and the Argon2 parameters which SBK chose were `-p=2`, `-m=539` and `-t=26`. This might take 1-2 minutes to calculate on the old machine, but on a more modern system it may take only 10 seconds. For easy math and to be conservative, let's assume that an attacker has access to future hardware that can calculate one of these hashes in 1 second, further assume that they have unlimited access to 1000 systems of this caliber (and more money to spend on electricity than they could ever get from your wallet). After $\frac{2^{47}}{1000 \times 86400 \times 365} = 4500$ years they would have 50:50 chance to have cracked your wallet. It would be cheaper for them to find you and persuade you to talk. Beware of shorter keys lengths though: if you use `--brainkey-len=4` (32 bits), the same attacker would need only $\frac{2^{31}}{1000 \times 86400} = 25$ days.

All of this is assuming of course, that the attacker has somehow gained access to your `salt`. It may be OK for you to use a value lower than `--brainkey-len=8`, as long as you can satisfy one of the following conditions:

- You are confident that your `salt` will *never* be found by an attacker.
- If your `salt` is found, then you have some way to know that this happened, so that you will at least have enough time to move your coins to a new wallet.
- You regularly generate a new wallet and move all your coins, so every `salt` becomes obsolete before any brute-force attack has enough time to succeed.

5.5.2 Parameter Choices

There is no correct choice when it comes to picking parameters, there are only trade-offs. To a certain extent you can trade-off entropy with KDF difficulty: If you are willing/able to memorize a longer brainkey (with more entropy), you could reduce the KDF difficulty and thereby reduce your your hardware requirements and/or time to wait loading your wallet. If you are very confident that your salt will never be found, you could have a very short `brainkey`.

The default KDF difficulty and key lengths used by SBK are chosen based on the following reasoning: The main constraint is the ability of a human to memorize words. The brainkey length is chosen to be as short as possible, while still being able to offer some protection against a brute-force attack. Since the risk of you

SBK: Introduction

SBK is a tool to generate and recover Bitcoin Wallets. The goal of SBK is to keep your bitcoin[1] safe and secure. This means:

- Your wallet is safe, even if your house burns down in a fire and all of your documents and devices are destroyed.
- Your wallet is safe, even if all your documents are stolen or a hacker copies every file from your computer
- Your wallet is safe, even if you trusted some people you shouldn't have (not too many though).
- Your wallet is safe, even if something happens to you (at least your family can still recover your bitcoin).

The goal of SBK is to enable most people[2] to live up to the security mantra of Bitcoin: Your keys, your coins; not your keys, not your coins{1}.

SBK is Free Open Source Software. SBK is not a service, not a company and certainly not yet another token[3]. The only purpose of SBK is to generate and recover the keys to your wallet (i.e. the wallet seed). SBK is not a wallet itself, it only creates and recovers the keys for such wallets. The Electrum Bitcoin Wallet{2} [4][5] is currently the only supported wallet.

1.1 Disclaimers

No Warranty

The software is provided under the MIT License, "as is", without warranty of any kind, express or implied...[3]. In particular, the author(s) of SBK cannot be held liable for any funds that are lost or stolen. The author(s) of SBK have no responsibility (and very likely no ability) to help with wallet recovery.

Concerns regarding Shamir's Secret Sharing

I acknowledge the concerns expressed by Jameson Lopp[4] and in Shamir Secret Snakeoil[5]. To that end, I have placed these disclaimers as the first thing for you to read. Further info see the chapter on Tradeoffs

Project Status: Alpha

As of January 2020, SBK is still in the experimental, pre-alpha, evaluation only, developmental prototype phase (hedge, hedge, hedge). At this point the primary reason for the software to be publicly available is for review.

For the moment not even the primary author of SBK is using it for any substantial amount of bitcoin. If you do use it, assume that all of your bitcoin will be lost.

If you are looking for viable present day alternatives, please review How To Store Bitcoin Safely[6] by Dan Held.

Since the `--wallet-name` is chosen by you and since it is not encoded using a mnemonic or ECC data, there is a greater risk that it may not be possible to decipher your handwriting. To reduce this risk, the set of valid characters is restricted. Valid characters are lower-case letters `"a-z"`, digits `"0-9"` and the dash `"-"` character. In other words, the `--wallet-name` must match the following regular expression: `^[a-z0-9\-]+$`[40].

For more information on the risks and responsible use of a wallet passphrase, the trezor blog has a a good entry[41] for the equivalent passphrase feature of their wallet.

5.5 Key Derivation

The purpose of the Key Derivation Function[42] is to make a brute-force attack incredibly expensive. The purpose of the `salt` is to make each brute-force attack specific to a particular wallet. Using a KDF together with a `salt` makes it practically impossible to brute-force your wallet and if an attacker has access to your `salt`, then you will at least have some time to move your coins before their attack succeeds (assuming you used a strong `brainkey` with at least `--brainkey-len=6`).

The KDF used by SBK is Argon2[43], which is designed to be ASIC-resistant, GPU-resistant and SBK chooses parameters to use as much memory as is available on your air-gapped computer. This approach mitigates the advantage an attacker has from investing in specialized hardware. The price you pay for this added security is that you have to wait a minute or two every time you want to load your wallet. This shouldn't be too much of an issue if you access your cold-storage wallet only every few weeks or months.

5.5.1 Brute Force Attack Hypothetical

Some back of the envelope calculations to illustrate the difficulty of a brute-force attack: If we assume the attacker has gained access to your `salt`, then they will have a 50% chance of loading your wallet if they can calculate 2^{47} hashes. Let's assume you used an average computer from 2012 as your air-gapped computer

> Aside: When parsing a share it is critical to verify that `x != 0` to prevent a forced secret attack, as described in point 3 of the "Design Rational" of SLIP-0039[39].

5.4 Wallet Name/Passphrase

A `--wallet-name` is effectively a passphrase, so it suffers from the same problem as all passphrases: they can be forgotten. One of the main purposes of SBK is to protect your wallet from being lost through any single point of failure:

- If you lose the `salt`, you have a backup.
- If you forget your `brainkey` you have a backup.
- If a `share` is partially unreadable, the error correction data provides redundancy.
- If a `share` is destroyed completely, there is redundancy in the form of other shares.

Please remember that you are at a much greater risk of loosing your bitcoin through a user error than you are from hacking or theft. The use of a `--wallet-name` can make all of SBK's protections null and void, if you use it to inadvertently introduce a single point of failure:

- If you forget the `--wallet-name` and you're the only person who ever knew it, then your wallet will be lost.
- If you write it down on a single piece of paper, and that piece of paper is destroyed, then your wallet will be lost.

To avoid such a single point of failure, the default value for `--wallet-name` is hard-coded to `empty` (literally). There are some legitimate reasons to use a `--wallet-name`, but if you do use it, do not treat it as a password. Instead, write it down in clear handwriting and make sure it is available when your wallet has to be recovered, for example by writing the wallet name(s) on some or all of the shares.

1.2 The Many Ways to Lose Your Coins

In the broadest sense, there are two ways for you to lose control of your bitcoin:

1. Loss: Your keys can be **lost**[6] and your wallet has effectively become a black hole.
2. Theft: Your keys can be leaked and somebody else will **steal** your bitcoin.

Your keys may be **lost** if they are vulnerable to any single point of failure (SPoF). This might be the case if:

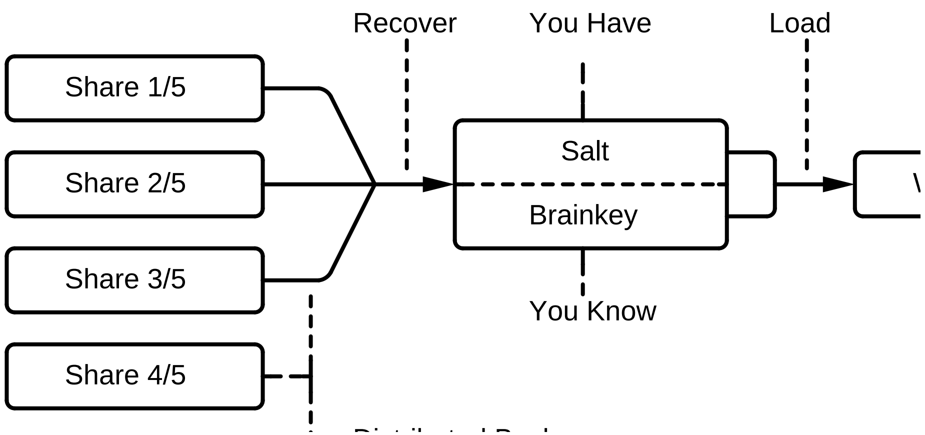
- Your 12-24 word written seed phrase is destroyed (e.g. due to fire or water damage).
- You forget the password that you used to encrypt your wallet.
- Your hard-drive fails and have no backup or seed phrase to recover.
- You have your keys, but you no longer have access to the software needed to use them.

Somebody might **steal** your keys if:

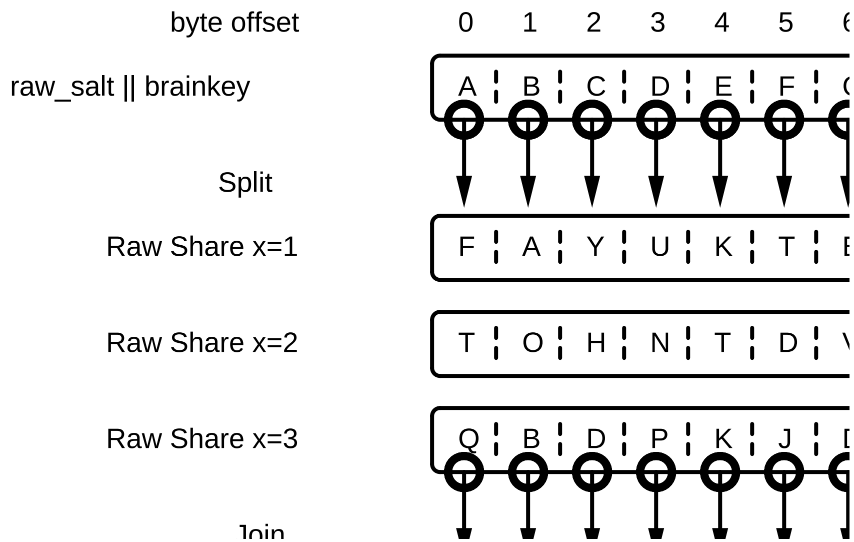
- You use a web wallet from an untrustworthy or negligent service provider or exchange.
- You use a computer that is connected to the Internet and has a vulnerable back-door
- Your wallet uses a written seed phrase, which a thief can find.
- You sell your computer or a hard drive, from which you didn't erase your wallet in a secure way.
- Your seed was generated in a predictable way, due to a software bug or a lack of entropy.

SBK is designed to protect against and mitigate these risks as much as possible. Most of the design choices are made to protect against the many kinds of human failure. Use of SBK may be tedious, but be aware that the design choices have one primary purpose: Peace of Mind. Yes it may be tedious to write down dozens and dozens of words once, but you won't lie awake at night with worry.

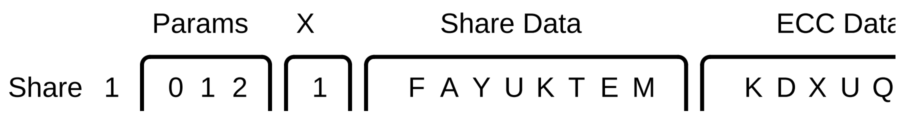
1.3 How SBK Works



5.3.3 Share Data



Shares are generated from the `shares_input` (`raw_salt || brainkey`). The split algorithm is applied to each byte separately and the points that make up each `raw_share` all have a common x-coordinate. In the preceding diagram for example, the first raw share would be 8 bytes represented here as `FAYUKTEM`, each letter representing the encoded y-coordinate for a byte. In order to recover the byte at `offset=7` of the `master_key`, we would use the join algorithm with the points `P(x=1, y=M)`, `P(x=2, y=W)` and `P(x=3, y=Z)`, to produce `P(x=0, y=H)`, where `H` represents the last byte of the `master_key`.



The "full" `share` also includes the serialized parameters as a prefix in the first four bytes, and it also includes ECC data of the same length as the `raw_share`. The ECC code used is a Reed-Solomon code.

The bean counters among you may have notice that 4 bytes is not enough to encode the complete range of valid parameters which the KDF would accept in theory. For example, the `kdf_time_cost`, which corresponds to the "Number of iterations *t*" in section 3.1 of the Argon 2 Spec[38] with a valid range of `1..2**32 - 1` would by itself already require 32 bits, much more than the 6bits available in the above encoding.

Since the distinction between 1000 iterations and 1001 iterations is not critical, the kdf parameters are not encoded exactly, but using a logarithmic scale. This log base is chosen so that the difficulty can be controlled reasonably well (increments of 1.25x) while still being able to represent values that are sufficiently large (`kdf_mem_cost` over 1 Terabyte per thread; `kdf_time_cost` over 1 million iterations). If you specified `--time-cost=1000` for example, this would be rounded to `floor(1.25**31 + 31) == 1040`.

Field Name	Size	Value	Range (inclusive)
`f_version`	4 bit	Hard-coded to `0`.	
`f_threshold`	4 bit	`threshold - 1`	1..16
`f_kdf_parallelism`	4 bit	`log2(kdf_parallelism)`	1, 2, 4, 8..32768
`f_kdf_mem_cost`	6 bit	`log(kdf_mem_cost) / log(1.25)`	1, 2, 3, 4, 6, 9, 12, 16...
`f_kdf_time_cost`	6 bit	`log(kdf_time_cost) / log(1.25)`	1, 2, 3, 4, 6, 9, 12, 16...

SBK has two ways for you to access your wallet, one for normal use and the other as a backup:

1. `Salt + Brainkey`: The `Salt` is a secret, very similar to a traditional 12-word wallet seed. It is written on a piece of paper and kept in a secure location, only you (the wallet *owner*) **have** access to. By itself, the `salt` is not enough to load your wallet. To load your wallet, you must also **know** your `brainkey`. A `brainkey` is passphrase which *only you know* and which is not stored on any computer or written on any piece of paper. In other words, the `brainkey` is in your brain and *only* in your brain.
2. `Shares`: A single `share` is one part of a backup of your wallet, written on a piece of paper or in some other physical form. When you combine enough `shares` together (e.g. 3 of 5 in total), you can recover your wallet. In such a scheme, any individual `share` is neither necessary nor sufficient to recover your wallet. This property is made possible by the Shamir's Secret Sharing[7] algorithm, which is used to generate the `shares`. You can distribute these in secure locations or give them to people whom you trust. Each `share` is useless by itself, so you don't have to place complete trust in any individual, location or institution. Not every `share` is required for recovery, so even if a few of them are lost or destroyed, you can still recover your wallet.

Using the `salt` and `brainkey`, you have direct access to your wallet, independent of any third party and with minimal risk of theft. The greatest risk you are exposed to here is that somebody might steal your `salt` and then additionally coerce you to reveal your `brainkey` (i.e. a $5 wrench attack[8]). This is in contrast to a typical 12-word wallet seed written on a piece of paper, which represents a single point of failure: If such a seed is lost, stolen or destroyed, your wallet is gone with it. In contrast to this, if you either forget your `brainkey` or if your lose your `salt`, then you can still recover your wallet from your backup `shares`.

Put differently, the regular way for you to access your wallet is secured by two factors: something you **have** (your `salt`) and something you **know** (your `brainkey`). To protect against loss of either one of these (as well as your untimely demise), you have a backup that is distributed in vaults, safety deposit boxes, hiding places and/or with trusted family and friends.

Term/Notation	Meaning
`kdf_input`	`kdf_input` = `master_key` || `wallet_name`
`wallet_seed`	The Electrum seed derived from the `kdf_input`.

For those keeping track, by default the total entropy used to generate the wallet seed is `12 + 4 == 16 bytes == 128 bits`. The 4 bytes of the `parameters` are not counted as they are very predictable.

5.3.2 Parameters

Any change in the parameters used to derive the wallet seed would result in a different wallet seed. This means that the parameters are just as important to keep safe as the `salt` itself. So we must either encode the parameters and keep them together with the `salt`, or we have to make them hard-coded constants in SBK itself. The latter would not allow you to choose a difficulty that is appropriate to your machine and level of paranoia, so parameters are not hard-coded. Instead they are encoded as a prefix of the `salt` and of every `share`. The downside of this is that there is more data that you have to manually copy and enter. This is why the encoding is kept as compact as possible (4 bytes == 4words == 2 x 6 digits).

Here is the data layout of these 4 bytes:

```
offset  0     3 4     7 8    11 12       17 18       23 24            31
        [ ver ] [thres] [kdf_p] [ kdf_mem ] [ kdf_time] [   share_no   ]
         4bit    4bit    4bit     6bit        6bit          8bit
```

> Aside: The `salt_len` is not an encoded parameter. Instead it is hard-coded to 12 bytes (96 bits). The brainkey adds another 32 bits of entropy.
>
> Aside: While the `threshold` is encoded, `num_shares` is not, as it is only used once when the `shares` are first created. It is not needed for recovery, so it is not encoded in the `parameters`.

Term/Notation	Meaning
`version`	Version number to support iteration of the data format.
`threshold`	Minimum number of shares required for recovery. min: 1, max: 16, default: 3
`num_shares`	The number of shamir shares to generate from the `master_key`.
KDF	Key Derivation Function. The algorithm used by SBK is Argon2.
`kdf_parallelism`	The degree parallelism/number of threads used by the KDF.
`kdf_mem_cost`	Amount of memory in MiB filled by the KDF.
`kdf_time_cost`	Number of passes over the memory used by the KDF.
`parameters`	4 byte encoding of parameters required by `sbk load-wallet`.
`||`	Concatenation operator: `"abc" || "def" -> "abcdef"`
`raw_salt`	12 bytes of random data (main source of entropy for the `wallet_seed`).
`salt`	`salt = parameters || raw_salt`
`brainkey`	Random data memorized by the owner of the wallet.
`shares_input`	`shares_input = raw_salt || brainkey`
`raw_share`	Encoded points in $\mathrm{GF}(256)$. See Share Data
`share`	`share = parameters || raw_share`
`master_key`	`master_key = salt || brainkey`
`wallet_name`	Identifier to generate multiple wallets from a single `master_key`.

Bitcoin is Sovereignty

You have more control over your bitcoin, than anything else in the world. Everything else can be taken from you against your will. Your posessions can be taken from you or destroyed, you can be evicted from your house, you can be thrown in a cage and ultimately you can be killed. There is nothing in your power that can provide an ultimate defence against any of these violations of your physical property. Bitcoin is different. You may well be thretened, somebody may try to extort you, but given some preparation and enough strength of will, nobody other than you can determine what will happen with your bitcoin. Even in the case of your death, you can arrange for them to be passed on according to your will and your will alone.

The promise of Bitcoin is to empower every individual to be sovereign over the fruits of their labor. No matter how weak and alone you are, Bitcoin is the beginning of a world where you are your master and nobody's slave. Bitcoin can deliver people from the shackles of the fiat printing press, the whip that commands their wealth, the leach that drains their life at the push of a button. The power of fiat money inevitably corrupts those who wield it, even those with the best of intentions. Even if we were assured this power would only ever be wielded by angles, the downfall of central bank administrators is how little they really know about what they imagine they can design[9].

Every person who would like to earn an honest living, save for their retirement and leave something for their children should be free to do so. We will live in a better world if they need not fear theft, be it by crooks, by buerocrats, by rulers or by a plurality of their fellow men. The world will prosper, when any man can cross a border and unlike his forefathers, who would arrive without a penny to their name, carry with him the fruits of his labor and start a new life.

The cliche argument against a crypto nerd, who fantasizes about fanciful cryptographic security, is that the bad guys will knock down his door and break his nuckles until he starts to talk. This attack is a valid concern, but its major flaw is that it does not scale. If bad guys attack an individual they may face a strong willed man, they may second guess themselves if he has plausible deniability, and in any case, it will be expensive for them to scale this attack to everbody who may or may not have lost their bitcoin in a boating accident.

Fanciful language asside, Bitcoin is not a game. SBK is about individual sovereignty. Your keys in your head, but without the risk that your wealth will be lost if you die. With SBK you can cross a border and look just like anybody else, but you carry with you the fruits of your labour. This is a powerful force for freedom in this world then he will have strength in numbers.

2.1 Hyperbitcoinization

...

2.2 Multi-Signature Wallet

SBK is simple: Something you have and something you know. Multisig is not as simple. You have multiple things, you need to do multiple signatures, ideally on multiple machines in different places. Complexity increases the risk of loss as less technically adept users will shut down more quickly or be more easilly tricked by scammers.

The main use case for SBK is for personal custody of bitcoin.

While multiple people may be involved to provide the distributed backup of your wallet, only you as the owner will have a claim on their bitcoin. If your use-case involves multiple people, with joint custody, you should instead use n of m multisig[10] transactions[11].

5.3 Implementation Details

5.3.1 Terms and Notation

$GF(2^{160-47})$. As you may see, this number exceeds the native integer representation of most computer architectures, which is one of the main reasons this approach typically isn't used.

In principle it would have been fine[11] for SBK to use $GF(p)$, but since other implementations typically use $GF(256)$ and innovation in cryptography is usually not a good thing, this is what SBK now also uses. The specific field used by SBK has been broadly studied already, which should make validation easier, even though the requirement for polynomial division makes arithmetic a bit harder to follow. The specific field uses the Rijndael irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, which is the same as SLIP0039[36] and (perhaps more importantly) AES/Rijndael[37][12].

2.3 SPoF: Single Point of Failure

The single point of failure that remains with SBK is with the recovery process.

2.4 Alternatives and Comparison

Method	Pro	Con
Seed Phrase	Simple	SPoF
Multisig	Complex	SPoF when transported
Hardware Wallet		
Hardware Wallet + Multisig		
Warp Wallet	Simple	No backup
SBK	Simple	SPoF when recovering

Method	Airgap Needed	SPoF	Simple	SPoF (Theft)	Tedious
Seed Phrase	Yes	Yes	Yes	Yes	
Hardware Wallet	No	Yes	Yes	No	
Multi-Sig	No	No	No	Yes	
Multi Hardware Wallet	No	No	No	Yes	+++
+ Multi-Sig					
SBK	Yes	No	Yes	No	++

2.5 Warp Wallet

SBK has similarities to warp wallet[12], except that it has an additional backup using Shamir's Secret Sharing. It is also similar to Shamir Backup[13] developed by SatoshiLabs s.r.o.[7], except that it uses a brainkey.

2.6 Trezor: Shamir Backup

TODO

> Figure 5d
>
> Please forgive the limitations of the diagram software, the graph is supposed to represent a parabola with minimum roughly at `x=3`.

Using this approach, we can

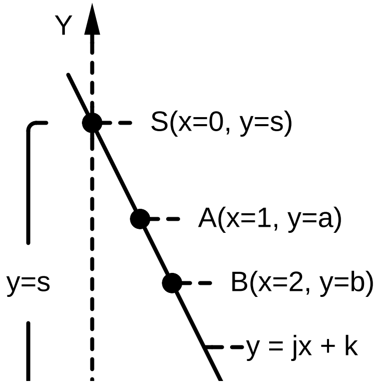
1. Encode a `secret` as a point: `S(x=0, y=secret)`
2. For a polynomial $y = ix^2 + jx + k$ which goes through `S`, we choose `k=secret` and random values for `i` and `j`.
3. Calculate 5 points `A`, `B`, `C`, `D` and `E` which lie on the polynomial (but which **crucially are not** at `x=0`, which would cause the secret to be leaked).
4. Use polynomial interpolation to recover `S` using any 3 of `A`, `B`, `C`, `D` or `E`.

The degree of the polynomial allows us control of the minimum number (aka. the `threshold`) of points/`shares` required to recover the secret. Calculating redundant `shares` allows us to protect against the loss of any individual `share`.

5.2.3 SSS: Choice of Galois Field

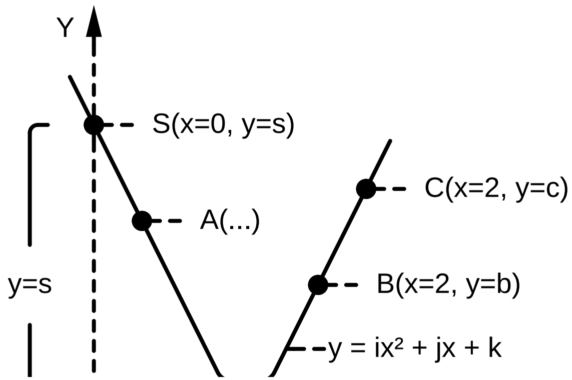
There is more to the story of course. My understanding is that the preceding scheme (which uses the traditional Cartesian plane) does not offer complete information security. Some information about the secret is leaked with each `share` and while an attacker who knows fewer points than the `threshold` may not be able to instantly determine the secret, they could at least derive some information to reduce their search space. I'm taking the cryptographer/mathematicians by their word that the solution is to use finite field arithmetic{34}.

Rather than calculating inside the Cartesian plane, we use either $\mathrm{GF}(p)$ (p being a prime number) or $\mathrm{GF}(p^n)$ (p^n being a power of a prime number, typically $\mathrm{GF}(2^8)$ aka. $\mathrm{GF}(256)$). In a previous iteration of SBK, $\mathrm{GF}(p)$ was used, with a value for p that corresponds to the level of entropy of the `brainkey`. The list of primes was from the largest that could satisfy $2^n - k \leq 2^n$ oeis.org/A014234{35}. For the default secret length of 20 byte/160 bit this would have been



Note that the parameter `j` is generated randomly and `k` is our secret `s`, so that if `x=0` then `y=s`. Recall that a polynomial of degree 1 is fully specified if you have any two distinct points through which it goes. In other words, if you know `A` and `B`, you can derive the parameters `j` and `k` of the equation `y = jx + k` and solve for `x=0` to recover `y=s`. If on the other hand, you have *only* `A` or *only* `B`, then there are an infinite number of lines which go through either. In other words, it is impossible to derive `S` from `A` alone or from `B` alone. To complete the picture, we could generate a further point `C`, so that we only require any two of `A`, `B` and `C` in order to recover `S`. This allows us to create a `2of3` scheme.

Similarly we can create a `3ofN` scheme with a polynomial of degree 2 (aka. a quadratic equation, aka. a parabola), a `4ofN` scheme with a polynomial of degree 3 (aka. a cubic equation) and so on.



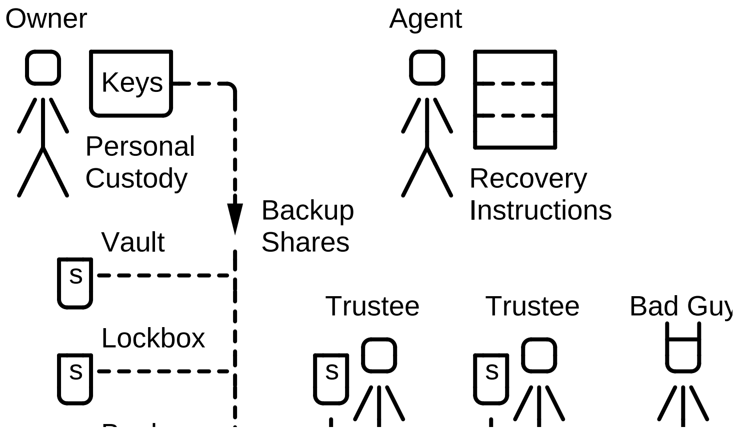
SBK - User Guide

The great thing about Bitcoin is that *you have complete control* of your own money: you are your own bank. The terrible thing about Bitcoin is that *you are responsible* for your own money: you are your own bank.

Unlike traditional financial systems, if your bitcoin is lost or stolen, you have no recourse to any institution whatsoever. No government, no bank, no company and no programmer has any obligation or even any ability to help you if something goes wrong.

The goal of SBK is to make it more easy for you to bear this burden of responsibility. SBK is designed for individuals who want to take **personal custody**[8] of their bitcoin while mitigating the risk of loss or theft.

3.1 Roles



There are four different roles involved with an SBK wallet:

1. **Owner**: You own some bitcoin that you want to protect from loss and theft.
2. **Agent**: The owner has instructed you to act on their behalf, should they not be able to.
3. **Trustee**: The owner has given you an SBK share, which is part of a backup of their wallet.
4. **Bad Guy**: You know of

No matter your role, you should make an effort to be dilligent. SBK may be built with redundancy, but it would be foolish to lean too much on that protection. If enough trustees neglect their responsibilities (e.g. by assuming that there are other trustees who are diligent enough), then the backup shares may become worthless and the wallet will be lost. Do not succumb to the moral hazard of trusting that others will do it better than you. Who knows, perhaps the last will of the owner has a clause regarding those who were negligent…

3.2 Tasks

3.2.1 Owner

As the owner of an SBK wallet, you generate the salt and brainkey, create the backup shares and make preparations so that your wallet can be recovered in a worst-case scenario.

3.2.2 Agent

As the agent of the owner, it is your responsibility to facilitate the recoverery their wallet and This may include the recovery of the owners wallet (in cooperation with the trustees) and the distribution of the bitcoin according to the wishes of the owner.

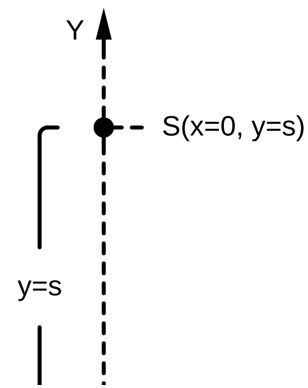
5.2.2 SSS: Shamir's Secret Sharing

With SSS, a key can be split into `shares` such that each `share` is completely independent of every other. Assuming `--scheme=3of5`:

1. Any two `shares` can be lost and the remaining three are enough to recover the original key.
2. Any individual `share` (or subset of `shares` below the `threshold`) is useless. This means that access to fewer than three `shares` does not provide an attacker with any advantage if they attempt to brute-force a wallet seed.

To get an intuition of how SSS works and why it is secure, it is enough to recall some high-school calculus.

Consider a point `S(x=0, y=s)` on the Cartesian plane, where the coordinate `y=s` is your secret encoded as a number:



Now consider $y = jx + k$, a polynomial of degree 1 (aka. a linear equation, aka. a line equation) which goes through point `S` and further points `A(x=1, y=a)` and `B(x=2, y=b)`.

If for example you have a wallet seed of 12 bytes `"abcd efgh ijkl"` (with 96 bits of entropy), you could split it into fragments: `"1: abcd"`, `"2: efgh"`, `"3: ijkl"`. This way each fragment (by itself) is not enough to recover your wallet. The downside is that you increase the risk of losing your wallet: If you lose even one fragment, you also lose the wallet.

To reduce this risk, you might want to add redundancy by making more fragments: `"4: cdef"`, `"5: ghij"`, `"6: klab"`. Now if fragment 1 is lost, you may still have access to fragment 4 and 6 from which you can still recover the secret.

There are two downsides to this approach:

1. Some of the fragments may be identical or have overlapping parts, so the redundancy is not as great as you might hope: Two fragments could be lost and if they are the only ones with a specific part of the secret (for example fragment 1 and 4 are the only ones with the bytes `cd`), then you may have lost your wallet, even though you have 4 other fragments that are perfectly preserved.
2. If a fragment falls in the hands of an attacker, they can try to guess the remaining 8 bytes, which leaves a search space of 2^{64} as opposed to the full 2^{96}. If you have wrongfully trusted two people, and they collude with each other (which they have a financial incentive to do), then they may have only 2^{32} combinations left for their brute-force search.

There may be slightly more clever schemes along these lines, but I won't go into them, as this was just to serve as a motivation for the more complex but better alternative used by SBK: Shamir's Secret Sharing.

3.2.3 Truestee

A person or institution who has custody of an SBK share, which is part of a wallet backup. You should keep this share *safe*, *secret* and *secure* so that it will be available if the owners wallet has to be recovered.

3.3 Minimal Owners Guide

I will start with a bare-bones guide for how to use SBK. It is written with the assumption that you are mostly worried that your wallet will be lost, for example due to a fire, software virus, hardware failure or your untimely demise.

If all you want is a geographically distributed backup of your wallet (to protect against loss and accidents), then this minimal guide may be enough for you. If you are additionally worried that some people that you currently trust might betray you (which is where things get complicated), then you should continue reading the full user guide.

3.3.1 Deciding on a Scheme

The first thing to do, as an owner, is to decide on a "scheme". This is the threshold `T` and number of backup shares `N`, controlled using `--scheme=TofN` when you initially create your wallet. The first parameter `T` is the *threshold*, which is the minimum number of shares that are required to recover your wallet. The second parameter `N` is the total number of shares that are created.

The default scheme is `3of5`. With this scheme:

- To recover your wallet, you will need at least 3 backup shares.
- You will not be able to recover your wallet, if more than 2 shares are lost or destroyed.

For `T` parameter in `--scheme=TofN`, you should consider the worst-case scenarios:

- How many backup shares could be destroyed at once?
- How many backup shares could bad actor collect?

You may well have geographically distribute your backup shares, but if they're written on paper and kept in an area that is prone to be flooded, then you may lose too many of them at once. If the child of a trustee can find a share in their houshold and in addition is at some point a guest in your house, where they also find a backup share, then it would be better if you have a threshold set to `T=3` or higher.

For the parameter `N` (the total number of shares), you should consider how many SBK shares you expect will be lost in a worst-case scenario. If you expect the recovery to be done years after the wallet was created, then you should assume that some of the shares will be lost, forgotten about or destroyed, even despite your best efforts and your trustees to choose secure locations.

If you expect at most 2 shares to be lost, then you should choose `N=T+2`. This means, if you have decided on `T=3` then you should choose `N=5`. With this scheme, if either your salt or brainkey are lost and also two backup shares are lost, then the remaining three shares will still be enough to recover your wallet.

> **T=1 is Stupid**
>
> If you were not worried that any share would ever fall into the hands of a bad actor, then you could set a threshold to `T=1`. In that case however, you may as well not bother to use SBK and instead just create a normal Bitcoin wallet with the usual 12-word wallet seed. For any redundancy you need, you can just make duplicate copies of the seed.

5.1.3 Loading Wallet

You can load the wallet if you have the `salt` and `brainkey`, either directly as the owner, or after you have recovered them from the backup `shares`.

1. Invoke the `sbk load-wallet` command.
2. Optionally specify a `--wallet-name`.
3. Enter the `salt` and `brainkey`.
4. The `wallet-seed` is derived using the KDF.
5. The Electrum Wallet file is created in a temporary directory (in memory only if supported).
6. The Electrum GUI is started in offline mode (use `--online` if you are not using an air-gapped computer).
7. Use wallet/sign transactions...
8. Once you close the wallet, all wallet files are overwritten and deleted[31].

5.2 Shamir's Secret Sharing

This section describes how the `shares` are generated.

> Aside: Since the writing of this section, two nice introduction videos to secret sharing have been published. One is Secret Sharing Explained Visually by Art of the Problem[32] and another is How to keep an open secret with mathematics. by Matt Parker/standupmaths[33].

5.2.1 Prelude: Naive Key Splitting

It's fairly obvious why you might want to split a secret key into multiple parts: Anybody who finds or steals the full key will have access to your wallet. To reduce the risk if being robbed, you can split the key into multiple parts. If somebody finds such a fragment, it will not be enough to access your wallet.

5.1.1 Key Generation

Steps involved in key generation:

1. Invoke the `sbk create` command.
2. Optionally specify `--scheme` (default is `"3of5"`, for a total of 5 shares, any 3 of which are enough for recovery).
3. Optionally specify `kdf-parameters`. These are `-p / --parallelism`, `-m / --memory-cost` and `-t --time-cost`. If not specified, these are chosen automatically based on the available memory and processing resources of your system.
4. The `salt` and `brainkey` are randomly generated.
5. The `shares` are generated from the `salt` and `brainkey`.
6. The mnemonic encoding for each of the above secrets is shown for the user to copy onto paper (or memorize in the case of the `brainkey`).

5.1.2 Key Recovery

Let's assume that you've already forgotten your `brainkey`, or that your handwriting is so bad that you can't read your `salt` anymore. To recover both, you can join/combine the backup `shares`:

1. Invoke the `sbk recover` command.
2. Enter as many `shares` as required.
3. The `shares` are joined using Shamir's Secret Sharing and the resulting secret is split into the `salt` and `brainkey`.
4. Write down `salt` and `brainkey`.

Note that the wallet is not loaded directly, instead the recovery produces the `salt` and `brainkey`. Loading the wallet is a separate step.

3.3.2 Preparation and Materials

Once you've decided on a scheme and after you have made plans about where you will keep your backup shares and who will be your trustees and agents, it's time to prepare some materials. To create a wallet, you will need the following:

- A download of the bootable sbklive_x64.iso image.
- A USB flash drive or SD card with at least 2GB
- A PC/Laptop
- A program to create a bootable flash drive, such as rufus.ie[14] on Windows or USB-creator on Ubuntu[15].
- A printer (with ink and paper of course)
- A ballpoint pen (or anything similar as long as the ink is not erasable)
- A stapler or adhesive tape

There are more materials that you could prepare to make your shares more robust, but this will do for now to understand the basic idea.

3.3.3 Air-Gapped System

Ideally you should use a computer that is dedicated to your wallet and nothing else. Every other use case, especially anything that involves a connection to the internet will increase the risk to your wallet. For use with SBK this computer should satisfy the following:

- It has no network card or wifi card (over which any keys could be transmitted to a bad actor).
- It has no HDD or SSD drive (on which keys could be stored and read back from by a bad actor who has access to it later).
- It has the fastest CPU and most Memory money can buy (so that you can run the key derivation with the highest difficulty).

To be practical, for this minimal guide at least, I'm going to assume that your system doesn't satisfy any of these recommendations. Instead I will assume that you will use your current computer but booted from a flash drive using the SBK live distribution. You and that it can at least satisfy the following reduced requirements:

- You have disconnected any network cable before you boot into SBK.
- You don't connect to any WiFi network and enable airplane mode as soon as possible.
- You boot from a flash drive using the SBK Linux live distribution.
- You use the flash drive *only* for SBK.
- You never connect the flash drive to any other system.
- You disconnect the flash drive from your computer before you boot back into your regular OS.

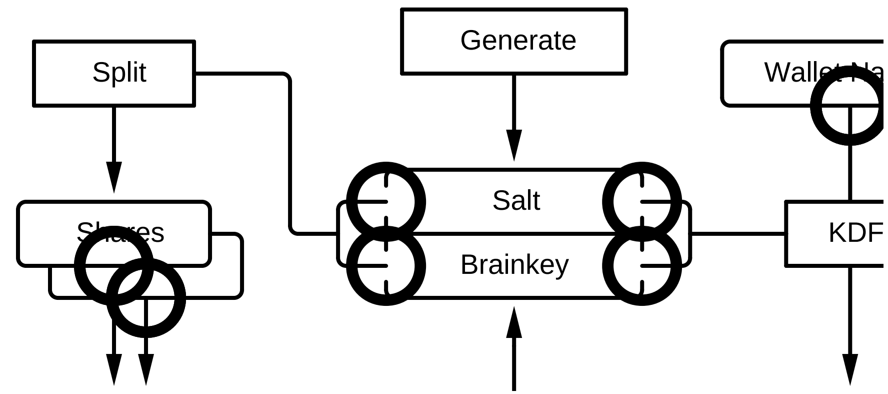
SBK does not have any persistence and should in theory run on a system that does not have any disk and is booted from a read-only medium. If you're using an SD card, you may want to switch it to read-only after you've written the SBK live image to it. The files that SBK creates will only ever be written to RAM (which is presumably volatile), so when you boot up your regular operating system again, there should be no trace of your wallet on any HDD or SSD. If we presume your regular OS to have security issues, then there should not be any files from your wallet that could possibly be leaked.

> Flash
>
> All data that is currently on your flash drive will be erased, so you should make a copy of any files you want to keep.

Implementation Overview

> Aside
>
> If you are doing code review, please be aware that some portions of the implementation, which might otherwise be deleted, are preserved for future didactic use while I rewrite SBK into a literate program. This relates in particular to the Luby Transform[30] based ECC in `sbk/ecc_lt.py` and the $GF(p)$ arithmetic in `sbk/gf.py`.

5.1 High Level Overview: Create, Join and Load



This diagram can only tell so much of course (some of the boxes might as well be labeled with "magic"). The next few sections explain in a little more detail how each step works.

4.5 Side Channel Attacks

Such attacks are mitigated substantially when you use the SBK Live distribution and do not have your computer connected to a network (either via cable or wifi) when you boot SBK Live to access your wallet. For EMI/DPA attacks to work, the attacker must have had access to your computer or be in close proximity and presumably have you as a specific target. Here again, the answer is multisig, with or without SBK.

href_keys_casa_sssss[29]

> Key splitting can function as an alternative to multisig, but after researching its practical application at Casa, we rejected implementing Shamir's Secret Sharing Scheme because it exposes clients to many more risks.

Cold Boot Attack

The previous contents of RAM can still be readable after a reboot[16]. While it is not reccomended to use SBK on a computer that you will later use for other purposes, if you do, the Tails image which SBK is uses features memory erasure[17]. For this to work propperly, you should do a clean shutdown, rather than a hard reset of your computer.

3.4 Extended Instructions

These instructions are written with the assumption that you have a high level of paranoia. You may even want to get some tinfoil out of your cupboard (though you won't be using it to make any hats).

3.4.1 Safe, Secret and Secure

The most important thing to understand, is that your wallet is generated using what is effectively a very large random number[9], known as a *wallet seed*. Anybody who has this random number also has your wallet and can take your bitcoin. If you lose this random number, your wallet is gone. With SBK you can create such a wallet seed in a way that allows you to keep it *safe*, *secret* and *secure*.

When you initially create a wallet, you will usually be instructed to write down your wallet seed on a piece of paper (for example in the form of a 12-word phrase) and to put it in a safe place. There are some disadvantages to this approach:

- Safety: The piece of paper may be destroyed (eg. in a fire) or become unreadable (eg. due to damage by water), so without a high degree of diligence on your part, such a wallet seed can be unsafe.
- Secrecy: You may not be the only one who has access to your computer or to the place you decide to keep your wallet seed. A hacker or a thief could gain access to your wallet seed and steal your bitcoin. Even a curious child, without any ill intent, might find your wallet seed and take a picture of it to ask "what is this?" on the internet, so that your wallet seed is the leaked to the public. If your lucky, an honest person will find it first, take the bitcoin before anybody else can and contact you to give them back. If you're not lucky, they don't contact you... In other words, a wallet seed can be difficult to keep secret.
- Security: The highest degree of vigilance is difficult to maintain over a long period of time. Even if you have kept your wallet seed safe and secret until now, that does not mean it will be safe and secret in the future. A wallet seed represents a single point of failure, which means you have to constantly think about its security.

This last point is perhaps the greatest benefit of SBK: You can worry much less. Yes, vigilance is still required, but not so much that any one mistake is a catastrophe and mostly on specific occasions which you can prepare for:

- When you create your wallet.
- When you access your wallet.
- When shares are distributed to trustees.
- When shares are collected from trustees.
- When your wallet is recovered from shares.

We are making an effort to accomidate validation of the implementation and audit of artifacts. As of this writing, these concerns are valid, at some point however, such concerns should be regarded as FUD by vendors who perhaps have a conflict of interest to disuade you from using pure Open Source non-custodial solutions from which they don't earn any money.

4.3 Social Recovery Complexities

Much of the criticisms of SSS key recovery revolve around bad actors who can forge shares and gain access to the other shares during a colaborative recovery process. If you can declare can declare a single person as the sole custodial of the inheritence, to whom all shares are given, then these criticisms do not apply. The custodial can determine for themselves which shares are invalid, as an invalid share will not produce a valid wallet.

If you use SBK with a multi-sig setup, and instruct mutliple custodials to do separate wallet recoveries, then these criticisms do not apply.

All risks regarding relative trustworthiness and holdouts are equally applicable to multisig setups, where some parties might refuse to sign transactions.

4.4 Inability to Verify Share Integrity

Each SBK share has error correction data based on a Reed-Solomon Code. This serves the dual purpose to protect against corruption and bad handwriting as well as to verify the correctness of the share.

To verify authenticity of a share, without revealing the share itself would be an improvement over what is currently implemented. It would make the life of a custodial easier and expand the use-cases for SBK if each share could be verified without revealing the share itself.

Tradeoffs

The use-case for SBK is the sovereign individual. Nothing epitimizes this more than a brainkey. SBK is first and foremost about direct and individual control of bitcoin, and secondarilly about Shamir's Secret Sharing, which is only for backup purposes, not a means to distribute keys. If your use-case matches this, then many criticisms of SSS are not applicable. This chapter will concern itself nonetheless with these criticisms.

In broad terms, SBK is a step up from a wallet seed, it is a pure-software alternative/complement to a hardware wallet but it does not offer all the benefits of a multisig setup. By all means use a multisig setup, in which SBK may play a role and thereby reduce your risk from depending on any individual vendor, software stack or hardware system.

4.1 Single Point of Failure

If you are concerned about a compromised device, despite all precautions to validate your SBK Live download and boot on an air-gapped system, then by all means, use a multisig setup to mitigate this risk. Sign substantial transactions on separate computers in separate locations using sofware from separate vendors that was separately downloaded and validated.

4.2 Software Implementation Bugs

Previous implementations of SSS in the Bitcoin wallets have suffered from broken implementations. This criticism can be leveled against any hardware and software wallet and bugs can be fixed. If complex implementations are an issue, then this criticism is much more applicape to hardware wallets.

Adding redundancy and making sure there is no single point of failure means that you have a much lower risk to lose your wallet due to a mistake, an accident or a disaster. In other words, SBK is designed with the assumption that you are human.

3.5 Weighing Risks

The greatest risk to your funds is human error (rather than for example a software bug), but it's worth breaking down what these errors typically look like:

- Bad IT Security: For convenience you may prefer to use your regular Windows based, network connected computer, or your regular smartphone not realizing that it has a back-door or may eventually have a back-door when it is infected with a virus. An attacker can then read the wallet files from your computer or use a keyboard logger to eavesdrop your wallet seed as you type it.
- Lack of Knowledge: You may have a poor understanding of how to use your wallet. You might for example not know the difference between the PIN to your wallet and your wallet seed{18}. Without appreciating this difference, you may never write down your wallet seed and lose your bitcoin when you switch to a new device or directly after you close the wallet software.
- Misplaced Trust: If you don't trust your technical abilities, you may prefer to trust others to do this for you. The trouble is that the people you trust may turn out to either be scammers or grossly negligent{19}.

To address these issues, SBK includes:

- A step by step guide on how to set up a secure air-gapped system.
- A step by step guide on how to use your wallet in a safe way.
- A design that does not require trust in any individual or organization[10].

The software required to load your wallet may no longer be available. SBK is hosted both on gitlab.com/mbarkhau/sbk and also on github.com/mbarkhau/sbk and you can download stand-alone versions of SBK that can be run from an USB-Stick.

3.5.1 Web Wallets: Leaked by Design

The most common case for a leaked coin is a web wallet, where your keys are in a certain sense leaked by design. The service provider of your wallet has control over your keys or if they don't then they might send you a software update to leak your keys. Note that this is not simply a question of whether or not you can trust in the good intentions and well aligned business interests of the service provider of a wallet, it is also a question of how competent they are to protect a massive honey pot (your wallet and those of all of their other users) from attackers (who might even be employees of the company) that what to take your keys.

SBK is not a service provider, has no access to your keys and can be audited for

3.5.2 Leaked over the Network

If the computer which you use to access your wallet is connected to the internet, then there is a chance that your keys will either be sent to an attacker or somehow be made public. This can happen for example if your computer is infected by a virus or malware. It may also happen if an unscrupulous associate of the NSA feels like exploiting one of the back-doors (which is an area of research for them[20]) but have not yet published.

3.5.3 Leaked by Bug

These keys are only ever on a system that you control and which you ideally never connect to a network (air-gap). This makes it next to impossible for your keys to be ever be leaked or stolen. You don't have to trust any third party service provider with your keys and the backup for your keys is distributed, without any single point of failure.

these secrets were not under your control and you therefore must presume them to have been copied/compromised, this step is nonetheless important in order to minimize risk.

3.8.3 Preparation

The first step in the recovery process is not to collect the shares or even to contact any of the trustees. The shares are presumed to be safe in their current locations and the recovery process introduces a risk that this will no longer be the case. To minimize this risk the first step should be to make preparations so that the recovery can be done in a deliberate and orderly manner.

3.8.4 Fund Transfer Preparation

The first question that needs to be answered, is what is to be done.

3.8.5 Collecting Shares

> Aside: If you recover your own wallet and you collect the `shares` personally, it *may* be safe to continue to use the wallet and to not generate new keys. If you are *not* the owner however, and the recovery process involves the cooperation of some trustees, then there is a higher risk that some of them will collude to gain access to more `shares` than just their own. In this case it is best to prepare a new wallet in advance and move all coins to it as soon as possible. For more information, see the Recovery Protocol

3.9 Checklist

- ☐ Print templates for N shares and 1 salt
- ☐ Install Electrum Wallet on your phone

3.8 Agent Guide

As the trusted agent of the owner, it is your responsibility to act on their behalf and in their interest, not as you see it but as they seen it or would have seen it. Part of this responsibility is to prepare yourself in advance and not react in an ad-hoc way only when a worst-case scenario is already underway. This guide is written to help you with this preparation.

3.8.1 Where are the Shares

> **Secrets: Salts and Shares**
>
> Since you and the agent should treat `salt` and `shares` in a very similar way, in this section I will refer to them both under the common term *secrets*.

Before the event that you have to act, the owner should give you instructions and over time they should keep you updated of any changes to these instructions. These you on a few things

- how to secure any secrets and keep you updated if any chances to these circumstances. This may include information as to the whereabouts of the secrets, or information about how this information can be obtained. It should also include information about how you may authenticate yourself to any trustee, so that you can both be assured of each others p

3.8.2 Secure Insecure Shares

Your first concern as the agent should be to secure any secrets that were under the control of the owner. Should the owner become incapacitated, there may be a newly added risk to such secrets. It might be the case for example, that the owner has a `salt` or some `shares` in their possession or in their home, which are now accessible to relatives or caretakers that may not be trusted by the owner. There may be keys to a safe or safety deposit box with such secrets. You as the trusted agent should secure any and all of these as soon as possible. While

> Aside: This is the main risk that SBK is subject to. The way keys are generated by SBK might be predictable in some subtle way, or the way in which Electrum it creates signatures might allow

Whenever you use any any bitcoin wallet, you are exposed to various risks:

1. You might make a mistake: You might forget a critical password, you might write down a secret phrase incorrectly, you might load your wallet on an insecure system etc.
2. You can fall prey to a scam: This can happen if you download your wallet software from an untrustworthy source, ie. from the website of a malicious programmer or scammer, rather than from the website of the original author.
3. The wallet software may have a bug: Your wallet may be generated in a way that it cannot be recovered or in a way that can be exploited by others to steal your funds. (As of this writing, such bugs may be the greatest risk when using SBK).

For most people, the greatest risk is usually the first: Important but complicated steps are either skipped or not done with diligence, so that your keys are lost or stolen. This is due to a combination of factors:

You can lose your funds through a lack of diligence when using your wallet. This can happen if you do not keep your keys secret, for example by loading your wallet on an insecure system, you may lose your keys in an accident or you may simply forget a critical password.

- Complicated and tedious
- Lack of justification
- Steps are complicated and tedious. If the extra effort is not justified, and if the consequences of skipping them are Without an understanding of Due to a lack of understanding of security practices, the consequences of which are either years in the future or appear to be , important steps are skipped . causes leads to the inability to diligently first and it is the risk that SBK is primarily designed to address. Far more funds are lost or stolen due to improper handling of keys, than are lost due to hacking or bugs. The goal of SBK is therefore to:

SBK is by no means free from tedium. It can be a considerable effort to prepare a secure computer, to manually copy dozens and dozens of words and numbers with diligence and to . The documentation of SBK is written to help you judge if this effort is justified for you.

- Minimize the risk of you losing your keys.
- Minimize the risk of your keys being exposed to vulnerable computer systems.
- Minimize the involvement of third parties who might steal your keys.
- Minimize the trust placed in any individual third party.

For more information on how to minimize the risk of downloading a malicious version of SBK, please review the section on software verification.

3.5.4 Software Verification

TODO

If you still trust printer manufacturers to create products that perform even the most rudimentary of their advertised functions, namely creating faithful physical copies, then you may find it enlightening to review some of the work{24} of David Kriesel's{25}. If printer manufacturers cannot even do this job right, how much confidence should we place in their ability to create devices that cannot be exploited while being connected to a network.

Suffice it to say, I recommend you do not trust your printer farther than you can throw it. SBK provides templates in A4 format{26} and US-Letter format{27} for you to print, but these do not contain any secret information and are only to make it easier for you to create shares. You will have to manually write down all of the data for your `salt` and `shares`.

3.7.3 Decoy Wallets

One of the legitimate uses for a `--wallet-name` is to enable plausible deniability{28} against an attacker who is in a position to extort you. If the attacker has access to the `salt` and if they can coerce you to reveal your `brainkey`, then they may be satisfied when they load a wallet and find some bitcoin. If you have set this up in advance, the wallet they load may in fact only be a decoy. Your main wallet would use a custom `--wallet-name`, which they do not know about and which you can plausibly deny the existence of.

While we're on the topic of plausible deniability; another approach you can take is to simply discard your `salt` and `brainkey` and to rely only the backup `shares`. If an attacker knows for certain that you have an SBK wallet, but they cannot find the `salt`, then you can plausibly claim, that you have thrown it away and that you never intended to access the wallet for years to come, so the backup `shares` were perfectly adequate. This is a plausible scenario if for example you were leaving the wallet as an inheritance, with your will containing the people and locations where shares can be found. The attacker would then have to extend their attack to recovering the shares, possibly involving more people, more time and more risk for them. The downside of actually throwing away your `salt` and `brainkey` is that you may now require the cooperation of the trustees and you may be faced with a holdout.

There are a two main considerations when you choose where/with whom to leave your backup shares.

- You want to ensure that -

Presumably you will only give a share to a person whom you can trust, so the following two issues of collusion and extortion should hopefully not be an issue. To preempt such issues however, you should make the following clear to any trustee:

- You have plenty of other backup shares to resort to. If they do not return their share to you upon request, there are others you can access and there is no point in them attempting to extort you.
- It is pointless for them to collude and gather shares. You have not given enough shares to people with whom they could collude, so any such attempt would be fruitless.
- If they do attempt to collude, it is enough for even just one trustee to warn you (the owner), so that you can create a new wallet and move all your funds away from the wallet they are trying to steal from.

You should not give your friends and family enough shares so that they could collude to steal from you. You should make it clear to them that such collusion will be fruitless so that they are not even tempted.

Ideally you will have access to enough backup shares so that you. This

3.7.1 Setting Up an Air-Gapped System

using an iso-image[23]

3.7.2 Creating Shares

It may appear strange that the supposed money of the future requires you to write dozens and dozens of words onto paper. Don't we have printers to save us from this kind of tedious and error prone work?

3.5.5 Security vs Usability

SBK is not the most convenient way to create a Bitcoin wallet. If you follow the recommended approach, during setup you will have to:

- Prepare an air-gapped system with SBK installed
- create `shares` and transcribe them onto paper,
- transcribe your `salt` and memorize your `brainkey`,
- distribute the shares
- provide [minimal instruction][#instructions-for-trustees] to any trustees

In addition, every time you want to use your wallet, you will have to

- manually enter a 12 word `salt` every time you use it,
- remember and manually enter your 6 word `brainkey`.

The price of the extra security provided by SBK is that it is a bit more tedious to use than other approaches. The intended use-case of an SBK wallet is for an infrequently accessed wallet, sometimes referred to as "cold storage"[21]. This is suitable if you intend to use bitcoin for long-term savings. If you intend to spend some of your bitcoin more frequently, you may want to use a separate "hot wallet"[22] which has only a smaller balance that you can afford to lose. This approach minimizes the risk to your funds also minimizes the tedium of using SBK.

3.6 Trustee Guide

The most common and least complicated role is that of the *trustee*, so that is the role that I will explain first.

3.6.1 What is an SBK Share

The owner of an SBK wallet trusts that you have their best interests at heart and that you can help them to avoid losing their bitcoin. To that end, they are entrusting you with part of a backup for their wallet, which is called an *SBK Share*. Such a share is a page of paper on which the following information is written:

- Minimal usage instructions
- A QR-Code that links to the extended instructions (this page)
- The name of the trustee (presumably you)
- Contact information of the owner
- The codewords of the SBK Share

The codewords may not be visible, as the template for an SBK Share is designed to be folded and sealed, such that they cannot be read without the seal being broken.

3.6.2 Tamper Evident Seal

The share may be sealed with tape, staples, glue tamper-evident

3.6.3 What to do with the SBK Share

3.6.4 Secrecy

They receive an SBK share from the owner, which is piece of paper that has been folded and sealed.

3.6.5 How to Verify an Agent

The owner may chose their agent and trustees, so that they do not know each other, which may reduce the risk that they will collude with each other to steal from the owner. In this case the owner may also give an *Agent Verification Token* to the trustee, in addition to their SBK share. They may also to to put in their safe or some other secure location to which only they have access.

3.7 Owners Guide

Before you create a wallet, you should make some preparations. You should:

1. Consider how to distribute your backup shares so that you minimize your vulnerability to bad actors.
2. Prepare materials to create shares. Ideally a share should survive a fire and it should have a tamper-evident seal.

We will start with the considerations wrt. bad actors. There are some risks that you will have to weigh, depending on your situation.

- Risk of Extortion: A person who has a share can assume that you have at least some bitcoin. Even if they are trustworthy and would never try to threaten and extort you, they might be careless about this information. Giving somebody one of your shares can be the equivalent of painting a target on your back and somebody might knock down your door in the middle of the night.
- Holdouts: A person who has a share might get the idea that you depend on them. This means that they could refuse to return the share to you unless you compensate them somehow.

There are two ways to protect yourself from extortion:

- Only use the backup shares and make sure a share from at least one person or institution is required. If the only way for you to recover your wallet is by using the backup shares, then it is not enough for extortionist to threaten you. They must also threaten the additional person or institution, which puts them at a much greater risk of being apprehended. To maintain the plausibility of this, it is best if you do
- In your safe at home.
- In safety deposit boxes.
- In secret and inaccessible locations.
- With trusted family or friends.

```python
    assert abs(in_params.kdf_t - kwargs['kdf_t']) / kwargs['kdf_t'] < 0.125

    # round trip
    params_data = params2bytes(in_params)
    out_params = bytes2params(params_data)

    is_stable_output = params2bytes(out_params) == params_data
    assert is_stable_output, out_params

    assert isinstance(params_data, bytes)
    assert len(params_data) == 3

    assert out_params.version  == in_params.version
    assert out_params.kdf_p    == in_params.kdf_p
    assert out_params.kdf_m    == in_params.kdf_m
    assert out_params.kdf_t    == in_params.kdf_t
    assert out_params.sss_x    == in_params.sss_x
    assert out_params.sss_t    == in_params.sss_t
```

The preceding validation also makes sure, after we have gone through one round of encoding, the output is stable. That is to say, if we encode the parameters again, we get the exact same encoded data as for the original inputs (which usually involves some rounding).

Furthermore, we want to be sure, that the shorter two byte representation used by the salt can be decoded with the relevant kdf parameters.

```python
# def: validate_salt_params
# dep: impl, common.impl_hex
def validate_params(in_params: Parameters) -> None:
    assert abs(in_params.kdf_m - kwargs['kdf_m']) / kwargs['kdf_m'] < 0.125
    assert abs(in_params.kdf_t - kwargs['kdf_t']) / kwargs['kdf_t'] < 0.125

    # round trip
    params_data = params2bytes(in_params)
    assert isinstance(params_data, bytes)
    assert len(params_data) == 3

    out_params = bytes2params(params_data[:2])
    assert out_params.version  == in_params.version
    assert out_params.kdf_p    == in_params.kdf_p
    assert out_params.kdf_m    == in_params.kdf_m
    assert out_params.kdf_t    == in_params.kdf_t
```


forgetting your brainkey is probably much higher than your risk of being subject to a brute-force attack, it is more important to mitigate the former risk than the latter. If your situation is different, and you are worried about the risk of a brute-force attack, then you could choose `--brainkey-len=8` to increase the entropy of your `brainkey` and/or choose `--target-duration=600` to increase the KDF difficulty.

5.5.3 KDF Implementation

Waiting 1-2 minutes for the key derivation is somewhat inconvenient, but it would be an even worse experience if you didn't even have a progress indicator and your machine appeared to be locked up while the KDF calculation was in progress. As a concession to usability, SBK has a wrapper function called `digest`, the main purpose of which is to implement a meaningful progress bar:

```python
import argon2     # pip install argon2-cffi

def digest(
    data: bytes, p: int, m: int, t: int, digest_len: int=32
) -> bytes:
    constant_kwargs = {
        'hash_len'   : 1024,
        'memory_cost': m * 1024,
        'parallelism': p,
        'type'       : argon2.low_level.Type.ID,
        'version'    : argon2.low_level.ARGON2_VERSION,
    }
    result = data

    remaining_iters = t
    remaining_steps = min(remaining_iters, 10)
    while remaining_iters > 0:
        step_iters = max(1, round(remaining_iters / remaining_steps))
        # progress indicator
        print(f"remaining: {remaining_iters:>3} of {t}  -  next: {step_iters}")

        result = argon2.low_level.hash_secret_raw(
            secret=result, salt=result, time_cost=step_iters, **constant_kwargs
        )
        remaining_steps -= 1
```

```
        remaining_iters -= step_iters

    assert remaining_steps == 0, remaining_steps
    assert remaining_iters == 0, remaining_iters
    return result[:digest_len]
```

Invocation with `t=1` produces the same result as using argon2 directly:

```
>>> digest_data = digest(b"test1234", p=1, m=1, t=1)
remaining:   1 of 1  -  next: 1
>>> import binascii
>>> print(binascii.hexlify(digest_data))
b'f874b69ca85a76f373a203e7d55a2974c3dc50d94886383b8502aaeebaaf362d'
```

You can verify this using antelle.net/argon2-browser/[44] for example (note however that `m=1` in SBK is m=1024 in argon2).

```
Params: pass=test1234, salt=test1234, time=1, mem=1024, hashLen=32, parallelism=1, type=0
Encoded: $argon2d$v=19$m=1024,t=1,p=1$dGVzdDEyMzQ$O2GpxMquN/amTCVwe5GHPJr89BvBVnM0ylSHfzez4l8
Hash: f874b69ca85a76f373a203e7d55a2974c3dc50d94886383b8502aaeebaaf362d
```

Invocation with `t>1` will split the iterations up to a maximum of 10 steps.

```
>>> digest_data = digest(b"test1234", p=1, m=1, t=87)
remaining:  87 of 87  -  next: 9
remaining:  78 of 87  -  next: 9
remaining:  69 of 87  -  next: 9
remaining:  60 of 87  -  next: 9
remaining:  51 of 87  -  next: 8
remaining:  43 of 87  -  next: 9
remaining:  34 of 87  -  next: 8
remaining:  26 of 87  -  next: 9
remaining:  17 of 87  -  next: 8
remaining:   9 of 87  -  next: 9
>>> print(binascii.hexlify(digest_data))
b'6cf1a22113182d8c66c8972e693b1cc3bb1d931a691265bad75e935b1254fccd'
```

This implementation is an unfortunate compromise. A better implementation would require an adjustment to the Argon2 library, which would be more effort. I would greatly appreciate feedback[45] on the effect this approach has on the strength of the KDF and if there is a better approach. My assessment so far is that using `t >= 20` has a comparable cost to plain argon2, with the wrapper


```
    return encoded_data[:-1]
```

```
# def: impl_bytes2params
def bytes2params(data: bytes) -> Parameters:
    is_salt_data = len(data) == 2
    if is_salt_data:
        data = data + b"\x00"   # append dummy sss_t and sss_x

    assert len(data) == 3, len(data)
    encoded_uint, = struct.unpack("<L", data + b"\x00")

    version   = (encoded_uint >> 0x00) & 0b0000_1111
    kdf_m_enc = (encoded_uint >> 0x04) & 0b0011_1111
    kdf_t_enc = (encoded_uint >> 0x0A) & 0b0011_1111
    sss_x_enc = (encoded_uint >> 0x10) & 0b0001_1111
    sss_t_enc = (encoded_uint >> 0x15) & 0b0000_0111

    assert version == SBK_VERSION_V0, f"Invalid version: {version}"

    kdf_m = param_exp(kdf_m_enc, 1.125) * 100
    kdf_t = param_exp(kdf_t_enc, 1.125)
    if is_salt_data:
        sss_x = -1
        sss_t = 2
    else:
        sss_x = sss_x_enc + 1
        sss_t = sss_t_enc + 2

    sss_n = sss_t
    return init_parameters(kdf_m, kdf_t, sss_x, sss_t, sss_n)
```

9.5.4 Fuzz Test Encode/Decode

This test shows that parameters are decoded accurately after a round trip of encoding and decoding.

```
# def: validate_share_params
# dep: impl, common.impl_hex
def validate_params(in_params: Parameters) -> None:
    assert abs(in_params.kdf_m - kwargs['kdf_m']) / kwargs['kdf_m'] < 0.125
```

```
            kdf_p=kdf_params.kdf_p,
            kdf_m=kdf_params.kdf_m,
            kdf_t=kdf_params.kdf_t,
            sss_x=sss_x,
            sss_t=sss_t,
            sss_n=sss_n,
        )
```

9.5.3 Parameter Encoding/Decoding

We test `params2bytes` and `bytes2params` together. We make sure that the round trip doesn't lose relevant information and otherwise only do sanity checks on the encoded representation of the parameters.

```
# def: impl_params2bytes
def params2bytes(params: Parameters) -> bytes:
    kdf_m_enc = param_log(params.kdf_m / 100, 1.125)
    kdf_t_enc = param_log(params.kdf_t, 1.125)

    assert params.version & 0b0000_1111 == params.version
    assert kdf_m_enc      & 0b0011_1111 == kdf_m_enc
    assert kdf_t_enc      & 0b0011_1111 == kdf_t_enc

    if params.sss_x > 0:
        sss_x_enc = params.sss_x - 1
    else:
        sss_x_enc = 0
    sss_t_enc = params.sss_t - 2

    assert sss_x_enc      & 0b0001_1111 == sss_x_enc
    assert sss_t_enc      & 0b0000_0111 == sss_t_enc

    encoded_uint = (
        0
        | params.version << 0x00
        | kdf_m_enc      << 0x04
        | kdf_t_enc      << 0x0A
        | sss_x_enc      << 0x10
        | sss_t_enc      << 0x15
    )
    encoded_data = struct.pack("<L", encoded_uint)
    assert encoded_data[-1:] == b"\x00", encoded_data[-1:]
```


adding very minimal overhead. The worst case is for `t=10` where the overhead of the wrapper ranges from 50-60%. This is plausible if we assume that the overhead is amortized the more iterations we do within argon2. I assume that low overhead compared to plain argon2 also means that there is very little room for an attacker to optimize and therefore that this approach is safe.

Another potential shortcoming that is perhaps much worse is a loss of entropy that may happen with each step. Between each step, the result is 1024 bytes long, which is hopefully sufficient for this to not be a concern. I am open to suggestions for a better construction.

5.6 Encoding Secrets: Mnemonics and Intcodes

> Aside: The work done in this section preceded the release of Trezor Shamir Backup/SLIP0039, which has many similarities to it. The wordlists of both are composed with similar considerations for length, edit distance and phonetic distinctness.

5.6.1 Prelude on Physical Deterioration

The most diligently implemented software cannot protect your secrets from physical deterioration and destruction. There are books, scrolls and tablets that have been preserved for centuries, provided they were protected from weather, fluctuations in humidity, exposure to light, from insects and if they used materials that did not break down in chemical reactions.

If you want your shares to survive until they are needed, there are simple ways to protect them from deterioration. Here are some inexpensive suggestions, ordered by increasing level of paranoia:

- Write clear, non-cursive and readable characters.
- Use a pen with archival ink. Ideally, such ink is inert, dries quickly and does not smear.
- Use acid-free paper[46].
- Use a pouch laminator[47] to create protective seal against the elements.
- Use a number punch set to punch the share data onto a metal plate[48]. Such a plate can survive a house fire much better than paper. Even with the better heat resistance, it is best to store such plates as close to the ground as possible, where the heat from a fire is much lower.
- Use a hole punch in a to mark the share data into a metal plate. This is can be slightly harder to read but it is easier to punch a hole in metal than to stamp a pattern, so you can use metals that are harder and more resistant to high temperatures.
- Use various metal seed[49] storage products[50], which can survive hotter fires (no affiliation).

5.6.2 Mnemonic for Memory

> Stop puking bits![51]
>
> — Dan Kaminsky

From personal experience I know that it is possible to remember phone numbers, mathematical constants, poems or an old ICQ number even after multiple decades. In light of this, a `brainkey` can be a reasonable choice to generate a wallet, provided you are diligent and regularly practice (spaced repetition[52]) recall of the `brainkey`, so you build up a habit.

9.5.2 Parameter Initialization

Parameters are initialized in two separate ways:

1. During initial generation of a salt or share.
2. When decoding a salt or share.

It is **critical** that we always initialize parameters in a normalized form. We cannot use any value for `kdf_m` or `kdf_t`, as not every possible value has an encoded representation. Before the kdf parameters are used to derive a key, we must make sure the parameter we used was parsed from a valid encoded form, which was then parsed by `param_exp`.

```
# def: impl_init_kdf_params
def init_kdf_params(kdf_m: ct.MebiBytes, kdf_t: ct.Iterations) -> KDFParams:
    kdf_m_enc = param_log(kdf_m / 100, 1.125)
    kdf_t_enc = param_log(kdf_t, 1.125)

    kdf_m = param_exp(kdf_m_enc, 1.125) * 100
    kdf_t = param_exp(kdf_t_enc, 1.125)
    return KDFParams(KDF_PARALLELISM, kdf_m, kdf_t)
```

The `Parameters` tuple is the dataclass used outside this module, but it shouldn't be instantiated directly. Instead, all instances are created via the `init_parameters` functioni, which ensures normalized values are used for `kdf_m` and `kdf_t`.

```
# def: impl_init_parameters
def init_parameters(
    kdf_m   : ct.MebiBytes,
    kdf_t   : ct.Iterations,
    sss_x   : int,
    sss_t   : int = DEFAULT_SSS_T,
    sss_n   : int = -1,
) -> Parameters:
    kdf_params = init_kdf_params(kdf_m, kdf_t)
    if not MIN_THRESHOLD <= sss_t <= MAX_THRESHOLD:
        raise ValueError(f"Invalid threshold: {sss_t}")
    elif kdf_params.kdf_m % 100 != 0:
        raise ValueError(f"Invalid kdf_m: {kdf_params.kdf_m} % 100 != 0")
    else:
        return Parameters(
            version=SBK_VERSION_V0,
```

We use little endian encoding, so least significant bits are at lower indexes, which at least in my mind is easier to reason about.

9.5 Implementation

The main implementation of `src/sbk/parameters.py`.

```
# def: impl
# dep: common.imports, type*, constant*, impl_*
```

9.5.1 Debug Overrides

For development and debugging, our life will be easier if we can override these values via environment variables. We intend only default values to be used by an end user.

```
# def: constant_overrides
if 'SBK_DEBUG_RAW_SALT_LEN' in os.environ:
    DEFAULT_RAW_SALT_LEN  = int(os.environ['SBK_DEBUG_RAW_SALT_LEN'])

if 'SBK_DEBUG_BRAINKEY_LEN' in os.environ:
    DEFAULT_BRAINKEY_LEN  = int(os.environ['SBK_DEBUG_BRAINKEY_LEN'])

MIN_ENTROPY      = int(os.getenv('SBK_MIN_ENTROPY'     , "16"))
MAX_ENTROPY_WAIT = int(os.getenv('SBK_MAX_ENTROPY_WAIT', "10"))

DEFAULT_KDF_T_TARGET  = int(os.getenv('SBK_KDF_T_TARGET')  or DEFAULT_KDF_T_TARGET)
DEFAULT_KDF_M_PERCENT = int(os.getenv('SBK_KDF_M_PERCENT') or DEFAULT_KDF_M_PERCENT)

DEFAULT_SSS_T = int(os.getenv('SBK_THRESHOLD') or DEFAULT_SSS_T)
DEFAULT_SSS_N = int(os.getenv('SBK_NUM_SHARES') or DEFAULT_SSS_N)
```


SBK uses a mnemonic encoding that is designed to help with memorization of the `brainkey`. The format is designed with the following in mind:

- Human memory can remember concrete objects, people and places more easily than abstract words.
- Human memory fills in gaps (often incorrectly) so ambiguous words must be avoided.

The technical criteria for the wordlist are:

- The wordlist has 256 words.
- All words must be at least 5 characters long.
- All words must be at most 8 characters long.
- All words must have a unique 3 character prefix.
- The 3 character prefix of a word may not be a part of may other word.
- The damerau levenshtein edit distance of any two words must be at least 3.

The wordlist is composed only of commonly used concrete nouns such as animals, creatures, famous people, characters, physical objects, materials, substances and well known places/organizations. The wordlist does not contain any abstract words, adjectives, adverbs. Consider that the very first word humans ever spoke may have been have been the equivalent of "mother" or "snake", rather than words for abstract concepts such as "agency" or "ambition".

> Aside: Some words on the wordlist may be provocative/obscene, such as "dildo" and "saddam", but they are used partially for that reason: provocative words are more memorable than plain and boring words, as I'm sure many parents with potty-mouthed children can attest.

Using such words makes it easier to use the Method of Loci[53] or to construct a story as a memory aid. As an example, given the following brainkey:

```
sunlight  origami   leibniz   gotham
geisha    barbeque  ontario   vivaldi
```

You might construct a picture in your mind of a beam of *sunlight* which falls on a piece of *origami* that was folded by *Leibniz* while he was in *Gotham* city. A *geisha* looks upon it as she eats her *barbeque* in *ontario* and listens to *vivalidi*. Please consider in an hour or two if it is easier for you to recall the previous picture or these random digits: 053-404 098-139 152-596 236-529. Both these digits and the previous set of words are encodings of the same raw data: `b"\x6f\x56\x7f\x5b"`

I hope this illustrates of ability of humans to remember what has been very important to us throughout history: stories about people and places.

> Caveat: The choices for the current wordlist are probably not optimal as I have not done exhaustive tests. It may be for example, that it is easier to memorize fewer words from a larger wordlist. The price for this is that a larger wordlist leads to smaller levenshtein/edit distances between words, to longer word lengths, to less phonetic distinctiveness and the to a larger burden on non-native speakers of English (because less frequently used words must be used to fill out the wordlist).
>
> Improving the wordlist is a rabbit hole that involves trade-offs and diminishing returns, so I'm leaving it as is.


```
+-------------+---+---+---+---+---+---+---+----+----+-------+-------+-------+
| n           | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  | 61    | 62    | 63    |
| log(exp(n)) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  | 61    | 62    | 63    |
| exp(n)      | 1 | 2 | 3 | 4 | 6 | 7 | 9 | 11 | 14 | 10546 | 11866 | 13350 |
+-------------+---+---+---+---+---+---+---+----+----+-------+-------+-------+
# exit: 0
```

With different choices for b we can now trade off precision vs magnitude. With a base of 11/10 we can have a magnitude of 4000x of our lowest value, where each increment is roughly 1/10 larger than the previous.

```
# exec
# dep: impl_log_and_exp
for b in [17/16, 11/10, 9/8, 6/5, 5/4]:
    s, o = param_coeffs(b)
    maxval = round(b**63 * s + o)
    print(f"{b=:.3f} {s=:<2} {o=:<3} {maxval=}")
```

```
# out
b=1.062 s=16 o=-15 maxval=714
b=1.100 s=9  o=-8  maxval=3639
b=1.125 s=8  o=-7  maxval=13350
b=1.200 s=5  o=-4  maxval=486839
b=1.250 s=4  o=-3  maxval=5097891
# exit: 0
```

9.4 Data Packing

A short interlude on how we pack/unpack binary data. We want to encode data in three bytes and our choice of endianness should make it easy to reason about when looking at the previous data layout diagram.

```
# exec
from struct import pack, unpack

numeric = (0xAB << 0) | (0xCD << 8) | (0xEF << 16)
data = b"\xAB\xCD\xEF\x00"

assert pack("<L", numeric) == data
assert unpack("<L", data)[0] == numeric
```

```
def param_exp(n: int, b: float) -> int:
    s, o = param_coeffs(b)
    v = round(b ** n * s + o)
    return v

def param_log(v: int, b: float) -> int:
    s, o = param_coeffs(b)
    n = log((v - o) / s) / log(b)
    return min(max(round(n), 0), 2**63)
```

9.3.1 Evaluate `param_exp` and `param_log`

```
# exec
# dep: impl_log_and_exp
import terminaltables as tt

for b in [1+1/10, 1+1/8]:
    s, o = param_coeffs(b)
    print(f"{b=:.3f} {s=:.3f} {o=:.3f}")

    data = [["n"], ["log(exp(n))"], ["exp(n)"]]
    for n in [0, 1, 2, 3, 4, 5, 6, 7, 8, 61, 62, 63]:
        e = param_exp(n, b)
        l = param_log(e, b)
        data[0].append(n)
        data[1].append(l)
        data[2].append(e)
    table = tt.AsciiTable(data)
    table.inner_heading_row_border = False
    print(table.table)
```

```
# out
b=1.100 s=9.000 o=-8.000
+-------------+---+---+---+---+---+---+---+----+----+------+------+------+
| n           | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  | 61   | 62   | 63   |
| log(exp(n)) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  | 61   | 62   | 63   |
| exp(n)      | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 10 | 11 | 3006 | 3308 | 3639 |
+-------------+---+---+---+---+---+---+---+----+----+------+------+------+
b=1.125 s=8.000 o=-7.000
```


5.6.3 Integer Codes

In addition to the mnemonic encoding, SBK uses a numeric encoding, consisting of two triplets of decimal digits: `053-404`. These have some benefits compared to the mnemonic encoding:

- They encode their position in the secret to protect against transposition errors during input.
- They can be used to detect input errors as they are a redundant encoding.
- They are used to encode not only the raw data, but also ECC data.
- They can be entered with one hand on a keypad while reading off a piece of paper.
- They are better suited for use with a punch/stamping set (which may consist only of decimal digits).

The primary purpose of this encoding is to give protection against incorrectly entered `shares`. Since the recovery process requires you to enter multiple `shares` and since the key derivation can take quite some, it is important to detect such input errors early. Without such protection, you could only detect an incorrect input when you see that you have loaded the wrong (ie. an empty) wallet. To make matters worse, this would be long after the input error happened and you would have no indication as to which words were entered incorrectly.

This is how the full `brainkey` is displayed by SBK.

```
     Data          Mnemonic                ECC
01: 021-729   geisha    tsunami     04: 258-287
02: 066-639   airport   forest      05: 308-329
03: 187-708   toronto   diesel      06: 361-894
```

The "Data" and "Mnemonic" sections both encode the same raw data: `b"\x6f\x56\x7f\x5b"`. The `intcodes` under the "ECC" label encode data for forward error correction[54]. To recover your wallet, it is enough to enter either the "Mnemonic", the "Data" or at least half of any of the `intcodes` (either from the "Data" and/or "ECC" sections). If enough has been entered, SBK will fill in the missing values and you can compare what has been filled in with your physical copy. If what has been filled in does not exactly match your copy, then you have made an input error somewhere.

The data portion of each `intcode` can be obtained by parsing it as a decimal integer and masking with `& 0xFFFF`.

```
intcode = int("187-708".replace("-", ""))
assert intcode == 187708
assert intcode == 0x2DD3C
assert intcode & 0xFFFF == 0xDD3C
```

The position/index of each code can be obtained by bit shifting with `>> 16`.

```
assert  21_729 >> 16 == 0
assert  66_639 >> 16 == 1
assert 187_708 >> 16 == 2
assert 361_894 >> 16 == 5
```

You may observe that a larger position/index would require more than 6 digits to represent. To ensure the decimal representation never uses more than 6 digits, the position index is limited using `% 13`:

```
assert 25 << 16 | 0xffff == 1703_935
assert (25 % 13) << 16 | 0xffff == 851_967
```

5.6.4 FEC: Forward Error Correction

As a `share` may be needed only years after it was created, there is a risk that it may become partially unreadable due to physical deterioration. An FEC code is used to have a better chance to recover such a `share`, so long as it is still partially intact.

SBK uses a Reed Solomon[55] Error Correction Code, implemented in `sbk/ecc_rs.py`. There is a minimal cli program which can be used to test it in isolation.

```
$ echo -n "WXYZ" | python -m sbk.ecc_rs --encode
5758595afbdc95be
$ echo "5758595afbdc95be" | python -m sbk.ecc_rs --decode
```


$$
\begin{aligned}
g(0) &= g(1) - 1 \\
g(0) &= sb^0 \\
g(0) &= s \\
g(1) &= sb \\
g(0) + 1 &= g(1) \\
s + 1 &= sb \\
1 &= sb - s \\
1 &= s(b-1) \\
s &= 1/(b-1)
\end{aligned}
$$

```
# def: param_coeffs
# dep: common.typing
def param_coeffs(b: float) -> Tuple[int, int]:
    assert b > 1
    s = int(1 / (b - 1))
    o = int(1 - s)

    v0 = b ** 0 * s + o
    v1 = b ** 1 * s + o
    assert v0 == 1
    assert 1.5 < v1 < 2.5
    return (s, o)
```

9.3 param_exp and param_log

In the context of the `kdf` module, for a given base, we will use `param_exp` to convert `n -> v` and `param_log` to convert `v -> n`, where `v` is the value for a parameter `-m` or `-t`.

$$
\begin{aligned}
\mathit{param\_exp}(n, b) &= \lfloor o + s \times b^n \rceil \\
\mathit{param\_log}(v, b) &= \lfloor \log_b(\frac{v - o}{s}) \rceil
\end{aligned}
$$

```
# def: impl_log_and_exp
# dep: param_coeffs
from math import log

```

9.2 Parameter Range and Encoding

For the remaining parameters `-m` and `-t`, we do want to encode them in the salt, as memory availability is widely variable and the number of iterations is the most straight forward way for users to trade off protection vs how long they are willing to wait when they access their wallet.

For `-m` we don't want to support low end hardware, as we expect to run on PC hardware starting with the x64 generation of multi-core CPUs. We would like to use a substantial portion of the available memory of systems starting from 1GB.

We chose an encoding where we cover a range that is large in magnitude rather than precision, which means that key derivation will use a lower value for `-m` than might exhaust a systems memory and a higher value for `-t` than would correspond exactly to how long the user chose as their preference to wait.

The general principle of encoding is to chose a base `b` for each parameter such that integer `n` encoded in 6bits covers our desired range for each parameter. We have `n` during decoding and our function `d(n: int) -> float`:

$$d(n) = p \ \mid p > 1, p \in \mathbb{R}$$

Which should satisfy

$$\begin{aligned} d(0) &= 1 \\ d(1) &> 1 \\ d(n) &\approx b^n \\ \lceil d(n) \rceil &\neq \lceil d(n+1) \rceil \end{aligned}$$

To satisfy (4) we can scale b^n by a factor s and then pull the curve down with an offset o so we satisfy (1). We first derive s from our constraints and then we have $o = 1 - s$.


```
WXYZ
$ python -c "print('\x57\x58\x59\x5a')"
WXYZ
```

Term	Value	Description
`message`	`WXYZ/5758595a`	ASCII and hex representation of the input message
`ecc_data`	`fbdc95be`	Redundant Error Correction Data, derived from the message.
`block`	`5758595afbdc95be`	Hex representation of `message` || `block`

As you can see, the `ecc_data` is a suffix added to the original message. My understanding is that this is called a systematic form encoding[56]. This RS implementation used by SBK uses a variable length polynomial with coefficients derived from the input message. In our example, using the message `5758595a`, the polynomial is defined using four data points and four additional error correction points:

```
        Data                  ECC
Point(x=0, y=0x57)   Point(x=4, y=0xfb)
Point(x=1, y=0x58)   Point(x=5, y=0xdc)
Point(x=2, y=0x59)   Point(x=6, y=0x95)
Point(x=3, y=0x5a)   Point(x=7, y=0xbe)
```

Each byte of the input message is interpreted as the y-coordinate of a point which lies on the polynomial, with the x-coordinate being the position in the block. Arithmetic is done using $\mathrm{GF}(256)$, just as for the Shamir's secret sharing, which allows for much of the implementation of `sbk/gf.py` and `sbk/gf_poly.py` to be reused.

With this approach, we can recover the original message even if only half of the block is available:

```
$ echo "5758595a        " | python -m sbk.ecc_rs --decode
WXYZ
$ echo "        fbdc95be" | python -m sbk.ecc_rs --decode
```

```
WXYZ
$ echo "5758        95be" | python -m sbk.ecc_rs --decode
WXYZ
```

Note that the missing/erased portions of the message are explicitly marked with whitespace. An erasure is easier to recover from than corruption. If a byte of data is incorrect rather than missing, at least one further correct byte is needed in order to recover the original message. Corruption is corrected in a process of trial and error, in which the most probable polynomial for the given set of points is determined.

```
$ echo "00000000fbdc95be" | python -m sbk.ecc_rs --decode
Traceback (most recent call last):
...
__main__.ECCDecodeError: Message too corrupt to recover.
$ echo "57000000fbdc95be" | python -m sbk.ecc_rs --decode
WXYZ
```



```
DEFAULT_KDF_T_TARGET  = ct.Seconds(90)
DEFAULT_KDF_M_PERCENT = 100

DEFAULT_SSS_T = 3
DEFAULT_SSS_N = 5
```

A note in particular on the lengths for salt and brainkey. The length of a share consists of the salt + brainkey + header. This gives us a total of 24 bytes/words. The values were chosen with the following priority of constraints:

1. Due to encoding constraints, the header length for a share is fixed at 3 bytes.
2. The main constraint on the brainkey is a minimum entropy to protect against a compromised salt. This must be balanced against the maximum number of words a human can be expected to memorize.
3. With the previous two constraints on the header and brainkey, any remaining constraints must be satisfied by the salt. The main constraint here is a minimum level of total entropy.

With an entropy of 8 words/bytes = 64 bits, the brainkey is expensive but perhaps not infeasible to brute force. This low value is only justified as the attack to defend against is the narrow case of a compromised salt. The wallet owner is intended ot be the only person with access to the salt (treating it similarly to a traditional wallet seed) and should be aware if it may have been compromised, giving them enough time to create a new wallet.

The resulting total entropy is at least `13 + 8 = 15byte = 168bit`. The headers are not completely random, so they are not counted as part of the entropy.

```
# def: constants_lens
SALT_HEADER_LEN  = 2
SHARE_HEADER_LEN = 3

DEFAULT_RAW_SALT_LEN  = 13
DEFAULT_BRAINKEY_LEN  = 8

# DEFAULT_RAW_SALT_LEN  = 5
# DEFAULT_BRAINKEY_LEN  = 4
```

```

    sss_x: int
    sss_t: int
    sss_n: int
```

The `sss_*` parameters may not always be available: - `sss_x` may be `-1` when params were decoded from a salt - `sss_n` are usually `-1` except when generating shares.

In addition we use a subset of the parameters for any context that is strictly concerned with key derivation and has nothing to do with shamir shares.

```
# def: type_kdf_params
class KDFParams(NamedTuple):
    kdf_p: ct.Parallelism
    kdf_m: ct.MebiBytes
    kdf_t: ct.Iterations
```

9.1.3 Module: `sbk.parameters`

```
# file: src/sbk/parameters.py
# include: common.boilerplate
# dep: common.imports, constant*, type*, impl*
```

```
# run: bash scripts/lint.sh src/sbk/parameters.py
# exit: 0
```

9.1.4 Constants

```
# def: constants
SBK_VERSION_V0 = 0

# constrained by f_threshold (3bits)
MIN_THRESHOLD = 2
MAX_THRESHOLD = 10

KDF_PARALLELISM       = ct.Parallelism(128)   # hardcoded
```


Utilities and Boilerplate

```
# def: boilerplate
#################################################
#    This is a generated file, do not edit    #
#################################################

# This file is part of the sbk project
# https://github.com/mbarkhau/sbk
#
# Copyright (c) 2019-2021 Manuel Barkhau (mbarkhau@gmail.com) - MIT License
# SPDX-License-Identifier: MIT
```

6.1 Build Helper Scripts

```
# file: scripts/lint.sh
#!/bin/bash
set -e;
black --quiet $@;
isort --quiet $@;
flake8 --ignore D,F,E203,E402,W503 $@;
# pylint --errors-only $@;
```

6.2 Imports

We provide a whole set of imports that are commonly used. For any individual module this may be excessive, but it greatly reduces boilerplate throughout the program.

```
# def: typing
import typing as typ
from typing import NewType, Callable, NamedTuple, Optional
from typing import Tuple, List, Dict, Set, Any
from typing import Generator, Iterator, Iterable, Sequence
from typing import Type, TypeVar, Generic, Union, Protocol
# from collections.abc import Generator, Iterator, Counter

# from typing import TypeAlias
TypeAlias = Any
```

```
# def: imports
import os
import re
import sys
import math
import time
import json
import base64
import struct
import logging
import hashlib
import threading
import pathlib as pl
import functools as ft
import itertools as it
import subprocess as sp

# dep: typing
import sbk.common_types as ct

logger = logging.getLogger(__name__)
```

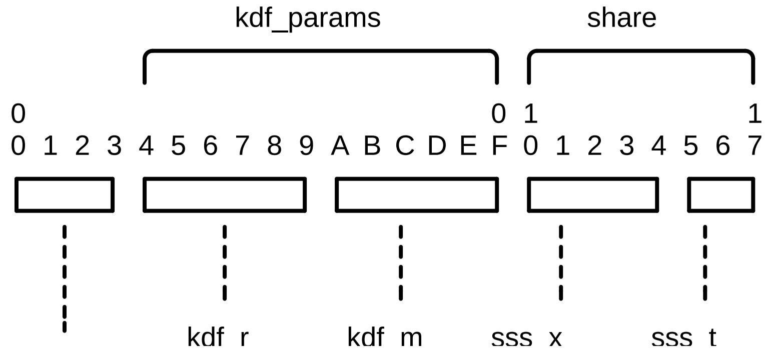
```
# def: debug_logging
_logfmt = "%(asctime)s.%(msecs)03d %(levelname)-7s " + "%(name)-16s - %(message)s"
logging.basicConfig(level=logging.DEBUG, format=_logfmt, datefmt="%Y-%m-%dT%H:%M:%S")
```


Parameters Encoding/Decoding

9.1 Overview

9.1.1 Data layout

Each Salt and Share is prefixed with the encoded parameters. These parameters are used to derive the wallet seed. For the salt, the header is 2 bytes, for each share it is 3 bytes.



9.1.2 Dataclass: `Parameters`

Throughout SBK, the decoded the parameters are passed with an instance of `Parameters`.

```
# def: type_parameters
class Parameters(NamedTuple):

    version   : int

    kdf_p: ct.Parallelism
    kdf_m: ct.MebiBytes
    kdf_t: ct.Iterations
```

```
↪ /bLPKtfccNQyBZN/64rM
14.326 seconds
# exit: 0
```

```
# run: bash -c "lscpu | grep -i core"
Thread(s) per core:              2
Core(s) per socket:              4
Model name:                      Intel(R) Core(TM) i7-8705G CPU @ 3.10GHz
# exit: 0
```

With this mid-range processer from 2018, using `-m=100MB` we can extrapolate that 130k iterations would take on the order of 30 minutes. This should suffice to make use of future hardware, given that much higher values will typically be used for `-m`.

8.4 Further reading

- Practical Cryptography for Developers - Argon2[68]
- ory.sh - Choose Argon2 Parameters[69]
- twelve21.io - Parameters for Argon2[70]

6.3 Module sbk.common_types

```python
# file: src/sbk/common_types.py
# dep: common.boilerplate
"""Types used across multiple modules."""

from typing import NewType, Sequence, Callable, Optional, NamedTuple
from typing import Tuple, List, Dict, Set, Any
# from typing import TypeAlias
TypeAlias = Any
# dep: types
```

```python
# def: types
RawSalt: TypeAlias = bytes

# ParamConfig data + RawSalt
Salt     : TypeAlias = bytes
BrainKey : TypeAlias = bytes
MasterKey: TypeAlias = bytes

class RawShare(NamedTuple):
    x_coord: int
    data   : bytes  # only the encoded GFPoint.y values

# ParamConfig data + RawShare.data
Share: TypeAlias  = bytes
Shares: TypeAlias = Sequence[Share]

SeedData: TypeAlias = bytes

ElectrumSeed: TypeAlias = str

LangCode: TypeAlias = str

# include: kdf_types
```

6.4 Constants for Configuration

```
# def: constants
DEFAULT_XDG_CONFIG_HOME = str(pl.Path("~").expanduser() / ".config")
XDG_CONFIG_HOME = pl.Path(os.environ.get('XDG_CONFIG_HOME', DEFAULT_XDG_CONFIG_HOME))

SBK_APP_DIR_STR = os.getenv('SBK_APP_DIR')
SBK_APP_DIR     = pl.Path(SBK_APP_DIR_STR) if SBK_APP_DIR_STR else XDG_CONFIG_HOME / "sbk"
```

6.5 KDF Types

Types for progress bar. This provides the common API for Qt and CLI based progress bar rendering, as we for the same kdf calculation code.

```
# def: kdf_types
ProgressIncrement    : TypeAlias = float
ProgressCallback     : TypeAlias = Callable[[ProgressIncrement], None]
MaybeProgressCallback: TypeAlias = Optional[ProgressCallback]

Parallelism : TypeAlias = int
MebiBytes   : TypeAlias = int
Iterations  : TypeAlias = int
Seconds     : TypeAlias = float
```

6.6 Hex Encode/Decode

```
# file: src/sbk/utils.py
# include: common.boilerplate
# dep: common.imports, impl_*
```

```
# def: impl_hex
def hex2bytes(hex_str: str) -> bytes:
    """Convert bytes to a hex string."""
    hex_str = hex_str.upper().zfill(2 * ((len(hex_str) + 1) // 2))
```


There does appear to be an overhead to the use of large values for `-p`, which an adversary may not have. If we consider any time on hash computation as a given, we should prefer to spend it on further iterations rather than on concurrency overhead, that can perhaps be mitigated by differen hardware choices.

At least for `-p=128` however, the overhead is quite low. Since users of SBK are unlikely to use hardware which will be underutilized with such a large value, it should be a fair trade-off to hard-code `-p=128` for `version=0`.

Feedback Welcome

If you know of hardware for which (or any other reason why) this value of `-p` is inappropriate, please open an issue on GitHub.

8.3 Memory and Time Parameters

Memory Swapping

On systems with swap the, behaviour of argon2 appears to be that it will exit with status: 137. At least on the systems we have tested it does not appear to use swap. Regardless, SBK Live does not create a swap partition.

For `version=0`, if we would like to protect against brute force As an arbitary choice for the lowest value for `-m`, a lower bound of 100 Mebibyte and `1.125` as a base. Systems which support such a small value have been readily available for over a decade, so this choice is already quite low. $100\ \mathrm{MB} \times 8 \times 1.125^{63} \approx 1300\ \mathrm{GB}$

For the parameter `-t` (number of iterations) we have a lower bound simply of 1 and use `1.125` as a base, which gives us an upper bound of $5 \times 1.125^{63} \approx 134\mathrm{k}$ iterations.

```
# run: python3 scripts/argon2cffi_test.py -t 1000 -m 16.6 -p 8 -l 24 -y 2
# timeout: 100
Encoded:    $argon2id$v=19$m=99334,t=1000,p=8$c29tZXNhbHQ$yXZeXaquxQcv
```

It appears that the the argon2cffi implementation does use multiple cores, where the cli implementation does not.

8.2 Cost of Threading

If we can establish that `-p=1024` parallel lanes contributes insignificant overhead compared to just `-p=1` (given large enough value for `-m`), then perhaps we won't have to encode the parameter `p` in the salt.

```
# run: python3 scripts/argon2cffi_test.py -t 3 -m 20 -p 8 -l 24 -y 2
# timeout: 100
Encoded:    $argon2id$v=19$m=1048576,t=3,p=8$c29tZXNhbHQ$rPe+PH3lwPgbjSq65GVqTLxDkmSCtetd
0.567 seconds   0.564 seconds   0.564 seconds   0.564 seconds   0.565 seconds
# exit: 0
```

```
# run: python3 scripts/argon2cffi_test.py -t 3 -m 20 -p 128 -l 24 -y 2
# timeout: 100
Encoded:    $argon2id$v=19$m=1048576,t=3,p=128$c29tZXNhbHQ$b9PXPtsjVyrVQQLCK5+ZpQ0qzoAVX763
0.679 seconds   0.659 seconds   0.661 seconds   0.666 seconds   0.650 seconds
# exit: 0
```

```
# run: python3 scripts/argon2cffi_test.py -t 3 -m 20 -p 1024 -l 24 -y 2
# timeout: 100
Encoded:    $argon2id$v=19$m=1048576,t=3,p=1024$c29tZXNhbHQ$w1lfUA36hCMZgJ37QjHmkm5FTx4giq7G
0.853 seconds   0.869 seconds   0.858 seconds   0.857 seconds   0.854 seconds
# exit: 0
```

Regarding the choice of `-p`, the Argon2 Spec (2015)[67] says:

> Argon2 may use up to 2^{24} threads in parallel, although in our experiments 8 threads already exhaust the available bandwidth and computing power of the machine.


```
    return base64.b16decode(hex_str.encode('ascii'))

def bytes2hex(data: bytes) -> str:
    """Convert bytes to a hex string."""
    return base64.b16encode(data).decode('ascii').lower()

def bytes_hex(data: bytes) -> str:
    """Display bytes data in hex form, rather than ascii."""
    chars           = (data[i : i + 1] for i in range(len(data)))
    char_hex        = [bytes2hex(c).lower() for c in chars]
    char_hex_padded = (c + " " if (i + 1) % 2 == 0 else c for i, c in enumerate(char_hex))
    return "".join(char_hex_padded).strip()
```

6.7 Integer Encoding/Decoding

```
# def: impl_int
def bytes2int(data: bytes) -> int:
    r"""Convert bytes to (arbitrary sized) integers.

    Parsed in big-endian order.
    """
    # NOTE: ord(data[i : i + 1]) is done for backward compatability
    #   with python2. This is because data[i] and iteration over bytes
    #   has different semantics depending on the version of python.
    num = 0
    for i in range(len(data)):
        num = num << 8
        num = num | ord(data[i : i + 1])
    return num


def int2bytes(num: int, zfill_bytes: int = 1) -> bytes:
    """Convert (arbitrary sized) int to bytes.

    Serialized in big-endian order.
    Only integers >= 0 are allowed.
    """
    assert num >= 0

```

```
    parts = []
    while num:
        parts.append(struct.pack("<B", num & 0xFF))
        num = num >> 8

    while len(parts) < zfill_bytes:
        parts.append(b"\x00")

    return b"".join(reversed(parts))
```

6.8 Progressbar

```
# def: impl_progressbar
class ProgressSmoother:

    increments: List[float]

    def __init__(self, progress_cb: ct.ProgressCallback) -> None:
        self.increments = [0]

        def fake_progress() -> None:
            step_duration = 0.1
            tzero         = time.time()
            while True:
                time.sleep(step_duration)
                if self.total_incr() == 0:
                    progress_cb(0.01)
                elif self.total_incr() >= 100:
                    progress_cb(100)
                    return
                else:
                    duration      = time.time() - tzero
                    steps         = duration / step_duration
                    incr_per_step = self.total_incr() / steps
                    progress_cb(incr_per_step)

        self._thread = threading.Thread(target=fake_progress)
```


It appears that the python argon2-cffi[66] implementation is significantly faster, which is perhaps mostly due to cli invokation overhead or due to multi-threadding. As we care about performance on the order of at least a few seconds, we measure a more expensive call and also limit parallelism to 1, to make sure that both implementations only use one core.

```
# run: bash scripts/argon2cli_test.sh -t 3 -m 17 -p 1 -l 24 -id
# timeout: 100
Encoded:    $argon2id$v=19$m=131072,t=3,p=1$c29tZXNhbHQ$mKtFTe5acsEv
↪ /wtRdOwuOxxX2QmF8+hu
0.329 seconds
0.312 seconds
0.313 seconds
# exit: 0
```

```
# run: python3 scripts/argon2cffi_test.py -t 3 -m 17 -p 1 -l 24 -y 2
# timeout: 100
Encoded:    $argon2id$v=19$m=131072,t=3,p=1$c29tZXNhbHQ$mKtFTe5acsEv
↪ /wtRdOwuOxxX2QmF8+hu
0.283 seconds   0.287 seconds   0.277 seconds   0.276 seconds   0.279 seconds
# exit: 0
```

With these settings the implementations seem comparable, let's try with a higher degree of parallelism.

```
# run: bash scripts/argon2cli_test.sh -t 3  -m 17 -p 8 -l 24 -id
# timeout: 100
Encoded:    $argon2id$v=19$m=131072,t=3,p=8$c29tZXNhbHQ$0
↪ g5ayzO4asYRYEIckSx6gB21upJ11Gih
0.318 seconds
0.319 seconds
0.315 seconds
# exit: 0
```

```
# run: python3 scripts/argon2cffi_test.py -t 3 -m 17 -p 8 -l 24 -y 2
# timeout: 100
Encoded:    $argon2id$v=19$m=131072,t=3,p=8$c29tZXNhbHQ$0
↪ g5ayzO4asYRYEIckSx6gB21upJ11Gih
0.082 seconds   0.072 seconds   0.071 seconds   0.071 seconds   0.074 seconds
# exit: 0
```

```
        b"password",
        b"somesalt",
        time_cost=t,
        memory_cost=int(2**m),
        parallelism=p,
        hash_len=l,
        type=argon2.Type(y),
    )
    duration = time.time() - tzero
    return (hash_encoded.decode("ascii"), duration)


def measure_argon2(*args, **kwargs) -> None:
    hash_encoded, duration = _measure_argon2(*args, **kwargs)
    print(f"Encoded:\t{hash_encoded}")
    print(f"{duration:.3f} seconds", end="   ")
    for _ in range(4):
        if duration > 3:
            return
        _, duration = _measure_argon2(*args, **kwargs)
        print(f"{duration:.3f} seconds", end="   ")


def main(args: List[str]) -> None:
    _t, t, _m, m, _p, p, _l, l, _y, y = args
    assert [_t, _m, _p, _l, _y] == ['-t', '-m', '-p', '-l', '-y']
    measure_argon2(int(t), float(m), int(p), int(l), int(y))


if __name__ == '__main__':
    main(sys.argv[1:])
```

```
# run: python3 scripts/argon2cffi_test.py -t 2 -m 16 -p 4 -l 24 -y 1
Encoded:    $argon2i$v=19$m=65536,t=2,p=4$c29tZXNhbHQ$RdescudvJCsgt3ub+b+dWRWJTmaaJObG
0.045 seconds   0.034 seconds   0.034 seconds   0.036 seconds   0.034 seconds
# exit: 0
```



```
        self._thread.start()

    def total_incr(self) -> float:
        return sum(self.increments) + max(self.increments) * 0.55

    def progress_cb(self, incr: ct.ProgressIncrement) -> None:
        self.increments.append(incr)

    def join(self) -> None:
        self._thread.join()
```

Primes where p < 2^n for GF(p)

As mentioned in 030_user_guide, the Galois Field we use can either be of the form $\mathrm{GF}(\mathrm{p})$ (where p is a prime number) or $\mathrm{GF}(\mathrm{p}^{\mathrm{n}})$ (and a reducing polynomial). This chapter concerns the prime numbers needed for $\mathrm{GF}(\mathrm{p})$.

While we don't use $\mathrm{GF}(\mathrm{p})$ in practice, the arithmetic in $\mathrm{GF}(\mathrm{p})$ is less complicated, so SBK includes a GF implementation for use with a prime number. In other words, this chapter is to help validate the GF logic on more simple case, it is not a functional part of the implementation of SBK.

7.1 API of `sbk.primes`

The API of this module has two functions.

```
def get_pow2prime(num_bits: int) -> int:
    ...
```

`get_pow2prime` returns the largest prime number for which $2^{\mathrm{n}} - \mathrm{k} \leq 2^{\mathrm{num\_bits}}$.

When we create a $\mathrm{GF}(\mathrm{p})$, we want to pick a prime that is appropriate for the amount of data we want to encode. If we want to encode a secret which has 128 bits, then we should pick a prime that is very close to 2^{128} . If we picked a larger prime, then the points we generate would be larger than needed, which would mean a longer mnemonic to write down, without any additional security (i.e. for no good reason). If we picked a smaller prime, then security would be compromised.

If we don't want to deal with such large primes, we need to chunk the secret and encode points separately. This is what we do in practice anyway [[user_guide#What is an SBK Share]], where each byte of a share represents a point in $\mathrm{GF}(2^8)$, but again, that is an extra complication. The use of larger primes allows us to validate with a simplified implementation.

8.1 Baseline Hashing Performance

As a baseline, we want to make sure that we are not measuring only a particular implementation of argon2. We especially want to be sure that the implementations we use are not slower than what an attacker would have access to.

```
# run: bash -c 'apt-cache show argon2 | grep -E "(Package|Architecture|Version)"'
Package: argon2
Architecture: amd64
Version: 0~20171227-0.2
# exit: 0
```

```
# file: scripts/argon2cli_test.sh
echo -n "password" | argon2 somesalt $@ | grep -E "(Encoded|seconds)"
for ((i=0;i<2;i++)); do
    echo -n "password" | argon2 somesalt $@ | grep seconds
done
```

```
# run: bash scripts/argon2cli_test.sh -t 2 -m 16 -p 4 -l 24
Encoded:    $argon2i$v=19$m=65536,t=2,p=4$c29tZXNhbHQ$RdescudvJCsgt3ub+b+dWRWJTmaaJObG
0.119 seconds
0.110 seconds
0.118 seconds
# exit: 0
```

This can be compared to the output of the reference implementation gh/argon2[65].

```
# file: scripts/argon2cffi_test.py
import sys
import time
import argon2
from typing import Tuple, List


def _measure_argon2(
    t: int, m: float, p: int, l: int = 24, y: int = 2
) -> Tuple[str, float]:
    tzero = time.time()
    hash_encoded = argon2.low_level.hash_secret(
```

KDF Parameter Investigation

As the KDF parameters are encoded in the salt (and shares), we want to have an encoding that is compact. This means, where possible, we should make parameters either static or implicit. Where not, the primary purpose of variable parameters is to support future hardware configurations, so that brute-force attacks continue to be infeasible.

The first two bytes of the salt are for parameter encoding, of which the first 3bits are for a version number. There is an upgrade path open if we want to use a more different approch to parameter encoding.

The parameters we're looking at are these:

- `p` : parallelism (number of lanes/threads)
- `m` : memory
- `t` : iterations
- `y` : hashType (0:i, 1:d, 2:id)

From the IETF draft on Argon2[64], we adopt `y=2` (Argon2id) without any further investigation, as it is declared the primary variant.

> **Side Channel Attacks**
>
> Considering that SBK is intended for offline use on a single system, rather than as part of an interactive client/server setup, the choice of `Argon2id` may not be optimal. The choice of `Argon2d` might be marginally better, as it would make brute force attacks more difficult, which are of greater concern. More investigation is welcome, even if only to quantify how marginal the benefit of an alternate choice is.


```
def is_prime(n: int) -> bool:
    ...
```

The main thing to know about `is_prime` is that it does not perform and exhaustive test of primality. It will return `True` or `False` if the primality of `n` can be determined with certainty, otherwise it will `raise NotImplementedError`. This function is only used for sanity checks, so it's fine that it only works with the subset of primes we're actually interested in.

7.2 Implementation of `sbk.primes`

We generate a python module and a test script.

```
# file: src/sbk/primes.py
# include: common.boilerplate
"""Prime constants for sbk.gf.GFNum and sbk.gf.Field."""
# dep: common.imports, constants, impl*
```

7.2.1 Constants

We start with a static/hardcoded definition of the primes we care about. We only care about exponents n which are mutliples of 8 because we will only be encoding secrets with a length in bytes.

```
# def: constants
# dep: pow2primes, small_primes
```

```
# def: pow2prime_params
Pow2PrimeN     : TypeAlias = int
Pow2PrimeK     : TypeAlias = int
Pow2PrimeItem  : TypeAlias = Tuple[Pow2PrimeN, Pow2PrimeK]
Pow2PrimeItems : TypeAlias = Iterator[Pow2PrimeItem]

POW2_PRIME_PARAMS: Dict[Pow2PrimeN, Pow2PrimeK] = {
      8:   5,  16:   15,  24:    3,  32:    5,   40:   87,
     48:  59,  56:    5,  64:   59,  72:   93,   80:   65,
     88: 299,  96:   17, 104:   17, 112:   75,  120:  119,
    128: 159, 136:  113, 144:   83, 152:   17,  160:   47,
    168: 257, 176:  233, 184:   33, 192:  237,  200:   75,
```

```
    208:  299, 216:  377, 224:   63, 232:  567,  240:  467,
    248:  237, 256:  189, 264:  275, 272:  237,  280:   47,
    288:  167, 296:  285, 304:   75, 312:  203,  320:  197,
    328:  155, 336:    3, 344:  119, 352:  657,  360:  719,
    368:  315, 376:   57, 384:  317, 392:  107,  400:  593,
    408: 1005, 416:  435, 424:  389, 432:  299,  440:   33,
    448:  203, 456:  627, 464:  437, 472:  209,  480:   47,
    488:   17, 496:  257, 504:  503, 512:  569,  520:  383,
    528:   65, 536:  149, 544:  759, 552:  503,  560:  717,
    568:  645, 576:  789, 584:  195, 592:  935,  600:   95,
    608:  527, 616:  459, 624:  117, 632:  813,  640:  305,
    648:  195, 656:  143, 664:   17, 672:  399,  680:  939,
    688:  759, 696:  447, 704:  245, 712:  489,  720:  395,
    728:   77, 736:  509, 744:  173, 752:  875,  760:  173,
    768:  825
    # 768:  825, 776: 1539, 784:  759, 792: 1299,  800:  105,
    # 808:   17, 816:  959, 824:  209, 832:  143,  840:  213,
    # 848:   17, 856:  459, 864:  243, 872:  177,  880:  113,
    # 888:  915, 896:  213, 904:  609, 912: 1935,  920:  185,
    # 928:  645, 936: 1325, 944:  573, 952:   99,  960:  167,
    # 968: 1347, 976: 2147, 984:  557, 992: 1779, 1000: 1245,
}
```

If we *do* ever want to serialize a share that uses $\mathrm{GF}(p)$, then we will somehow have to encode which prime is used. That would be done most easilly as an index of `POW2_PRIMES` using only one byte.

```
# exec
# dep: common.imports, pow2prime_params
assert len(POW2_PRIME_PARAMS) < 256
```

Evaluate of the parameters into the actual `POW2_PRIMES`.

```
# def: pow2primes
# dep: pow2prime_params
def pow2prime(n: Pow2PrimeN, k: Pow2PrimeK) -> int:
    if n % 8 == 0:
        return int(2 ** n - k)
    else:
        raise ValueError(f"Invalid n={n} (n % 8 != 0)")


POW2_PRIMES = [
    pow2prime(n, k)
    for n, k in sorted(POW2_PRIME_PARAMS.items())
```



```
        print(f"2**{exp:<4} - {k:<4}", verification_url)

if __name__ == '__main__':
    download_oeis_org_a014234()
```

Truncated output of running the `main` function.

```
# run: bash -c "python src/sbk/primes.py | tail"
2**928  - 645  https://www.wolframalpha.com/input/?i=factors(2%5E928+-+645)
2**936  - 1325 https://www.wolframalpha.com/input/?i=factors(2%5E936+-+1325)
2**944  - 573  https://www.wolframalpha.com/input/?i=factors(2%5E944+-+573)
2**952  - 99   https://www.wolframalpha.com/input/?i=factors(2%5E952+-+99)
2**960  - 167  https://www.wolframalpha.com/input/?i=factors(2%5E960+-+167)
2**968  - 1347 https://www.wolframalpha.com/input/?i=factors(2%5E968+-+1347)
2**976  - 2147 https://www.wolframalpha.com/input/?i=factors(2%5E976+-+2147)
2**984  - 557  https://www.wolframalpha.com/input/?i=factors(2%5E984+-+557)
2**992  - 1779 https://www.wolframalpha.com/input/?i=factors(2%5E992+-+1779)
2**1000 - 1245 https://www.wolframalpha.com/input/?i=factors(2%5E1000+-+1245)
# exit: 0
```

So that you don't need to run the test suite, the `sbk.primes` module is has a `main` funciton which downloads the A014234 dataset…

```
# def: impl_read_oeis_org_a014234
def read_oeis_org_a014234() -> str:
    import tempfile
    import urllib.request

    cache_path = pl.Path(tempfile.gettempdir()) / "oeis_org_b014234.txt"
    min_mtime = time.time() - 10000
    if cache_path.exists() and cache_path.stat().st_mtime > min_mtime:
        with cache_path.open(mode="r") as fobj:
            content = fobj.read()
    else:
        a014234_url = "https://oeis.org/A014234/b014234.txt"
        with urllib.request.urlopen(a014234_url) as fobj:
            data = typ.cast(Union[bytes, str], fobj.read())

        if isinstance(data, bytes):
            content = data.decode("utf-8")
        else:
            content = data

        with cache_path.open(mode="w") as fobj:
            fobj.write(content)
    return content
```

…, runs it throught the `a014234_verify` validation and generates urls for wolframalpha.com, that you can use to double check the constants.

```
# def: impl_download_oeis_org_a014234
def download_oeis_org_a014234() -> None:
    """Helper to verify local primes against https://oeis.org/A014234.

    $ source activate
    $ python -m sbk.primes
    """
    content = read_oeis_org_a014234()
    for exp, k in a014234_verify(content):
        verification_url = f"https://www.wolframalpha.com/input/?i=factors(2%5E{exp}+-+{k})"
```



```
]
```

Now we can provide accessor methods to get the appropriate prime for a given length of data. In some cases we may not want to store the actual prime itself, but rather we can just store the much smaller index the prime in `POW2_PRIME_PARAMS`.

```
# def: impl_get_pow2prime
def get_pow2prime_index(num_bits: int) -> int:
    if num_bits % 8 != 0:
        err = f"Invalid num_bits={num_bits}, not a multiple of 8"
        raise ValueError(err)

    target_exp = num_bits
    for p2pp_idx, param_exp in enumerate(POW2_PRIME_PARAMS):
        if param_exp >= target_exp:
            return p2pp_idx

    err = f"Invalid num_bits={num_bits}, no known 2**n-k primes "
    raise ValueError(err)


def get_pow2prime(num_bits: int) -> int:
    p2pp_idx = get_pow2prime_index(num_bits)
    return POW2_PRIMES[p2pp_idx]
```

7.2.2 Basic Validation

Our main concern here is that we define a constant that isn't actually a prime (presumably by accident), so let's start with some basic sanity checks based on numbers we know to be prime[57].

```
# exec
# dep: common.imports, pow2primes
known_primes = [
    251,
    65521,
    4294967291,
    18446744073709551557,
    340282366920938463463374607431768211297,
    281474976710597,
    79228162514264337593543950319,
```

```
    1461501637330902918203684832716283019655932542929,
    6277101735386680763835789423207666416102355444464034512659,
]
missing_primes = set(known_primes) - set(POW2_PRIMES)
assert not any(missing_primes)
```

We use the small primes for the `basic_prime_test` and as bases for the Miller-Rabin test. I'm not actually sure that prime bases are any better for the MR test than random numbers, it's just a visible pattern from the wikipedia article[58].

Primes oeis.org/A000040[59]

```
# def: small_primes
SMALL_PRIMES = [
      2,   3,   5,   7,  11,  13,  17,  19,  23,
     29,  31,  37,  41,  43,  47,  53,  59,  61,
     67,  71,  73,  79,  83,  89,  97, 101, 103,
    107, 109, 113, 127, 131, 137, 139, 149, 151,
    157, 163, 167, 173, 179, 181, 191, 193, 197,
    199, 211, 223, 227, 229, 233, 239, 241, 251,
    257, 263, 269, 271, 277, 281, 283, 293, 307,
]

PRIMES = sorted(set(SMALL_PRIMES + POW2_PRIMES))
```

7.2.3 Primality Testing

All the primes we actually use are constants and are well known. The primality testing code here is for verification and as a safety net against accidental changes. We start with the most basic test if `n` is a prime.

```
# def: impl_is_prime
# dep: constants
def is_prime(n: int) -> bool:
    for p in PRIMES:
        if n == p:
            return True
        psq = p * p
        if n < psq and n % p == 0:
```


The format from aeis.org is a text file where each line consists of `n` and the largest prime `p` such that $p < 2^n$.

```
# run: bash -c "head test/test_primes_a014234.txt | tr ' ' ':' | tr '\n' ' '"
1:2 2:3 3:7 4:13 5:31 6:61 7:127 8:251 9:509 10:1021
# exit: 0
```

We can calculate $k = 2^n - p$, e.g. $2^8 - 251 = 5$. Assuming we have the content of such a file, we can use it to verify the constants of `POW2_PRIME_PARAMS`.

```
# def: impl_a014234_verify
def a014234_verify(a014234_content: str) -> Pow2PrimeItems:
    for line in a014234_content.splitlines():
        if not line.strip():
            continue

        n, p = map(int, line.strip().split())
        if n % 8 != 0:
            continue

        k = (2 ** n) - p
        assert pow2prime(n, k) == p

        if n <= 768:
            assert POW2_PRIME_PARAMS[n] == k

        yield (n, k)
```

For the tests we'll be nice and not download the file for every test run and instead use a local copy. Note that `a014234_verify` uses assertions internally and the assertions of the test itself just make sure that the content had some entries that were yielded (which wouldn't be the case if `content` were empty for example).

```
# def: test_a014234_verfiy
def test_a014234_verfiy():
    fixture = pl.Path(__file__).parent / "test_primes_a014234.txt"
    with fixture.open(mode="r") as fobj:
        content = fobj.read()

    p2p_primes = list(sbk.primes.a014234_verify(content))
    assert len(p2p_primes) >= 96, "not enough entries in content"
```

```python
# def: impl_validate_pow2_prime_params
# Hardcoded digest of POW2_PRIME_PARAMS
_V1_PRIMES_VERIFICATION_SHA256 = "8303b97ae70cb01e36abd0a625d7e8a427569cc656e861d90a94c3bc697923e7"


def validate_pow2_prime_params() -> None:
    sha256 = hashlib.sha256()
    for n, k in sorted(POW2_PRIME_PARAMS.items()):
        sha256.update(str((n, k)).encode('ascii'))

    digest      = sha256.hexdigest()
    has_changed = len(POW2_PRIME_PARAMS) != 96 or digest != _V1_PRIMES_VERIFICATION_SHA256

    if has_changed:
        logger.error(f"Current  hash: {digest}")
        logger.error(f"Expected hash: {_V1_PRIMES_VERIFICATION_SHA256}")
        raise Exception("Integrity error: POW2_PRIMES changed!")


validate_pow2_prime_params()
```

With this test, we verify that any manipulation the `POW2_PRIME_PARAMS` list will cause the digest to change.

```python
# def: test_primelist_validation
def test_primelist_validation():
    sbk.primes.validate_pow2_prime_params()
    _original = sorted(sbk.primes.POW2_PRIME_PARAMS.items())
    try:
        sbk.primes.POW2_PRIME_PARAMS[-1] = (768, 1)
        sbk.primes.validate_pow2_prime_params()
        assert False, "expected Exception"
    except Exception as ex:
        assert "Integrity error" in str(ex)
    finally:
        sbk.primes.POW2_PRIME_PARAMS = _original
```

Finally we perform some validation against oeis.org. This is where the parameters for `n` and `k` originally came from, so it is mainly a validation in the sense that it help to convince you that no mistake was made.


```python
            return False

    # This is not an exhaustive test, it's only used used only to
    # catch programming errors, so we bail if can't say for sure that
    # n is prime.
    if n > max(SMALL_PRIMES) ** 2:
        raise NotImplementedError
    else:
        return True
```

The MR test is only used for validation of the constants declared in `POW2_PRIMES`. The implementation was developed using the following resources:

- en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Miller_test[60]
- jeremykun.com/2013/06/16/miller-rabin-primality-test/[61]
- miller-rabin.appspot.com/[62]
- gist.github.com/Ayrx/5884790[63]

```python
# def: impl_is_probable_prime
# include: _miller_test_bases, _is_composite
def is_probable_prime(n: int, k: int = 100) -> bool:
    # Early exit if not prime
    for p in SMALL_PRIMES:
        if n == p:
            return True
        if n % p == 0:
            return False

    r = 0
    d = n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    for a in _miller_test_bases(n, k):
        x = pow(a, d, n)
        if x not in (1, n - 1) and _is_composite(n, r, x):
            return False

    return True
```

```
# def: _miller_test_bases
from random import randrange

# Jim Sinclair
_mr_js_bases = {2, 325, 9375, 28178, 450775, 9780504, 1795265022}


def _miller_test_bases(n: int, k: int, accuracy: int = 100) -> Set[int]:
    if n < 2 ** 64:
        return _mr_js_bases
    else:
        random_bases = {randrange(2, n - 1) for _ in range(accuracy)}
        return (_mr_js_bases | set(SMALL_PRIMES[:13]) | random_bases)
```

```
# def: _is_composite
def _is_composite(n: int, r: int, x: int) -> bool:
    for _ in range(r - 1):
        x = pow(x, 2, n)
        if x == n - 1:
            return False
    return True
```

Basic test of is_probable_prime.

```
# def: test_setup
# dep: common.imports, constants, pow2primes, impl_is_probable_prime
```

```
# exec
# dep: test_setup
assert is_probable_prime(2 ** 127 -  1)
assert is_probable_prime(2 **  64 - 59)
assert not is_probable_prime(60)

# http://oeis.org/A020230
assert not is_probable_prime(7 * 73 * 103)
assert not is_probable_prime(89 * 683)
assert not is_probable_prime(42420396931)
```

Test the constants with `is_probable_prime`.

```
# exec
# timeout: 90
# dep: test_setup
```



```
for i, (n, k) in enumerate(POW2_PRIME_PARAMS.items()):
    prime = POW2_PRIMES[i]
    assert prime == 2 ** n - k, (i, n, k)
    assert is_probable_prime(prime), (n, k)
```

7.2.4 Validation

Here we want to make sure the parameters don't change inadvertantly. If we encode any shares that use these primes, we want to be sure that we can decode them later on. We could encode the prime we use for the share (or the parameters `n` and `k`, but the smallest encoding uses only the index of the prime in the `POW2_PRIMES` list. For such an encoding to work, we have to be sure that we preserve the same primes at the same indexs, otherwise a share would become useless or the user would have to know which version of the software was used to create some old shares.

For the verification, we simply greate a string representation of the `POW2_PRIME_PARAMS` and hard-code its digest, which should never change.

```
    assert out_params.sss_x    == -1
    assert out_params.sss_t    == MIN_THRESHOLD
```

```
# def: fuzztest_harness
import random

rand = random.Random(0)

kwargs_range = {
    'kdf_m'   : [rand.randint(1, 1000000) for _ in range(100)],
    'kdf_t'   : [rand.randint(1, 10000) for _ in range(100)],
    'sss_x'   : list(range(1, 2**5)),
    'sss_t'   : list(range(2, 2**3 + 2)),
}

for _ in range(100):
    kwargs = {
        k: rand.choice(choices)
        for k, choices in kwargs_range.items()
    }
    in_params = init_parameters(**kwargs)
    validate_params(in_params)

print("ok")
```

```
# file: test/fuzztest_share_params.py
# dep: validate_share_params, fuzztest_harness
```

```
# file: test/fuzztest_salt_params.py
# dep: validate_salt_params, fuzztest_harness
```

```
# run: bash scripts/lint.sh test/fuzztest_*.py
# exit: 0
```

```
# run: python test/fuzztest_share_params.py
ok
# exit: 0
```

```
# run: python test/fuzztest_salt_params.py
ok
# exit: 0
```

9.6 Utils

Parse various data lengths. An initial design intended these to be variable, but that design was disregarded to simplify the initial implementation. The `SecretLens` construct remains.

```
# def: impl_len_utils
class SecretLens(NamedTuple):
    raw_salt  : int
    brainkey  : int
    master_key: int
    raw_share : int
    salt      : int
    share     : int


def raw_secret_lens() -> SecretLens:
    raw_salt = DEFAULT_RAW_SALT_LEN
    brainkey = DEFAULT_BRAINKEY_LEN

    raw_share  = raw_salt + brainkey
    master_key = raw_salt + brainkey
    salt  = SALT_HEADER_LEN  + raw_salt
    share = SHARE_HEADER_LEN + raw_share
    return SecretLens(raw_salt, brainkey, master_key, raw_share, salt, share)
```


55. https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction
56. https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction#Systematic_encoding_procedure:_The_message_as_an_initial_sequence_of_values
57. https://oeis.org/A132358
58. https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Testing_against_small_sets_of_bases
59. https://oeis.org/A000040/list
60. https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Miller_test
61. https://jeremykun.com/2013/06/16/miller-rabin-primality-test/
62. http://miller-rabin.appspot.com/
63. https://gist.github.com/Ayrx/5884790
64. https://datatracker.ietf.org/doc/draft-irtf-cfrg-argon2/13/
65. https://github.com/P-H-C/phc-winner-argon2#command-line-utility
66. https://pypi.org/project/argon2-cffi/
67. https://www.password-hashing.net/argon2-specs.pdf
68. https://cryptobook.nakov.com/mac-and-key-derivation/argon2
69. https://www.ory.sh/choose-recommended-argon2-parameters-password-hashing/
70. https://www.twelve21.io/how-to-choose-the-right-parameters-for-argon2/
71. https://argon2-cffi.readthedocs.io/en/stable/
72. https://argon2-cffi.readthedocs.io/en/stable/api.html#argon2.low_level.hash_secret_raw
73. http://www.coding2learn.org/blog/2013/07/29/kids-cant-use-computers/
74. https://github.com/mbarkhau/sbk
75. https://github.com/trezor/python-shamir-mnemonic/

31. https://en.wikipedia.org/wiki/Data_remanence
32. https://www.youtube.com/watch?v=iFY5SyY3IMQ
33. https://www.youtube.com/watch?v=K54ildEW9-Q
34. https://en.wikipedia.org/wiki/Finite_field
35. https://oeis.org/A014234
36. https://github.com/satoshilabs/slips/blob/master/slip-0039.md#shamirs-secret-sharing
37. https://doi.org/10.6028/NIST.FIPS.197
38. https://github.com/P-H-C/phc-winner-argon2
39. https://github.com/satoshilabs/slips/blob/master/slip-0039.md#design-rationale
40. https://regex101.com/r/v9eqiM/2
41. https://blog.trezor.io/passphrase-the-ultimate-protection-for-your-accounts-3a311990925b
42. https://en.wikipedia.org/wiki/Key_derivation_function
43. https://github.com/P-H-C/phc-winner-argon2
44. https://antelle.net/argon2-browser/
45. https://gitlab.com/mbarkhau/sbk/issues/1
46. https://en.wikipedia.org/wiki/Acid-free_paper
47. https://en.wikipedia.org/wiki/Pouch_laminator
48. https://www.youtube.com/watch?v=TrB62cPPNxc
49. https://blog.lopp.net/metal-bitcoin-seed-storage-stress-test-round-iii/
50. https://jlopp.github.io/metal-bitcoin-storage-reviews/
51. https://youtu.be/xneBjc8z0DE?t=2460
52. https://en.wikipedia.org/wiki/Spaced_repetition
53. https://en.wikipedia.org/wiki/Method_of_loci
54. https://en.wikipedia.org/wiki/Forward_error_correction

KDF: Key Derivation

As mentioned in User Guide, the KDF is used to make a brute-force attack expensive and indeed infeasable. It should not be possible for an attacker, even with access to the `salt` (but **without** access to the `brainkey`), to recover a wallet. This means we must make it infeasable to calculate a significant fraction of $256^6 = 2^{48}$ hashes.

With the `sbk.kdf` module we have a two things we need to accomodate for.

1. Provide an API for correct use of the `argon2` library.
2. Implement a meaningful progress meter, so that the UI can signal to the user that expensive key derivation is in progress.

10.1 Public API of `sbk.kdf`

```
def digest(
    data       : bytes,
    params     : parameters.KDFParams,
    hash_len   : int,
    progress_cb: ct.ProgressCallback | None = None,
) -> bytes:
    ...
```

10.2 KDF - Implementation

```
# file: src/sbk/kdf.py
# include: common.boilerplate
# dep: common.imports, imports, constants*, impl*, main
```

```
# run: bash scripts/lint.sh src/sbk/kdf.py
# exit: 0
```

The internal digest function uses the python argon2-cffi[71] library. We determined earlier in kdf_parameters that this implementation matched others both in terms of its output and in terms of performance.

```
# def: imports
import argon2

from sbk import utils
from sbk import parameters
```

To implement a meaningful progress bar, we will split the digest into steps. It would be nice if we had an easy way to hook into the argon2 implementation and inspect it or provide a callback which would give us information about its progress. Instead we split the whole calculation into steps and measure the time for each step to determine the progress.

We will feed the hash output of one step into the following iteration as input, the `HASH_LEN` is chosen to be much larger than the original input. Without having done any investigation, my assumption is that this makes loss of entropy between each iteration negligable and does not e.g. reduce the search space for the final step so much that an attacker could simply skip all but the last step. Feedback welcome.

```
# def: constants
HASH_LEN = 128
DIGEST_STEPS = 10
MEASUREMENT_SIGNIFICANCE_THRESHOLD = ct.Seconds(2)
```

We wrap the internal function hash_secret_raw[72]. This way we can use our own types throughout sbk and only do the mapping to argon2 conventions once.

```
# def: impl_digest_step
def _digest(data: bytes, p: ct.Parallelism, m: ct.MebiBytes, t: ct.Iterations)
↪ -> bytes:
    result = argon2.low_level.hash_secret_raw(
        secret=data,
```


9. https://www.cato.org/sites/cato.org/files/articles/hayek-use-knowledge-society.pdf
10. https://en.bitcoin.it/wiki/Multisignature
11. https://electrum.readthedocs.io/en/latest/multisig.html
12. https://keybase.io/warp/warp_1.0.9_SHA256_a2067491ab582bde779f4505055807c2479354633a2216b22cf1e92d1a6e4a87.html
13. https://wiki.trezor.io/Shamir_Backup
14. https://rufus.ie/
15. https://ubuntu.com/tutorials/tutorial-create-a-usb-stick-on-ubuntu
16. https://en.wikipedia.org/wiki/Cold_boot_attack
17. https://tails.boum.org/contribute/design/memory_erasure/
18. https://twitter.com/PeterSchiff/status/1220135541330542592
19. https://en.m.wikipedia.org/wiki/Mt._Gox
20. https://media.defense.gov/2020/Jan/14/2002234275/-1/-1/0/CSA-WINDOWS-10-CRYPT-LIB-20190114.PDF
21. https://en.bitcoin.it/wiki/Cold_storage
22. https://en.bitcoin.it/wiki/Hot_wallet
23. https://sbk.dev/downloads/
24. https://www.youtube.com/watch?v=c0O6UXrOZJo
25. http://www.dkriesel.com/en/blog/2013/0802_xerox-workcentres_are_switching_written_numbers_when_scanning
26. https://sbk.dev/downloads/template_a4.pdf
27. https://sbk.dev/downloads/template_us_letter.pdf
28. https://en.wikipedia.org/wiki/Plausible_deniability
29. https://blog.keys.casa/shamirs-secret-sharing-security-shortcomings/
30. https://en.wikipedia.org/wiki/Luby_transform_code

11. Reasons it may have been fine to use $\mathrm{GF}(\mathrm{p})$

    - A common reason to use $\mathrm{GF}(256)$ is to be compatible with low-end systems. Since SBK uses a computationally and memory intensive KDF, systems with constrained CPU and RAM defeat the purpose of SBK and are not a target. Such systems would either take a long time to derive a hardened `wallet-seed` or these seeds would be cracked more easily by machines that are much more powerful and easily obtained.
    - $\mathrm{GF}(256)$ uses arithmetic that is natively supported by practically every programming language and hardware platform. Depending on the size of p, a $\mathrm{GF}(\mathrm{p})$ field requires support for big integers. Python has native support for big integers, so arithmetic with large values is not an issue for SBK. Since SBK uses Electrum (which is implemented with python), it is not an extra dependency for SBK to require a python interpreter.
    - Implementing finite field arithmetic for $\mathrm{GF}(\mathrm{p})$ is slightly easier to understand and should be easier to review.

12. I was quite happy to see the same numbers pop out as for the reference implementation of SLIP0039[75]

1. https://www.youtube.com/watch?v=AcrEEnDLm58
2. https://electrum.org
3. https://gitlab.com/mbarkhau/sbk/blob/master/LICENSE
4. https://blog.keys.casa/shamirs-secret-sharing-security-shortcomings/
5. https://en.bitcoin.it/wiki/Shamir_Secret_Snakeoil
6. https://www.youtube.com/watch?v=5WWfQM0SFXQ
7. https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing
8. https://xkcd.com/538/


```
        salt=data,
        hash_len=HASH_LEN,
        parallelism=p,
        memory_cost=m * 1024,
        time_cost=t,
        type=argon2.low_level.Type.ID,
    )
    return typ.cast(bytes, result)
```

```
# exec
# dep: common.imports, imports, constants, impl_digest_step
data = _digest(b"Not your keys, not your coins", p=128, m=32, t=8)
assert len(data) == HASH_LEN
while data:
    print(utils.bytes2hex(data[:32]))
    data = data[32:]
```

```
# out
2d2eb3584a94cf592b9b2bbe0fa26b215fb9eb955d5cf6c46dbbdcf16b651f46
c74217a0d3c76a57c705edb5eb6db37cfe4963ca807b9302388c61434516abc4
4dd473a83ebab10b5708ae3c93b56bbdd09cf0852a1d6cb30847ce83ebfb18f9
243d249b4eb4e1da19ae0b97f6a2d89c824286d26c539443ecab0d7b7f191bbc
# exit: 0
```

```
# def: impl_digest
def digest(
    data       : bytes,
    kdf_params : parameters.KDFParams,
    hash_len   : int,
    progress_cb: ct.MaybeProgressCallback = None,
) -> bytes:
    _ps          : Optional[utils.ProgressSmoother]
    if progress_cb:
        _ps = utils.ProgressSmoother(progress_cb)
    else:
        _ps = None

    remaining_iters   = kdf_params.kdf_t
    remaining_steps   = min(remaining_iters, DIGEST_STEPS)

```

```
    progress_per_iter = 100 / kdf_params.kdf_t

    constant_kwargs = {
        'p': kdf_params.kdf_p,
        'm': kdf_params.kdf_m,
    }
    result = data

    while remaining_iters > 0:
        step_iters = max(1, round(remaining_iters / remaining_steps))
        result     = _digest(result, t=step_iters, **constant_kwargs)
        sys.stdout.flush()

        if _ps:
            _ps.progress_cb(step_iters * progress_per_iter)

        remaining_iters -= step_iters
        remaining_steps -= 1

    assert remaining_iters == 0, remaining_iters
    assert remaining_steps == 0, remaining_steps

    if _ps:
        _ps.join()

    return result[:hash_len]
```

```
# def: main
def main(args: List[str]) -> int:
    memory_mb  = int(args[0])
    kdf_p, kdf_m, kdf_t = parameters.init_kdf_params(kdf_m=memory_mb, kdf_t=1)
    try:
        _digest(b"saltsaltsaltsaltbrainkey", kdf_p, kdf_m, kdf_t)
        return 0
    except argon2.exceptions.HashingError:
        return -1

if __name__ == '__main__':
    sys.exit(main(sys.argv[1:]))
```

The main function is used separately by `sbk.sys_info` to detect how much memory we can use with the digest function.

1. Throughout the documentation I will use upper case "Bitcoin" to mean the project and lower case "bitcoin" to mean a quantity of the digital asset.
2. If you consider yourself tech illiterate[73], SBK may be a bit challanging. If you take your time you should be able to manage, so long as you don't blindly proceed when you don't understand what you're doing.
3. This project has nothing to do with an ERC20 token that apparently exists. I will not even dignify it with a link.
4. The SBK project is not associated with the Electrum Bitcoin Wallet or Electrum Technologies GmbH.
5. SBK may well at some point be implemented as an Electrum plugin. You are welcome to contribute at github.com/mbarkhau/sbk[74].
6. Nothing of value has truly been lost, but your purchasing power has been redistributed to the remaining holders of bitcoin.
7. The SBK project is not associated with SatoshiLabs s.r.o.
8. SBK is not designed for institutions where more than one person will usually have joint custody over bitcoin belonging to a treasury. For this use-case you should look into a multi-signature setup.
9. Much larger than most people can (or at least are willing to) memorize in their head.
10. You do need to trust the development process for the wallet software that you use. SBK is Open Source and will also provide a bounty once a final version is released.

Footnotes and Links


```
# run: python -m sbk.kdf 100
# exit: 0
```

```
# def: impl_kdf_params_for_duration
def kdf_params_for_duration(
    baseline_kdf_params : parameters.KDFParams,
    target_duration     : ct.Seconds,
    max_measurement_time: ct.Seconds = 5,
) -> parameters.KDFParams:
    test_kdf_params = parameters.init_kdf_params(kdf_m=baseline_kdf_params.kdf_m, kdf_t=1)
    digest_kwargs = {
        # we only vary t, the baseline should be chosen to max out the others
        'p': test_kdf_params.kdf_p,
        'm': test_kdf_params.kdf_m,
    }

    tgt_step_duration = target_duration / DIGEST_STEPS
    total_time        = 0.0

    while True:
        tzero = time.time()
        digest_kwargs['t'] = test_kdf_params.kdf_t
        _digest(b"\x00\x00\x00\x00\x00\x00\x00\x00\x00", **digest_kwargs)
        duration = time.time() - tzero
        total_time += duration

        iters_per_sec = test_kdf_params.kdf_t / duration
        step_iters    = tgt_step_duration * iters_per_sec * 1.25

        # t = test_kdf_params.kdf_t
        # print(f"< {duration:4.3f} t: {t} i/s: {iters_per_sec} tgt: {step_iters}")
        is_tgt_exceeded            = duration   > tgt_step_duration
        is_measurement_significant = duration   > MEASUREMENT_SIGNIFICANCE_THRESHOLD
        is_enough_already          = total_time > max_measurement_time
        if is_tgt_exceeded or is_measurement_significant or is_enough_already:
```

```
            new_t = round(step_iters * DIGEST_STEPS)
            return parameters.init_kdf_params(kdf_m=test_kdf_params.kdf_m, kdf_t=new_t)
        else:
            # min_iters is used to make sure we're always measuring with a higher value for t
            min_iters       = math.ceil(test_kdf_params.kdf_t * 1.25)
            min_t           = round(1.25 * MEASUREMENT_SIGNIFICANCE_THRESHOLD * iters_per_sec)
            new_t           = max(min_iters, min_t)
            test_kdf_params =  parameters.init_kdf_params(kdf_m=test_kdf_params.kdf_m, kdf_t=new_t)
```



```

    return (result, duration)
```

We use a fairly low level and explicit api here mainly to validate against the test vectors of the IETF test

```
# exec
# dep: measure_digest

expected = '0d640df58d78766c08c037a34a8b53c9d01ef0452d75b65eb52520e96b01e659'
result, _ = measure_digest(p=4, m=32, t=1)
assert len(result) == len(expected)
print(result, result == expected)
```

```
# out
656d3661f9c30da2edd65a9b2a3ee3f02e3ce69df00e3c31d89cf9aecfda90f7 False
# exit: 0
```

```python

def measure_digest(p: int, m: int, t: int) -> Tuple[str, Seconds]:
    version = argon2.low_level.ARGON2_VERSION
    assert version == 19, version

    tzero = time.time()

    hash_len = 32
    password = b"\x01" * 32
    salt     = b"\x02" * 16
    secret   = b"\x03" * 8
    adata    = b"\x04" * 12

    # Make sure you keep FFI objects alive until *after* the core call!

    cpassword = ffi.new("uint8_t[]", password)
    csalt     = ffi.new("uint8_t[]", salt)
    csecret   = ffi.new("uint8_t[]", password)
    cadata    = ffi.new("uint8_t[]", adata)

    cout      = ffi.new("uint8_t[]", hash_len)
    ctx = ffi.new(
        "argon2_context *", dict(
            version=version,
            out=cout, outlen=hash_len,
            pwd=cpassword, pwdlen=len(password),
            salt=csalt, saltlen=len(salt),
            secret=csecret, secretlen=len(secret),
            ad=cadata, adlen=len(adata),
            t_cost=t,
            m_cost=m,
            lanes=p,
            threads=1,
            allocate_cbk=ffi.NULL, free_cbk=ffi.NULL,
            flags=argon2.low_level.lib.ARGON2_DEFAULT_FLAGS,
        )
    )

    argon2.low_level.core(ctx, argon2.low_level.Type.ID.value)

    result_data = bytes(ffi.buffer(ctx.out, ctx.outlen))

    duration = int((time.time() - tzero) * 1000)
    result = result_data.hex()
```


System Information

The appropriate kdf parameters depend on the hardware available on the system where sbk is running. To provide the best protection the against a brute force attack, We want to use a substantial portion of the available system memory.

To this end, the `sbk.sys_info` implements parsing and evaluation of the usable memory of the users system.

11.1 Module: `sbk.sys_info`

```python
# file: src/sbk/sys_info.py
# include: common.boilerplate
# dep: common.imports, imports, common.constants, constants*, impl*, main
```

```python
# run: bash scripts/lint.sh src/sbk/sys_info.py
# exit: 0
```

```python
# def: imports
from . import kdf
from . import parameters
```

11.1.1 Module Main and Selftest

We start with a selftest function to illustrate and exercise teh system information we will be gathering.

```python
# def: main
def main() -> int:
    # xinclude: common.debug_logging
    print("lang: ", detect_lang())
    print("Mem Info:", memory_info())
    print("Memory Info (uncached):", _init_sys_info())
```

```
    print("Memory Info (cached)  :", load_sys_info())
    return 0

if __name__ == '__main__':
    main()
```

```
# run: python -m sbk.sys_info
# options: {"timeout": 90, "capture_file": "captures/sys_info.json"}
lang:  en
Mem Info: (15886, 13722)
Memory Info (uncached): SystemInfo(total_mb=15886, usable_mb=10291)
Memory Info (cached)  : SystemInfo(total_mb=15886, usable_mb=10291)
# exit: 0
```

11.2 Language Detection

SBK currently only supports English. It is unlikely we will support region specific languages such as `en_US`, `en_GB`, `en_AU`, only `en`. In the future we want to enable translation when available and this detection is just in preparation for that.

Furthermore, if non-phonetic scripts are ever to be supported, the whole concept of edit distance to match words of a language specific wordlist will have to be reconsidered.

```
# def: constants_lang
DEFAULT_LANG = ct.LangCode('en')
SUPPORTED_LANGUAGES = {'en'}

# PR welcome
# SUPPORTED_LANGUAGES |= {'es', 'pt', 'ru', 'fr', de', 'it', 'tr'}
#
# non-phonetic systems may be a design issue for wordlists
# SUPPORTED_LANGUAGES |= {'ar', 'ko', 'cn', 'jp'}

KB_LAYOUT_TO_LANG = {'us': 'en'}
```


Design Considerations

- Tradoffs: Usability, Security, Convenience, Long Term Peace of Mind
- Why not Split only the Brainkey and distribute the salt to everybody?
    - Less transcription, perhaps
    - Less secure, because anybody with the salt can start to brute force

12.1 Links

- Hamming Codes by 3Blue1Brown: https://www.youtube.com/watch?v=X8jsijhllIA

12.2 Future Work

- It may very well be appropriate to implement SBK as a Plugin for Electrum
- usage example https://www.reddit.com/r/Bitcoin/comments/hqeqnn/update_mom_made_btc_wallet_before_she_died_and_we/
- alternative https://github.com/lacksfish/insure-gui

12.3 Digest Measurement

```
# def: measure_digest
import time
from typing import Tuple
import argon2
from argon2.low_level import ffi

Seconds = float

```

```python
        if not _SYS_INFO_KW and cache_path.exists():
            try:
                with cache_path.open(mode="rb") as fobj:
                    _SYS_INFO_KW.update(json.load(fobj))
            except Exception as ex:
                logger.warning(f"Error reading cache file {cache_path}: {ex}")

        if _SYS_INFO_KW:
            return SystemInfo(
                total_mb=_SYS_INFO_KW['total_mb'],
                usable_mb=_SYS_INFO_KW['usable_mb'],
            )

    return _init_sys_info()
```

The serialization logic writes a file in `SBK_APP_DIR` which may be loaded the next time sbk is run. The cache file is always updated whenever the memory info is evaluated (which typically should only happen once).

```python
# def: impl_sys_info_dump_cache
_SYS_INFO_KW: Dict[str, int] = {}

def _dump_sys_info(sys_info: SystemInfo) -> None:
    _SYS_INFO_KW.update({
        'total_mb' : sys_info.total_mb,
        'usable_mb': sys_info.usable_mb,
    })

    cache_path = SYSINFO_CACHE_FPATH
    try:
        cache_path.parent.mkdir(exist_ok=True, parents=True)
    except Exception as ex:
        logger.warning(f"Unable to create cache dir {cache_path.parent}: {ex}")
        return

    try:
        with cache_path.open(mode="w", encoding="utf-8") as fobj:
            json.dump(_SYS_INFO_KW, fobj, indent=4)
    except Exception as ex:
        logger.warning(f"Error writing cache file {cache_path}: {ex}")
```



```python
# def: impl_detect_lang
def detect_lang() -> ct.LangCode:
    try:
        localectl_output = sp.check_output("localectl").decode("utf-8")
        lang = _parse_lang(localectl_output)
        kb_lang = _parse_keyboard_lang(localectl_output)
        return lang or kb_lang or DEFAULT_LANG
    except Exception:
        logger.warning("Fallback to default lang: en", exc_info=True)
        return ct.LangCode('en')
```

```python
# def: impl_parse_lang
def _parse_lang(localectl_output: str) -> Optional[ct.LangCode]:
    lang_match = re.search(r"LANG=([a-z]+)", localectl_output)
    if lang_match:
        lang = lang_match.group(1)
        logger.debug(f"lang: {lang}")
        if lang in SUPPORTED_LANGUAGES:
            return ct.LangCode(lang)
    return None
```

```python
# def: impl_parse_keyboard_lang
def _parse_keyboard_lang(localectl_output: str) -> Optional[ct.LangCode]:
    keyboard_match = re.search(r"X11 Layout: ([a-z]+)", localectl_output)
    if keyboard_match:
        layout = keyboard_match.group(1)
        logger.debug(f"keyboard: {layout}")
        if layout in KB_LAYOUT_TO_LANG:
            return ct.LangCode(KB_LAYOUT_TO_LANG[layout])
    return None
```

11.3 Memory Detection

```python
# def: constants
# Fallback value for systems on which total memory cannot be detected
FALLBACK_MEM_MB = int(os.getenv("SBK_FALLBACK_MEM_MB", "1024"))

# cache so we don't have to check usable memory every time
SYSINFO_CACHE_FPATH = SBK_APP_DIR / "sys_info_measurements.json"
```

```python
# def: impl_type_sysinfo
class SystemInfo(NamedTuple):
    total_mb : ct.MebiBytes
    usable_mb: ct.MebiBytes
```

While `/proc/meminfo` is Linux specific, this is the only OS we really care about anyway.

```python
# def: impl_parse_meminfo
def _parse_meminfo(meminfo_text: str) -> Tuple[ct.MebiBytes, ct.MebiBytes]:
    total_mb = FALLBACK_MEM_MB
    avail_mb = FALLBACK_MEM_MB

    for line in meminfo_text.splitlines():
        if line.startswith("Mem"):
            key, num, unit = line.strip().split()
            if key == "MemTotal:":
                assert unit == "kB"
                total_mb = int(num) // 1024
            elif key == "MemAvailable:":
                assert unit == "kB"
                avail_mb = int(num) // 1024
    return (total_mb, avail_mb)


def memory_info() -> Tuple[ct.MebiBytes, ct.MebiBytes]:
    meminfo_path = pl.Path("/proc/meminfo")
    if meminfo_path.exists():
        try:
            with meminfo_path.open(mode="r", encoding="utf-8") as fobj:
                return _parse_meminfo(fobj.read())
        except Exception:
            logger.warning("Error while evaluating system memory", exc_info=True)
    return (FALLBACK_MEM_MB, FALLBACK_MEM_MB)
```

We could use a binary search approach to determine how much memory we can use, but there is a tradeoff with usability here. We don't want the memory check take too long, so we start with what the system tells us we can use and then only reduce that if there are are any issues.


```python
# def: impl_init_sys_info
def _init_sys_info() -> SystemInfo:
    total_mb, avail_mb = memory_info()

    check_mb = avail_mb
    while check_mb > 100:
        logger.debug(f"testing check_mb={check_mb}")
        if _is_usable_kdf_m(check_mb):
            break
        else:
            check_mb = int(check_mb * 0.75)    # try a bit less

    usable_mb = max(check_mb, 100)
    nfo = SystemInfo(total_mb, usable_mb)
    _dump_sys_info(nfo)
    return nfo
```

Our measurement function invokes a subprocess, as the argon2-cffi will kill the process it's running on if too high a value is specified for memory. I am not aware of a more elegant way to determine the maxiumum memory we can use.

```python
# def: impl_measure
def _is_usable_kdf_m(memory_mb: ct.MebiBytes) -> bool:
    retcode = sp.call([sys.executable, "-m", "sbk.kdf", str(memory_mb)])
    return retcode == 0
```

11.4 SysInfo Caching

It can take a few seconds to measure how much memory we can use for the kdf, but that information won't change from one run to the next. It's worthwhile to cache this information for later use.

The primary public function `load_sys_info` encapsulates the high level caching logic, both within the current process (using `_SYS_INFO_KW`) and on disk.

```python
# def: impl_load_sys_info
def load_sys_info(use_cache: bool = True) -> SystemInfo:
    if use_cache:
        cache_path = SYSINFO_CACHE_FPATH
```