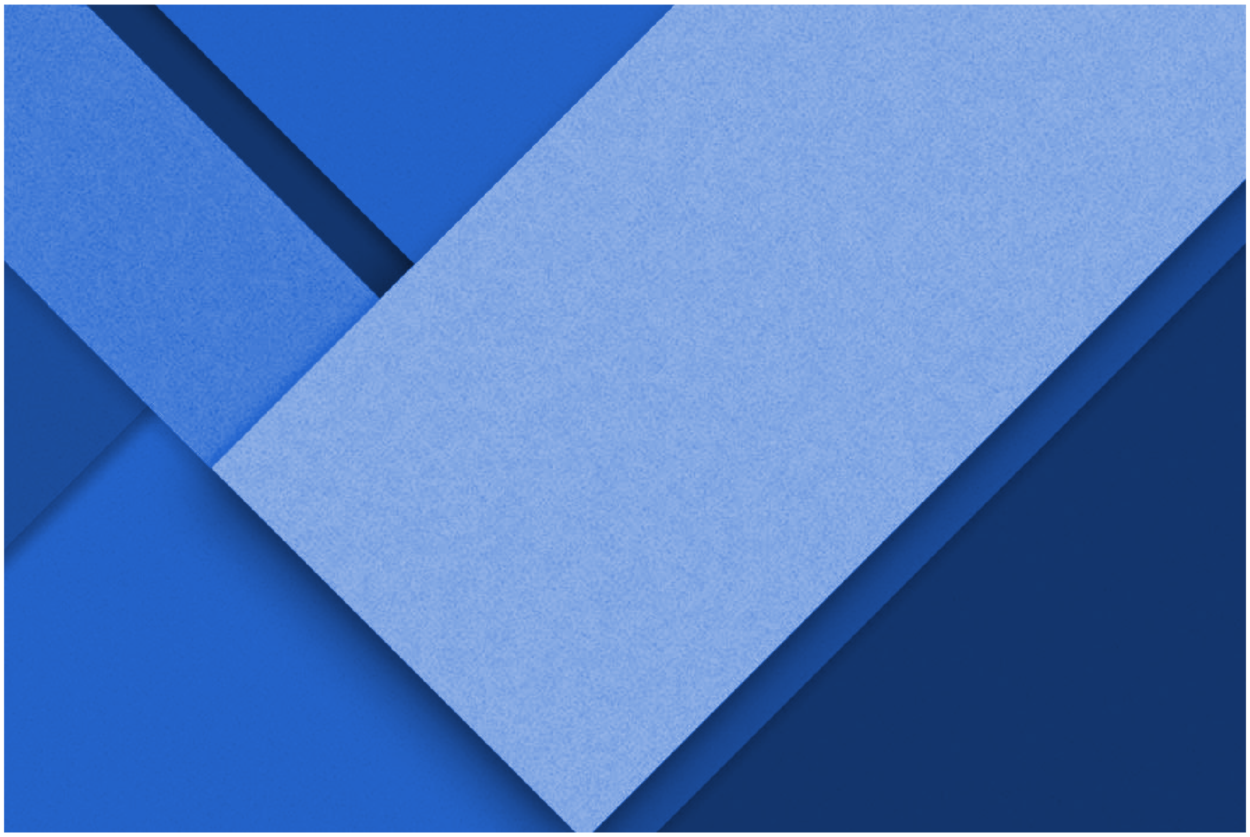


BIP32

26-01-2021

Team CPP {Maria Naseem, Mahnoor Ghous, Umar Farooq}
MediaPark SMC Pvt Ltd.
ISE Tower Islamabad.

Overview:

Bitcoin improvement proposal 32 is one of the most important BIPs we have. BIP 32 gave us Hierarchical Deterministic Wallets. That is, the ability to create a tree of keys from a single seed(Generated from BIP39).

Goals:

1. Build the complete understanding of Bip32
2. Proper flow of the algorithm considering c++ language.

Goals BIP32:

1. Have one master key to backup
2. Have good privacy by using separate addresses for each payment
3. Have the (option) for third parties to generate new receiving addresses automatically

Specifications:

The specification is intended to set a standard for deterministic wallets that can be interchanged between different clients.

The specification consists of two parts.

In the first part, a system for deriving a tree of keypairs from a single seed is presented.

The second part demonstrates how to build a wallet structure on top of such a tree.

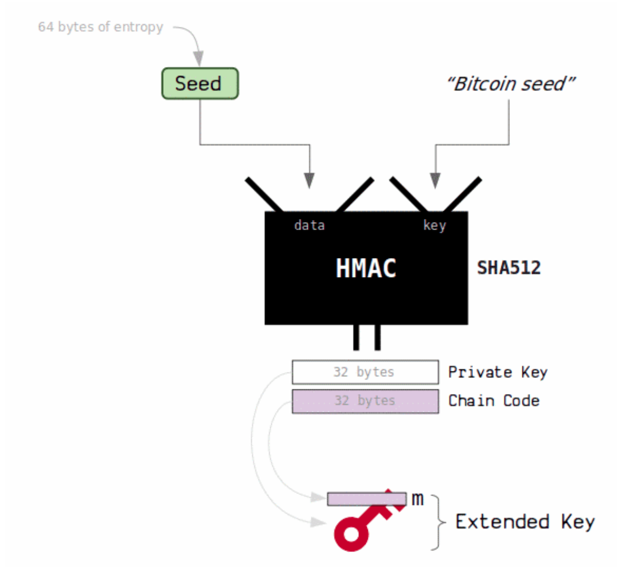
Milestones:

I. Master Key

The first goal is to convert the seed into what is called a *master key* or *root key*. “Root” is illustrative, in that all derivative keys will branch off from it

The first extended keys (master keys) are created by putting a seed through the HMAC-SHA512 hash function.

You can think of a HMAC as a hash function that allows you to pass data along with an *additional secret key* to produce a new set of random bytes.

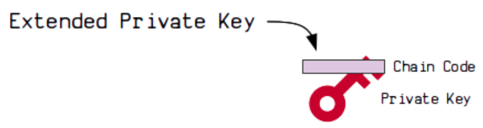


The HMAC function returns 64 bytes of data (which is totally unpredictable). We split this in to two halves to create our master extended private key:

1. The left half will be the private key, which is just like any other private key.
2. The right half will be the chain code, which is just an extra 32 bytes of random data.

Extended Private Key

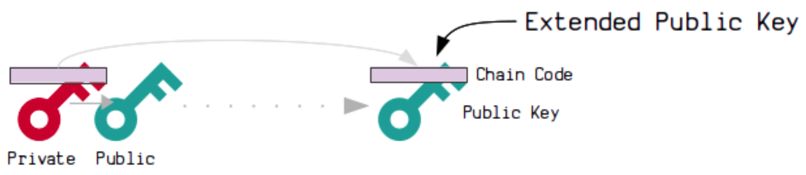
So an extended private key is ultimately just a normal private key coupled with a chain code.



A private key with an extra 32 bytes.

Extended Public Key

We can also create a corresponding extended public key. This just involves taking the private key and calculating its corresponding public key, and coupling that with the same chain code.



And there we have our initial master extended private key and master extended public key.

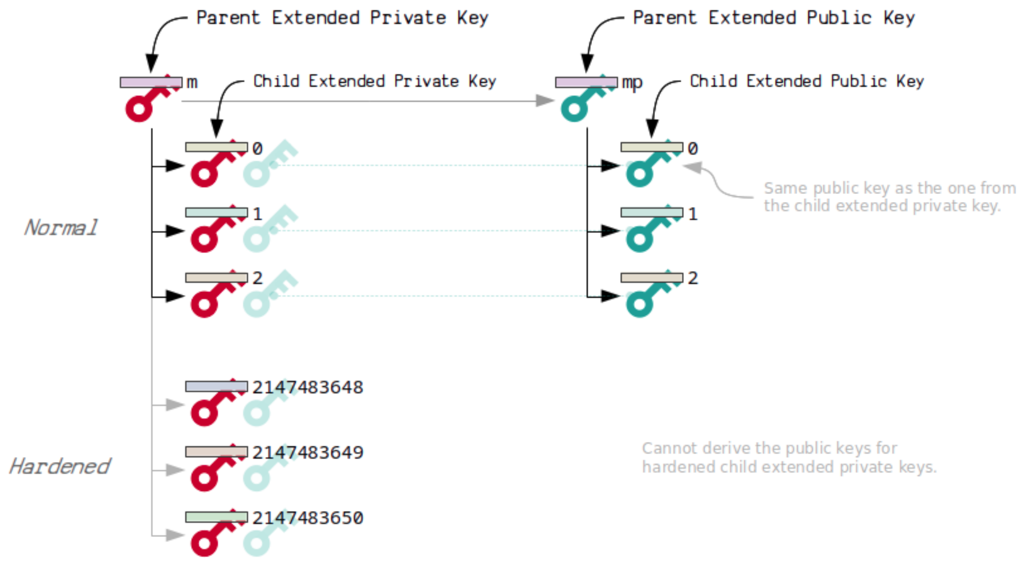
Tip: As you can see, extended keys are nothing special in themselves; they are just a set of normal keys that share the same chain code (an extra 32 bytes of entropy). The real magic of extended keys is how we generate their children.

II. Extended Key Tree

All extended keys can derive *child extended keys*.

- **extended private keys** can generate child keys with new private keys and public keys.
- **extended public keys** can generate child keys with new public keys only.

Each child also has an *index* number (up to 2**32).


Parent Extended Private Key
Parent Extended Public Key
m
Child Extended Private Key
mp
Child Extended Public Key
0
1
2
Normal
Same public key as the one from the child extended private key.
2147483648
2147483649
2147483650
Hardened
Cannot derive the public keys for hardened child extended private keys.

The cool thing about extended public keys is that they can generate the same public keys as the extended private key.

For security, you can derive two types of children from an extended private key:

1. Normal - The extended private key and extended public key can generate the same public key.
   Indexes 0 to 2147483647 (the first half of all possible children)
2. Hardened - Only the extended private key can generate the public key.
   Indexes 2147483648 to 4294967295 (the last half of all possible children)

In other words, a hardened child gives you the option of creating a "secret" or "internal" public key, as the extended public key cannot derive them.

III. Child Extended Key Derivation

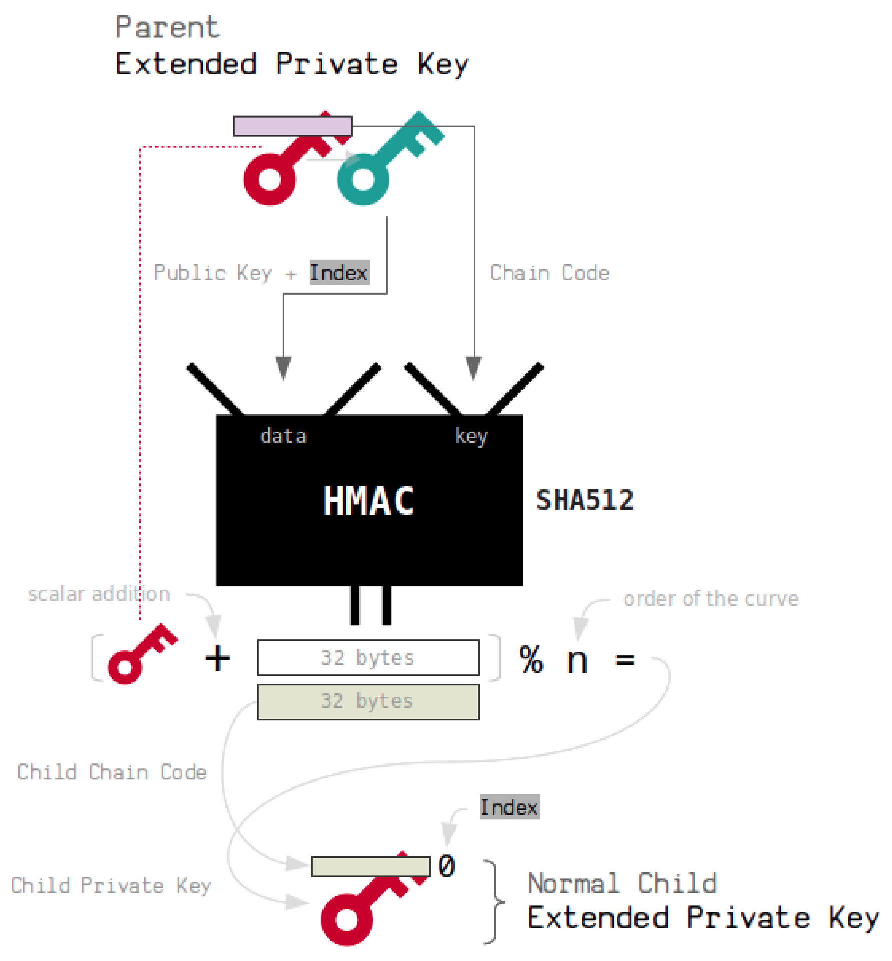
Both extended private keys and extended public keys can derive children, each with their own unique index number.

There are 3 methods for deriving child keys:

1. *Normal* Child extended private key
2. *Hardened* Child extended private key
3. *Normal* Child extended public key

Tip: Derived child extended keys (and parent keys) are independent of each other. In other words, you wouldn't know that two public keys in an extended tree are connected in any way.

1. *Normal* Child extended private key



"Scalar addition" just means to traditional arithmetic addition.

1. Work out the public key. (This is so a corresponding extended public key can put the same data into the HMAC function when deriving their children.)
2. Use an index between 0 and 2147483647. Indexes in this range are designated for *normal* child extended keys.
3. Put data and key through HMAC.
   - *data* = public key+index (concatenated)
   - *key* = chain code

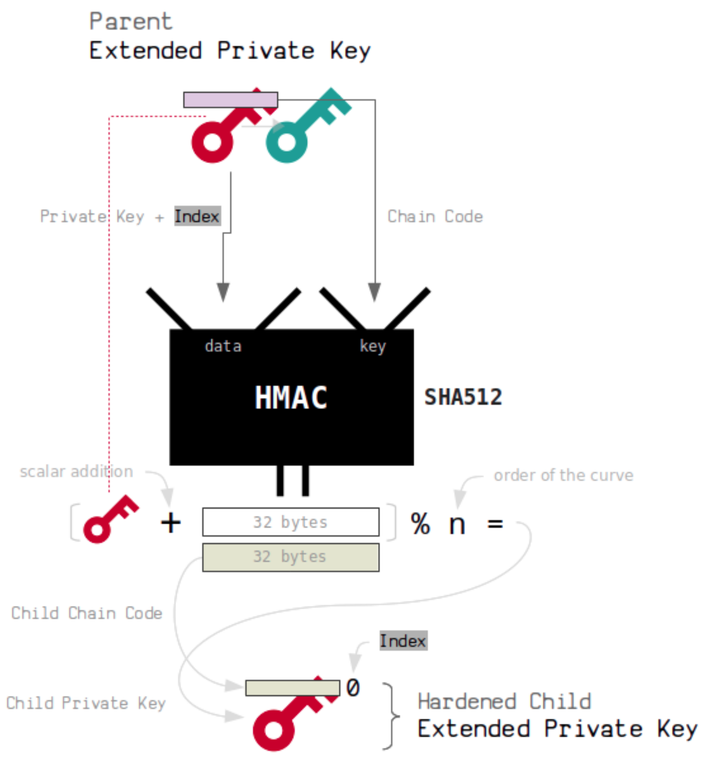
The new chain code is the last 32 bytes of the result from the HMAC. This is just a unique set of bytes that we can use for the new chain code.

The new private key is the first 32 bytes of the result from the HMAC added to the original private key. This essentially just takes the original private key and *increases* it by a random 32-byte number.

We modulus the new private key by the order of the curve to keep the new private key within the valid range of numbers for the elliptic curve.

So in summary, we use the data inside the parent extended private key (public key+index, chain code) and put it through the HMAC function to produce some new random bytes. We use these new random bytes to construct the next private key from the old one.

2. *Hardened* Child extended private key



1. Use an index between 2147483647 and 4294967295. Indexes in this range are designated for hardened child extended keys.
2. Put data and key through HMAC.
   - *data* = private key+index (concatenated)
   - *key* = chain code

The **new chain code** is the last 32 bytes of the result from the HMAC.

The **new private key** is the first 32 bytes of the result from the HMAC added to the original private key. This again just takes the original private key and *increases* it by a random 32-byte number.

However, this hardened child key was constructed by putting the private key into the HMAC function (which an extended public key does not have access to), which means that child extended private keys derived in this way will have a public key that cannot be derived by a corresponding extended public key.

Hardened derivation should be the default unless there is a good reason why you need to be able to generate public keys without access to the private key.

3. *Normal* Child extended public key

1. Use an index between 0 and 2147483647. Indexes in this range are designated for *normal* child extended keys.
2. Put data and key through HMAC.
    - *data* = public key+index (concatenated)
    - *key* = chain code

The **new chain code** is the last 32 bytes of the result from the HMAC. This will be the same chain code as the *normal* child extended private key above, because if you look back you will see that we put the same inputs into the HMAC function.

The **new public key** is the original public key point added to the first 32 bytes of the result of the HMAC as a point on the curve (multiplied by the generator to get this as a pont).

So in summary, we put the same data and key into the HMAC function as we did when generating the child extended private key. We can then work out the child public key via *elliptic curve point addition* with the same first 32 bytes of the HMAC result (which means it corresponds to the private key in the child extended private key).

4. *Hardened* Child extended public key

Not possible.

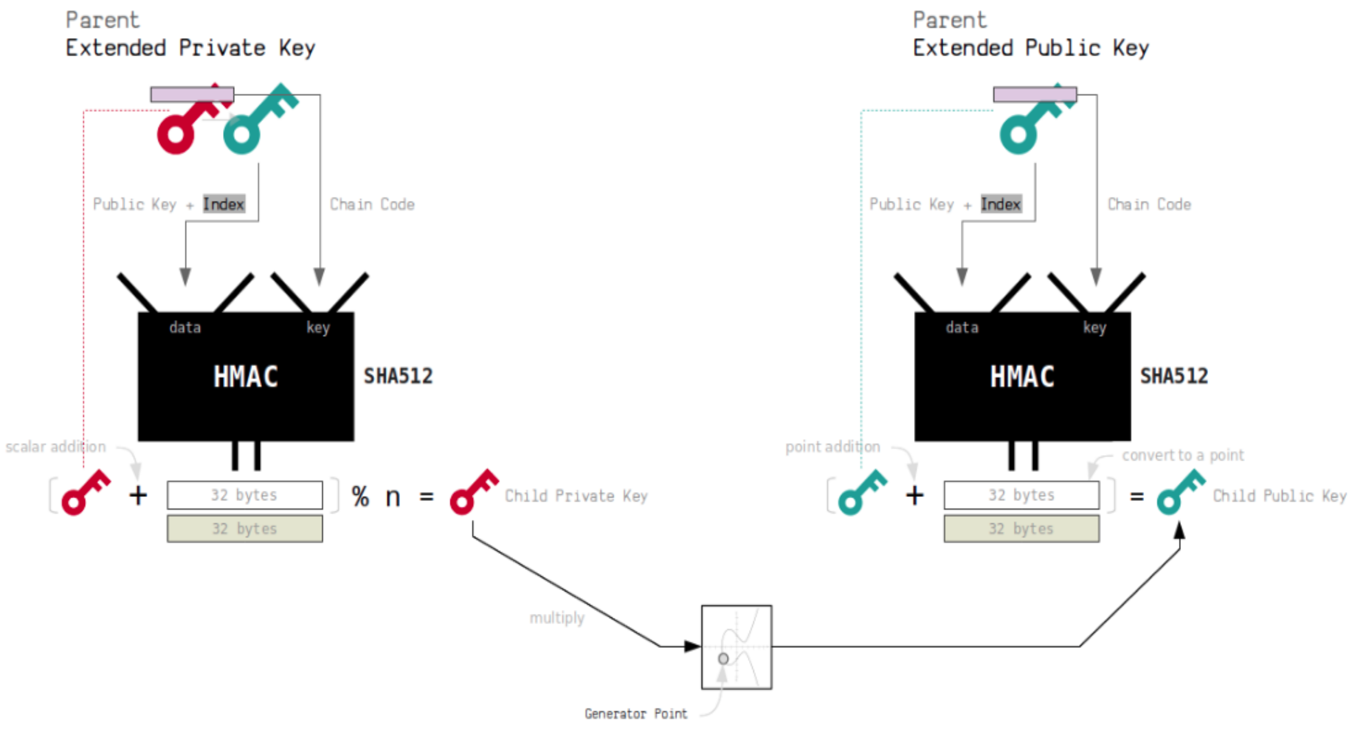
IV. Child Extended Key Derivation

In other words, how is it possible that a public key derived from an extended public key corresponds to a private key derived from an extended private key?

Well, for both child extended keys, we are putting the *same inputs* in the HMAC function, so we're getting the same data as a result. Using the first 32 bytes of this data (which is basically a number) we then:

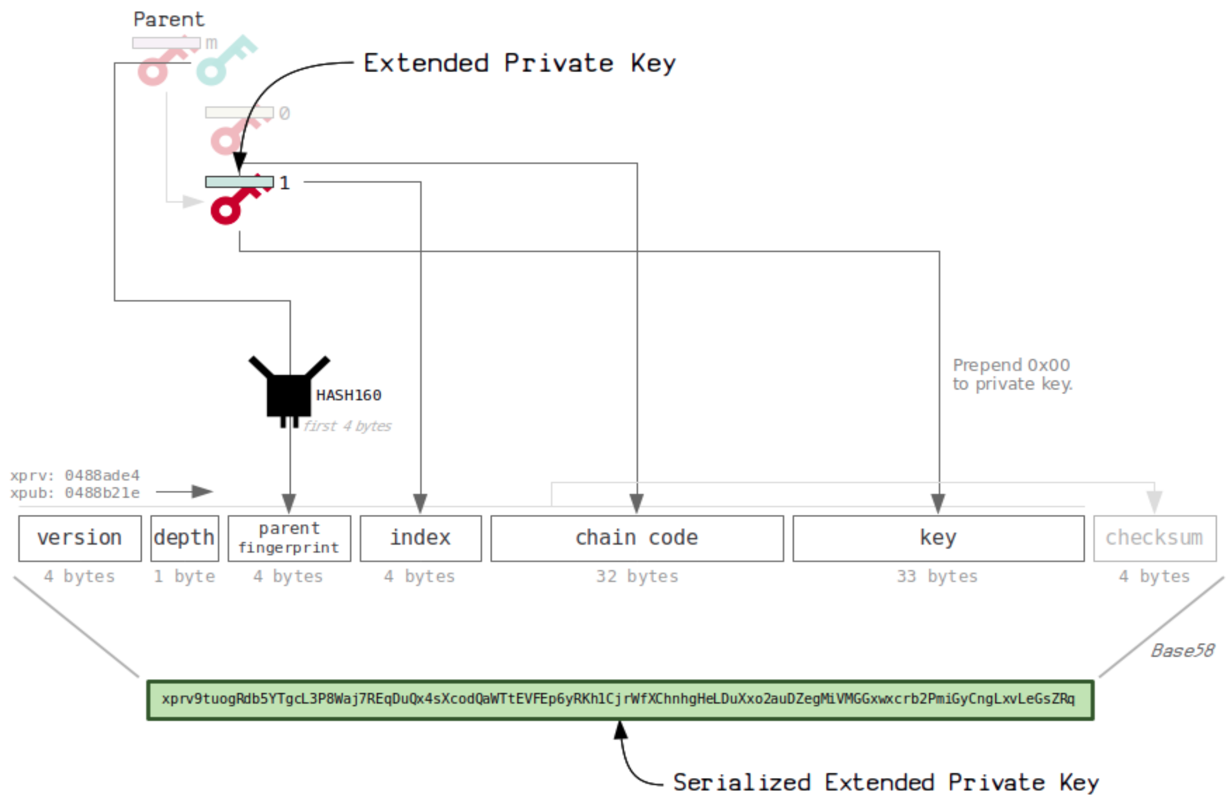
- *Increase* the parent private key by this number to create the **child** private key.
- *Increase* the parent public key by the same amount to create the **child** public key.

And due to the way elliptic curve mathematics works, the child private key will correspond to the child public key.



V. Serialization

An extended key can be serialized to make it easier to pass around. This serialized data contains the private key/public key and chain code, along with some additional metadata.



A serialized key contains the following fields:

4 bytes	**Version**	Places "xprv" `0488ade4` or "xpub" `0488b21e` at the start.
1 byte	**Depth**	How many derivations deep this extended key is from the **master key**.
4 bytes	**Parent Fingerprint**	The first 4 bytes of the hash160 of the parent's *public* key. This helps to identify the parent later.
4 bytes	**Child Number**	The *index* number of this child from the parent.
32 bytes	**Chain Code**	The extra 32 byte secret. This prevents others from deriving child keys without it.
33 bytes	**Key**	The private key (prepend `0x00`) or public key.

A checksum is then added to this data (to help detect errors), before finally converting everything to Base58 (to create a human-friendly extended key format).

An extended private key looks like this:

xprv9tuogRdb5YTgcL3P8Waj7REqDuQx4sXcodQaWTtEVFEp6yRKh1CjrWfXChnhgHeLDuXxo2auDZegMiVMGGxwxcrb2PmiGyCngLxvLeGsZRq

An extended public key looks like this:

```
xpub67uA5wAUuv1ypp7rEY7jUZBZmwFSULFUArLBJrHr3amnymkUEYWzQJz13zLacZv33sSux
KVmerpZeFExapBNt8HpAqtTtWqDQRAgyqSKUHu
```

As you can see they're pretty long, but that's because they contain extra useful information about the extended key.

Tip: The *fingerprint*, *depth*, and *child number* are not required for deriving child extended keys – they just help you to identify the current key's parent and position in the tree.

Note: The 4-byte field for the *child number* is the reason why extended keys are limited to deriving children with indexes between 0 and 4,294,967,295 (0xffffffff).