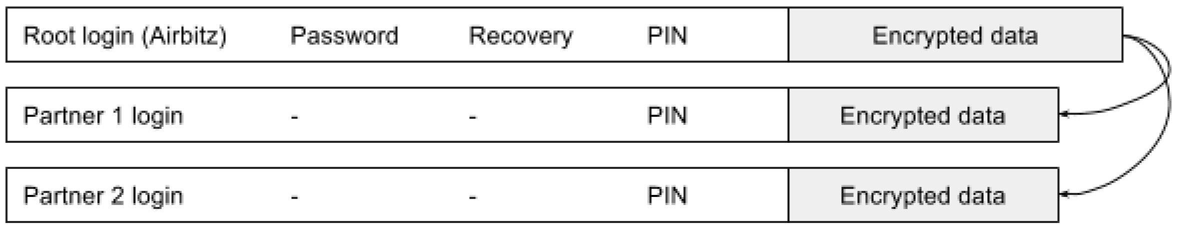
As long as the client has a copy of the otpKey, it can access the auth server without user intervention. The first login on a new device is different, however, since the otpKey would not be present. In these cases, the user must scan the otpKey from an existing device to complete the login. An attacker attempting to brute-force the login would not have access to an existing device, and would therefore be thwarted.

To prevent users from being permanently locked out of their accounts, Airbitz also includes the option to request a “2-factor reset” when the user lacks an otpKey. When this happens, the auth server begins a one-week countdown until the reset takes place.

The Airbitz application periodically checks for pending 2-factor resets in the background. If an attacker requests a reset, the legitimate user will see a notification on their device, regardless of whether or not they are currently logged into Airbitz. The legitimate user can then cancel the reset and take any other defensive measures they see necessary, such as changing passwords.

# Edge Login

Every Airbitz partner application has an “appId”. When a user logs into a partner application, the Airbitz client code creates a new bundle of login data associated with that particular appId. Each bundle has its own loginKey, which is encrypted with the master Airbitz loginKey. Thus, gaining access to the root Airbitz loginKey provides access to all application-specific loginKeys. On the other hand, gaining access to a single child loginKey does not provide access to any other child loginKey or to the root Airbitz loginKey. This minimizes the damage a single compromised application can do.



Each child login also has its own pin2Key. This makes it possible to perform PIN login into a partner application without requiring access to the root Airbitz login. Only the root login has a password or recovery answers, though, since those provide access to the entire tree in any case.

To request an edge login, the partner app places its appId into a communications “lobby” on the Airbitz auth server. The partner app then displays a barcode with the lobby’s address. Assuming the user scans the barcode and approves the request, the Airbitz app simply returns the application-specific loginKey and login data and through the lobby. At that point, the partner application is logged in.

The lobby is protected with ECDH cryptography, so nobody but the communicating parties can read the login reply, including the auth server itself.

## Lobby Cryptography

To create a lobby, the requesting application begins by generating a random secp256k1 keypair, which will be used to encrypt any lobby replies. The lobby’s id is derived from the keypair:

```
lobbyId = SHA256(SHA256(publicKey.x))
```

The lobbyId can be truncated to any length, but current implementations take the first 80 bits. This is long enough to be unique, and is strong enough to make tampering with the public key impractical. The requesting application sends the lobbyId, public key, and request contents to the auth server. It then displays the lobbyId as a barcode.

The replying application scans the barcode and fetches the lobby contents from the auth server. It immediately verifies that the public key contained in the lobby still matches the lobbyId, preventing any man-in-the-middle attacks (the QR code is the root of trust in this security model). Assuming the user approves the request, the replying application creates its own secp256k1 keypair and performs a standard ECDH multiplication to derive a shared point. It turns the shared point into an encryption key using the following KDF:

```
lobbyKey = HMAC-SHA256(0x00000001 | secretX, “dataKey”)
```

The replying application encrypts its reply using this key, and uploads it to the lobby along with the matching public key.

The requesting application then retrieves the reply, performs the matching ECDH derivation, and decrypts the contents. Since only the requesting application knows the original secret key, only the requesting application can decrypt the reply.