Update 12/22/2016

We've integrated a great deal of feedback over the past few weeks -- thanks to all who have contributed. We're actively developing the full protocol and welcome additional feedback.

Preface for Community Reviewers

Secure cold storage of bitcoins is difficult, and near-impossible for an amateur. We are solving this problem and would like your feedback on our approach.

We attempted to follow the consensus advice for creating secure bitcoin storage - setting up multi-sig paper wallets using air-gapped computers. To our surprise, this common advice was difficult to follow. Obscure risk vectors are everywhere, and it is difficult to identify, evaluate, and mitigate all of them. There were also a confusing variety of tools to choose from at each step, most of which weren't built around this use case.

We were surprised to discover there were no good tutorials for navigating this process, despite Bitcoin being several years old. This should *not* be a gap in the Bitcoin ecosystem in 2016!

We are solving this problem creating an open source, step-by-step guide that removes all confusion from the process of creating secure cold storage.

As a first step, we've written this design document detailing the technical decisions we have made so far. This is not the step-by-step guide, but a summary that we have put together for more efficient critique.

Our hope is that this project, when completed, will serve as the *de facto* guide for anyone looking for secure cold storage of bitcoin. We want it to be as close to bulletproof as a reasonable person can get. We also hope it will inspire developers to make cold storage easier than it is with currently available tools.

Feedback Needed

We invite your feedback. **Please feel free to comment anywhere in this document using the Insert -> Comment feature in Google Docs.**

If you offer substantial feedback, we will thank you in the acknowledgments of the final publication, unless you specifically request to remain anonymous.

Thanks for your time!

Glacier Design Document

Version 0.1 Alpha

James Hogan, Jacob Lyles

Introduction

This document describes the design approach to Glacier, a protocol for securely manage one's own storage of Bitcoins (as opposed to using a third party wallet service). It is designed for the following use case:

1. Personal storage (vs. institutional)
2. A large amount of Bitcoin ($100,000+, no maximum)
3. Long-term storage (years or decades)
4. Infrequent transactions (i.e. savings, not spending)
5. No highly-resourced targeted attacks (e.g. being personally targeted by a sophisticated criminal organization)
6. Technical unskilled users (diligence and technical literacy will be required, but not expertise)

This document outlines the high-level approach for the Glacier protocol. The full step-by-step protocol is available here.

Overview

The full Glacier protocol is a document which will outline a highly-detailed, step-by-step process for high-security cold storage of bitcoins, plus with a small piece of companion software, GlacierScript.

Key aspects of Glacier include:

- Multi-signature wallets, with processes & sufficient key count to account for scenarios such as death of a signatory, lost keys, etc.

- Keys stored on paper, in separate locations, with processes for periodic visual inspection to verify continued custody & integrity

- Combining entropy from two sources, casino-grade dice and /dev/random, to defend against compromise of either single source

- Performing all crypto operations on factory-sealed computers, with all wireless hardware removed, never to be connected to a network

- Performing all crypto operations in an Ubuntu instance, booted from USB & running on RAM disk, so no data can be written to persistent media

- Only moving data (i.e. private keys, signed transactions) off the aforementioned “crypto computers” via hand transcription or QR codes (no networks, USB sticks, printers)

- Taking care to avoid malware infection of the crypto computers (by using factory-sealed USB drives w/cryptographically signed firmware to move application software to the computer, verifying checksums of all software packages, etc.)

- Using deterministic methods to generate keys from entropy, and running all processes that generate sensitive data (keys, addresses, transactions) *twice,* once on each of two different secure computing environments. This detects attacks which generate or display flawed data -- if data matches, compromise is highly unlikely.

- Accepting some tradeoffs due to lack of available tools / ease of initial implementation: Allowing address reuse, and no BIP39 / HD wallet support

The remainder of this document describes the reasoning behind the above decisions.

Design Principles

Principle 1: Very Low Risk Tolerance

Striving for "maximum" security is both futile (there's no such thing as zero risk) and expensive (security usually comes at a cost). Indeed, to the extent we "overshoot" and design Glacier to be *too* secure, it will also be *unnecessarily costly*. Fewer people will use it, and it will do less good in the world.

While mindful of this, we nonetheless consciously decided to adopt a **very low risk tolerance** and take steps to defend against obscure attack vectors. We think this is appropriate because **attack vectors which are obscure today may be common tomorrow:**

- Glacier is intended to be appropriate for long-term storage. If Bitcoin adoption explodes, the incentive for attackers explodes proportionately. You have to assume that well-funded criminal organizations will deploy sophisticated malware in the years ahead.

- As common attack vectors get closed due to maturing security tools and practices, attackers will be forced to use the more obscure attack vectors.

- If Glacier becomes popular, it will be used to store a lot of funds, and attackers will have an incentive to exploit Glacier's specific vulnerabilities, even if those vulnerabilities are obscure today.

Principle 2: Prefer Risk Elimination over Risk Reduction

When looking at an obscure risk vector, it can be tempting to say "this is too far-fetched -- we don't need to protect against this."

For example, one may consider generating private keys using dice, because random number generators can have vulnerabilities. But should we worry about *which* dice we use? Regular dice certainly are biased, but is this bias actually exploitable to guess a private key, or is that just paranoia?

Honestly, we don't know. It's difficult to assess, but we do know that hackers surprise us sometimes. If we frequently assume that particular risks are low, and a single one of those assumptions is wrong, it exposes a vulnerability that undermines the security of the entire protocol.

In short, it is difficult (and therefore risky!) to assess the *degree* of a risk. Therefore, given Principle 1 ("very low risk tolerance"), we usually prefer to *eliminate* rather than *reduce* risk if we can. (In this example, we eliminate risk by recommending the use of unbiased casino dice over regular dice.)

Principle 3: Layered security

Glacier will be designed with the assumption that there may be flaws in its design or execution. To minimize the risk of failure due to these flaws, many aspects of the protocol include multiple layers of redundant security.

Principle 4: Simplicity

Secure cold storage is rare in the Bitcoin ecosystem today in part because it is so overwhelmingly time-consuming and takes so much expertise to do it properly. Glacier exists to fix that.

Following the Glacier protocol should therefore be very straightforward. It should be:

- **Self-explanatory:** Avoiding language or concepts which require more than everyday technical skill to understand.

- **Self-contained:** If the user gets stuck and needs to resort to Google or ask someone for advice, we've failed.

- **Directive:** The user should need to make as few choices or judgment calls as possible.

Principle 5: Offer Limited Choice

The question of how to securely store bitcoins is, for the most part, an objective question. Some approaches entail objectively greater risk than others, but for many people, assessing those risks may be overwhelmingly complex. Glacier exists in large part to free users from this burden by offering clear, specific recommendations rather than putting them in the position of having to choose between options.

That said, there *are* limited situations where we believe offering options is appropriate:

- Very different security/cost tradeoffs may be appropriate for different users, whether due to personal preference (e.g. differing risk tolerance or cost sensitivity) or personal circumstance (e.g. someone wanting to secure $100M vs. someone wanting to secure $100,000).

- One option may *not* provide the best possible security, privacy, or experience, but the superior alternative may require additional technical expertise to implement with existing tools (and thus be inappropriate for much of our audience).

- We may be highly uncertain which option is best, even after deep consideration and consultation with other security experts. We think that acknowledging the uncertainty in these cases is helpful as a matter of record for furthering community debate about best security practices.

In these situations, the protocol will provide a default recommendation whenever possible in order to reduce the burden of choice for users. Alternatives will generally be mentioned in footnotes. In cases where even a default recommendation seems inappropriate, the protocol will offer guidance as to which factors should be used to decide between the given alternatives.

Strategic Approach

We have defined security objectives that together prevent the loss or theft of funds. Each objective is supported by one or more tactics.

1. **Objective: Prevent electronic theft or loss of funds**

    a. **Tactic: Isolate all sensitive wallet operations.** Key generation and transaction signing should be done in an environment with as limited means of data extraction as possible. Internet access is out of the question.

    b. Tactic: Don't store sensitive data electronically. Compared to paper, electronic data is more subject to theft by malware (you can't connect paper to the Internet, no matter how hard you try). Electronic storage media are also subject to hardware failure.

    c. **Tactic: Avoid malware infection by using clean environments.** Malware could interfere with the generation or display of sensitive data. Much like biological surgery takes place in a sterile operating room, we can take steps to create computational environments that have a very low chance of malware infection.

2. **Objective: Prevent usage of flawed sensitive data (keys, addresses, transactions, redemption script)**

    a. **Tactic: Avoid malware infection by using clean environments.** Same as 1c above.

    b. **Tactic: Use verified, reliable software.** We select widely-used open-source software with a strong track record of reliability and security, cryptographically verify it against published checksums, and take care to minimize any possible exposure to corrupting malware after verification.

    c. **Tactic: Detect flawed data by duplicating generation in different environments.** The data generation algorithms are deterministic. Given this, if any element of our hardware/software stack is facilitating the generation of flawed data, *and* that element is present in one environment but not the other, then the data generated by each environment will be different. This alerts us to the presence of a problem.

    d. **Tactic: Use trusted, high-quality entropy.** Most sources of entropy have flaws; for example, software algorithms that generate random numbers can be vulnerable to exploitation, if they (either due to algorithmic weakness or other security vulnerability) provide data that is not truly random. We use cryptographic techniques to mitigate these weaknesses.

3. **Objective: Prevent physical theft or loss of keys**

    a. **Tactic: Multisignature security.** Prevents possession of a single key (whether legitimate possession by a trusted signatory, or illegitimate possession due to theft or loss) from enabling theft of funds.

    b. **Tactic: Secure, distributed storage.** Under lock and key, across multiple locations, to further deter against theft or loss.

    c. **Tactic: Physical resiliency.** Use highly durable paper and/or lamination to record the keys.

4. **Objective: Ensure long-term access to necessary tools for accessing funds**

    a. **Tactic: Select widely-used, well-maintained tools.** These are most likely to remain available and suitable for many years to come.

    b. **Tactic: Favor simple technical approaches based on public standards.** Should the specific tools we recommend become unsuitable for Glacier, it will be relatively easy for new tools to incorporate approaches if they are simple and standardized.

    c. **Tactic: Protocol backwards compatibility.** Ensure that future revisions of the Glacier protocol always include instructions for accessing funds stored using all previous versions of the protocol.

Technical Decisions

Key Generation

Use a combination of /dev/random and dice for key generation entropy

/dev/random and casino-grade dice are both high-quality sources of entropy, yet each has potential security risks.

/dev/random is software, and software random number generators can have flaws or vulnerabilities. There has been at least one such attack in the past used to steal bitcoins.

True casino-grade dice (which are necessary to protect against cryptographically-significant dice bias) should, if used properly, provide near-perfect entropy. However, "used properly" is not a safe assumption. For example, if a user reads dice in a non-random way (such as in ascending order) most of the entropy can be lost. Even in the face of clear, strongly worded directions, this is a mistake that, realistically speaking, some users will make. Additionally, there is some risk that dice marketed as "casino-grade" will be nonetheless biased.

We can eliminate any impact of weakness in *one* entropy source by combining *both* entropy sources using a XOR operation. This process is:

- Mathematically sound (proof)
- Trivial to implement
- Verifiable, by running it on two separate secure computers and comparing the output[1]

160 bits of entropy

Bitcoin addresses are 160-bit hashes of a 256-bit public key. Any keypair which has a public key that hashes to an address can spend the coins at that address. Therefore, the most entropy that is useful is 160 bits. This corresponds to 62 dice rolls.

Deterministic key generation

As described in Tactic 2c, deterministic key generation can help us detect the generation of flawed keys.

[1] See Tactic 2c for additional context.

Hardware Environment

No hardware wallets

Hardware wallets such as the Trezor perform Bitcoin operations in relatively simple, highly controlled hardware environments that are unlikely to have malware and for which secure data storage (e.g. of private keys) is a primary design consideration. The software and firmware they run is open source. They are natural candidates for use in Glacier.

The primary security drawback is that today's hardware wallets operate via a physical USB link to a regular computer.[2] While they employ extensive safeguards to prevent any sensitive data from being transmitted over this connection, it's possible that an undiscovered vulnerability could be exploited by malware to steal private keys from the device. Such a vulnerability could be in the wallet's open source software, closed source software (if any), or hardware.

Also, it is nearly impossible to verify that the hardware or software has not been tampered with. An attacker could compromise the manufacturing processes, or ship a malicious device that looks like the hardware wallet you ordered.

We think that hardware wallets such as Trezor, and Ledger, and KeepKey are great for storing a *moderate* amount of funds with *high* security, but for our use case of *large* amounts of funds with *very high* security, we prefer a more robust approach. Although the risks with hardware wallets are quite small, due to Design Principle 2 ("prefer risk elimination over risk reduction"), we opt for a different approach. See the section on "eternally quarantined hardware" below.

"Eternally quarantined" hardware for cryptographic tasks

Quarantined hardware means we take great care to control what information leaves the hardware. Generally speaking, this means we try to *completely eliminate* most channels that might move data -- whether over a network, USB stick, printer, or other means -- because *any* channel might be used by malware to steal private keys.

Eternally quarantined hardware means we use factory-sealed hardware for this purpose (to minimize risk of prior malware infection), and *never* lift the quarantine (since any malware infection which *does*

[2] One could mitigate this risk by making sure to *only* connect the hardware wallet to a computer that is clean of malware, is not connected to a network, and never *will* connect to a network. But then you've just recreated the security that the hardware wallet was intended to provide (you've overshot it, actually), so there's not much of a point in using the hardware wallet`.

occur[3] might wait indefinitely for an opportunity to use an available exfiltration channel). This renders the hardware essentially useless for anything else but executing the protocol.

No printers

Modern printers run sophisticated software, and so could be a vector for malware. They generally come with wireless cards, which may not be easily removable. They also generally store data, including private keys, violating Tactic 3b ("don't store sensitive data electronically").

Use USBs to transfer software to quarantined environments

Using any network connection is clearly off-limits, since the attack surface is unnecessarily large.

The primary reason for using USBs is that they are ubiquitious and easy to use. USBs contain firmware which is subject to malware infection, but the same is true of other media, such as SD cards, and DVDs (which require a DVD drive, presumably connected via USB on any modern laptop, and thus subject to the same firmware infection).

It's worth noting that the Zcash parameter generation ceremony used append-only DVDs to transfer data between systems. This can help provide evidence of tampering, but because we are only transfering known datasets (i.e. specific software distributions), we are able to detect any tampering with checksum verification instead.

Non-persistent memory environments for cryptographic tasks

There are many ways data can get written to disk. In addition to explicit user request, there are logs (application, shell, and OS), caches & memory swap, and so on. Rather than trying to identify and prevent or clean every one of these, it's safer to just operate in an environment that uses non-persistent memory as much as possible.

Our approach is to boot off a USB drive, using an OS that operates entirely within RAM. All application software will be read off a USB drive as well.

[3] The primary malware infection vectors for "eternally quarantined" hardware are quite narrow, and include:

- Firmware infection at the factory
- The USB drives which are plugged into the quarantined computer to boot the OS and run the application software. If firmware-protected USBs are used, and the USB software is cryptographically verified, this vector is very unlikely, although we're unclear if it's possible for malware to hide in the boot sector of the USB.

Software Tools

Ubuntu

Given our previous decisions (particularly “non-persistent memory environments for cryptographic tasks”), Ubuntu is well-suited to our needs in a number of ways:

- Ability to boot off a USB
- Ability to run OS and all applications on a RAM disk
- Open source with a good security track record
- Free

The primary alternative we considered is Tails, a Linux-based OS which has security and privacy as its primary design goals. Given our use of an eternally quarantined hardware environment, however, we already protect against almost all of the risk vectors that Tails does, so there is little added benefit. In addition, the process for creating Tails USB drives is cumbersome, and we were unable to get it to work reliably on all of the laptops we tried.

We also considered macOS, due to Apple’s strong track record with security and user privacy. However, in addition to being closed source, Macs do not (to our knowledge) have a way of physically removing the wireless cards, which prevents the creation of a strongly quarantined environment. It also doesn’t appear that there is a way to boot macOS off a USB such that it only uses a RAM disk, undermining Tactic 1b (“don’t store sensitive data electronically”).

Bitcoin Core

Bitcoin Core meets several key requirements for our wallet software:

1. Allows for user-provided entropy (in the form of an imported WIF private key)
2. Supports multi-sig functionality
3. Offers means to verify software integrity (via developer-provided checksums)
4. Well-reviewed open source code
5. Transparent functionality (RPC console makes clear exactly what operations you’re doing)

Bitcoin Core does have some drawbacks which increase protocol complexity (and therefore increases execution risk and lower adoption):

- The inability to generate a private key from user-provided entropy requires additional preparation work to transform user entropy into a WIF private key
- Lack of support for BIP39 (mnemonic private key generation)
- Working with Bitcoin Core transactions requires modifying JSON data structures

Electrum is a popular GUI wallet program which we investigated. However:

- Electrum doesn’t allow users to provide their own entropy.[4]

[4] It did until recently. The --entropy parameter to the make_seed command line parameter was recently removed.

- There is no automated way to cryptographically verify the integrity of all software dependencies. [5]
- It is difficult to install in the controlled, clean environment we are creating.[6]
- Electrum's multi-signature functionality is oriented around a very different use case, in which generating a multisignature wallet requires generating several normal wallets as well. Signing a multisignature transaction is similarly complicated. The process is cumbersome and confusing, which is not only inconvenient but risky as it increases the chance of error in protocol design or execution.

We also looked at a variety of other wallet applications, including libbitcoin-explorer (bx), Armory, Multibit, bitaddress.org, and many others, and could not find any that met all of the requirements.

GlacierScript

Glacier requires some operations which Bitcoin Core doesn't handle, including:

- Secure combination of multiple entropy sources
- Conversion of user-provided entropy to WIF keys

In addition, the process of using Bitcoin Core for creating multisig addresses and withdrawals is not straightforward, requiring extensive use of the command line API. This complexity not only increases the possibility of user error, but increases the general burden of protocol execution, making the cost/benefit of cold storage less attractive.

GlacierScript is a Python script which automates these tasks. The functionality is designed to be as simple as possible -- aside from storing sensitive data in memory, the only cryptographic work it does is a single XOR. The code is readable and well-commented to facilitate review.

Any program has the possibility for flaws or vulnerabilities. To consider these risks in the context of GlacierScript, it's helpful to think of the alternative not as "no program at all" but "a program with its pieces scattered throughout a PDF which the user will copy-and-paste by hand to a command line." The attack surface of GlacierScript does not seem to be higher than the alternative, and the risk of user error much lower.

[5] We aren't sure that the python packages on which Electrum depends are verifiable via checksums. Python package managers happily install packages where the checksums are missing, so we would need to manually verify each package.

[6] When booting off a Ubuntu USB, Electrum won't be pre-installed, so it needs to be installed after booting, using packages stored on a second USB drive. Creating this USB drive is also cumbersome. apt-get, by default, downloads *and installs* packages. Downloading *and* verifying integrity *without* installing requires a lot of manual work.

Integrity Verification

GPG-verify checksums ourselves and hardcode them in the protocol

From a strict security standpoint, it's not a best practice to ask users to trust the checksums *we* provide for *someone else's* software, for two reasons:

- We lied about doing the GPG verification, or failed to do it properly, AND nobody has double-checked our work to discover the error and raise the alarm.

- Another attacker:
    - Compromised the official distribution of the software being verified, AND
    - Compromised our protocol document to update the hardcoded checksum to match their malicious code, AND
    - Compromised the checksum of our protocol document, AND
    - Compromised the GPG signature of our protocol document checksum, AND
    - Was unable to compromise the officially-distributed checksum and/or GPG signature of the software being verified

These risks are exceptionally small, and the amount of trust required is small compared to the trust required for one to adopt Glacier as a whole. We believe the risks are outweighed by the benefits of making the protocol simpler.

All hardcoded checksums will include convenient links for people who wish to manually perform the GPG verification themselves based on the official developer-provided checksum and signature.

When visually verifying random data, (good) spot checking suffices

In multiple areas, Glacier asks users to verify random data (such as a private key or software checksum) visually. When doing so, we recommends that users check only the first 8 characters, last 8 characters, and a few characters "somewhere in the middle" of a checksum. There are two reasons for this:

- Many people won't check a full string anyway, even if instructed to do so. A specific, less burdensome instruction has a greater chance of being followed.

- This degree of checking is safe:

  An attacker with that's made a malicious code change *could* brute force variations of the code (e.g. with each variant containing a unique comment, like an incrementing number) until they find a fingerprint that's similar **enough** to actual fingerprint to pass most spot checks. So it's dangerous for the spot check to be too sparse -- for example, checking 3 characters at the

beginning and end of a hexadecimal number is a trivial 24 bits of security.

Hexadecimal numbers are the smallest character set we'll be verifying, so we need a guideline robust enough to be secure for hexadecimal numbers. 8 characters at the beginning and 8 at the end is 64 bits of security, plus a few characters at some arbitrary place in the middle brings things to over 100 bits of effective security. As of 2012, a typical computer could calculate ~2^{16} fingerprints per second, at which rate cracking 100 bits would take ~2^{83} seconds, or 10^{17} years. If Moore's Law holds for this operation, that level of computation will be impractical until the latter part of the century.

Correctness Verification Using Duplicate Environments

Use two different environments to generate all sensitive data (keys, addresses, transactions)

Per Tactic 2c, all sensitive data should be generated twice, once on each of two different eternally quarantined computing environments, to verify the data is created correctly.

Use two different environments to create quarantined USBs

We will use USB drives (with cryptographic firmware protection) to move software to the quarantined computing environments. These USBs will be created using existing networked computers, known as "setup computers".

There is a possibility for malware to infect the quarantined computers by propagating from an infected setup computer via the USB drives. Such malware would need to be quite sophisticated, as it would need to somehow forge false positives for the checksum verifications on the USBs -- but it is possible.

To reduce this risk, we will use two different setup computers to create the quarantined USBs for each of the two quarantined computing environments. Unless both setup computers happen to have the same malware infection, this means that the malware can only make it to one of the two quarantined environments. This stops compromise by malware which generates flawed sensitive data (since the flawed data will be detected when the two quarantined environments produce different results).

Both quarantined environments can run the same software stack

The risk of widely-trusted software (such as Ubuntu or Bitcoin Core) generating flawed sensitive data is very small. Indeed, if such a flaw were widespread, the security of Bitcoin would be undermined to such an extent that your bitcoins are unlikely to be valuable enough to be worth securing.

There *is* a very small risk that issues with the software may result in flawed data generation in rare cases only. Such an issue might plausibly go undetected by the public, and if you are unlucky enough to get hit by such a case, the security of your funds would be affected.

We feel this risk is too small to be worth addressing. Although Design Principle 2 ("prefer risk elimination over risk reduction") would suggest we should eliminate this risk by running different software stacks on each computing environment, this would introduce a significant amount of complexity to both the design and execution of the protocol. In addition, although Design Principle 1 is to adopt a very low risk tolerance, that is in part because most small risks will grow significantly with Bitcoin's success, whereas this particular risk would not grow as much.

It *might* be a minor improvement for a future version of Glacier to eliminate the risk by adopting different software stacks for each environment. However, we are not prioritizing this for v1.0.

Both quarantined environments should have different hardware stacks

While Glacier's recommended software is not likely to generate flawed data on its own, it's possible that malware might interfere with either the generation or display of critical data.

Because we're operating in an eternally quarantined environment, the primary malware infection vectors are hardware-related -- e.g. a manufacturer-specific firmware vulnerability, or a flawed USB stick that is susceptible to infection of its firmware.

We can significantly reduce this risk by using hardware from different manufacturers for each secure computing environment. This won't eliminate the risk -- there could be a vulnerability in chips used by multiple laptop manufacturers, for example. But it does eliminate many scenarios (e.g. a vulnerability installed in the firmware of a particular manufacturer's product line).

Transfer data off quarantined hardware through low-bandwidth visual channels (QR codes and manual transcription)

There is a need to move *some* data off the quarantined computer. Individual private keys must be distributed to signatories (along with public keys so multisig redeem scripts can be built), and signed transactions must be transmitted to the Bitcoin network.

As described previously, the quarantine means the options for moving this data are limited. There is no network access and no printer. Nor do we want to transfer data off via USB; in the unlikely event of a malware infection on the quarantined computer, the USB drive may be used as a channel to relay sensitive data to complementary malware on another computer.

For private keys, our solution is to transcribe them from the quarantined computers by hand[7] using BIP-0039 mnemonics. For public keys and signed transactions, our solution is to transfer them off the quarantined computers using QR codes, the content of which can be verified with a manual spot-check.

[7] One alternative to storing the keys on paper is to not store them at all -- i.e. memorize them (a "brain wallet"). Brain wallets have several critical drawbacks, the most significant of which is that they're not well-suited to multisignature security -- you probably can't rely on all of your signatories to memorize a complex mnemonic, nor is there an easy way to verify whether their current recollection is accurate. It's also hard to make a passphrase that is neither too easy to remember (and therefore hackable) nor too hard to remember (and therefore forgettable).

Side Channels

Take simple precautions against side channel attacks

Side channel attacks are not terribly common today, but working implementations exist for a wide variety of attacks, and activity in this space has been increasing over the past few years.

While some side channel attacks require *two* malicious / compromised devices in close physical proximity -- one to transmit data, and one to receive it -- uncompromised devices can still leak data through side channels, requiring only one compromised device to detect that data.

For example, there is a working implementation of an iPhone application that can decipher keystrokes made on a nearby keyboard (*not* on the iPhone) with seismic data (i.e., vibrations detected via the phone's accelerometer). The process has enough accuracy to radically simplify a brute force attack on a private key. And progress is being made about being able to execute the same attack using acoustic data.

This is assuming the quarantined computer is not compromised. But although we are taking many precautions to protect against that, we're also assuming one quarantined environment *could* be compromised -- that's the reason for a duplicate verification environment. And if a quarantined environment is compromised, the opportunity for side channel attacks increases drastically.

Finally, per Design Principle 1, we conservatively assume the sophistication of attacks against Bitcoin, and against Glacier in particular, may increase over time.

For all of these reasons, Glacier includes relatively straightforward precautions against side channel attacks, such as using a table fan to generate white noise and turning off one's phone and putting it in a Faraday bag. This will not protected against sophisticated, targeted side channel attacks (for example, done by a criminal surveillance team with specialized equipment), but such scenarios are beyond the scope of this protocol.

Address Reuse and HD Wallets

Address reuse after withdrawing funds from cold storage is tolerable

This position is counter to what's generally considered best practices, for privacy and security reasons. However, given that we are not using HD wallets in 1.0, address reuse enables us to do a test withdrawal, which is important -- you want to be sure you're not putting your $10M into an unlockable box, especially when the construction of that box is a highly manual process subject to human error.

We think the security drawbacks of address reuse are meaningful but tolerable, and outweighed by the reliability benefits of doing a test withdrawal.

Drawbacks are:

- Some argue that Bitcoin's cryptography is only resistant to quantum computing if addresses are not reused.

- Spending address use publicly publishes (to the blockchain) additional mathematical information related to the private key. While this *theoretically* should not compromise the security of the private key, Bitcoin and its underlying cryptographic algorithms have seen vulnerabilities related to address re-use before (1, 2). Address reuse therefore exposes one to the risk of additional unknown vulnerabilities.

- Spending coins from an address requires posting the address's full public key on the blockchain, which reduces the bits of effective entropy in the private key from 2^{160} to 2^{128}. This by itself is not a significant problem; if Moore's Law continues as-is, it will take about 100 years before a 128-bit key can be brute forced.[8] But it could *become* significant in conjunction with other factors that incrementally weaken security, such as radical technological breakthroughs or newly-discovered cryptographic vulnerabilities.

- Less privacy, but this is not a primary design goal of Glacier. A minor amount of privacy can be created by routing funds through an intermediate transaction and/or coin mixers if desired.

Given that we are not using HD wallets in 1.0, benefits of address reuse include:

- Doing a test withdrawal after setting up multisig keys. The key generation process is complex, involving several sets of private keys, addresses,and redemption scripts, on two different

[8] The fastest supercomputer in the world can do 10^{17} operations per second. Given the (inaccurate but lower-bound) assumption that guessing a private key takes only one operation, it would take this supercomputer $2^{128}/10^{17} = 10^{21}$ seconds to crack a 128-bit key, or about 10^{13} years. For reference, this is 1,000 times older than the age of the universe (10^{10} years).

Moore's Law states that the number of transistors on a chip doubles every two years, which is approximately equivalent to a 10^3 increase every 20 years, or a 10^{15} increase every 100 years. After 100 years, this would bring the time to crack a 128-bit key down from 10^{13} years to a few days.

computers. Information is transcribed by hand and QR codes. There's a lot of possibility for human error, and some possibility for error in design of Glacier as well.

- Enabling partial withdrawals without needing to set up a new secure multisig address (which, at least with the level of security defined in this protocol, is extremely time-consuming).

Address reuse after depositing funds into cold storage is tolerable

Reusing addresses for depositing does not have the same security concerns as reusing addresses for withdrawing. It *does* have some privacy issues, but avoiding re-use is very expensive (it requires us to securely generate and manage a new private key for each deposit, until Glacier is adapted to use HD wallets -- see below).

The privacy impact is relatively minor because the cold storage address is unlikely to be used for a large number of deposits (it will probably be a small number of large deposits). This reduces the impact to privacy, both for the fundholder and others.

Glacier *does* establish some basic privacy by having users route cold storage deposits through an intermediary address that they control. (The alternative, having a third party directly deposit funds into a cold storage address, gives the third party direct visibility into the cold storage account.)

Users may opt to use mixers if they value the incremental privacy more than the incremental risk of trusting a third-party mixing service.

No hierarchical deterministic (BIP-0032) keys in Glacier v1.0

Using HD keys *could* be quite beneficial to Glacier in terms of privacy and ease of use. However, our primary design goal is security, and in the interest of publishing v1.0, we're considering HD keys out of scope for now. To illustrate why, this section will briefly outline the complications involved in integrating HD wallets.

First, let's acknowledge that *not* using HD wallets has significant drawbacks:

- **Withdrawing partial funds from cold storage becomes cumbersome.** Without HD wallets, you either need to reuse addresses (which negatively impacts security) or securely create new multisig change addresses whenever bitcoins are spent. That's a lot of work.

- **Depositing funds in cold storage can impact privacy.** Glacier allows deposit address reuse, in order to avoid the tiresome process of securely creating a new private key for each deposit. However, re-use does have some privacy impact.

Using HD wallets would mitigate these drawbacks, because it would allow us to generate new addresses without having to generate new secure private keys.

Unfortunately, the implementation cost is non-trivial, in part because Bitcoin Core doesn't support importing a raw HD master key (which we need to do, since we are generating our own key based on our own entropy) -- it only supports importing an HD wallet.dat file (Bitcoin Core's native format), which is a binary Berkeley DB file.

Creating a wallet.dat file ourselves would require an undesirable amount of complexity for v1.0, and maybe for the protocol in general, so we will not be using HD wallets for now. We'll advocate for HD wallet support in Bitcoin Core.