Cascade: Pastel Network's Storage Layer

Jeffrey Emanuel

November, 2022

Contents

1 Introduction

In this article, we introduce *Cascade*, the storage layer used in Pastel Network. While the original Pastel Whitepaper from 2018 gives a detailed outline of the basic ideas and motivations for the overall design of the storage layer, the actual implementation details have evolved significantly during the development process as various issues surfaced. While the current design is largely in accordance with our original plan, it does have some important changes which we will describe and explain below. Most of the major differences involve changes we made to allow for greater future scalability of the system, which we primarily achieved by storing less data in the blockchain itself (which is done in the form of actual coin transactions to "artificial" addresses that encode the stored data) and storing more in the storage layer itself. The system we previously were going to use only to store file data evolved to also store metadata about those files— namely, lists of hashes of the various file "chunks" which correspond to a given original file stored by a user.

Despite these changes, the security and reliability of the system were maintained by always including the SHA3-256 hash of additional metadata in the blockchain itself, so that nodes can easily verify data retrieved from Cascade by comparing it to the most secure component of the overall system, the underlying blockchain which is secured through mining. The other significant change that is not covered in the original Pastel Whitepaper is the critical subject of *storage challenges*; this refers to how Pastel's Supernodes[1] verify that other nodes are following the rules of the protocol and are in fact storing all the files they are supposed to be storing.

To make the current article self-contained, we will first give an overview of the basic principles that Cascade is based on, along with the motivation for those principles and how they stem from various security, reliability, and censorship resistance concerns. We will then give a detailed description of how the system is actually implemented in the Pastel software. We will then compare our design to other competing projects, such as IPFS, Arweave, Filecoin, and Siacoin, and attempt to demonstrate why we believe Cascade to be strictly better than these other protocols, particularly when it comes to censorship resistance and decentralization. We will then discuss the subject of storage

[1]A Supernode, henceforth written as SN, represents the heart of Pastel's infrastructure; anyone can create a new SN without permission— the only requirement is that they acquire 5 million PSL coins (to collateralize, or "self stake" their SN) and know how to set up a Linux machine connected to the internet with a static IP address. In return for providing services such as file storage and duplicate image detection, SN operators receive a proportionate share of the SN rewards pool, which is 20 percent of the overall mining block reward.

challenges: why they are needed and how they should be designed to give the best security and reliability properties while minimizing wasteful protocol overhead. Finally, we discuss a road map for future development, covering subjects such as fine-grained access controls and the ability for SN operators to store only content that is not explicit in nature (determined automatically using a deep learning model to generate a "NSFW Score").

2 Philosophy

All complex engineering problems involve a series of trade-offs and compromises. That's because various objectives that you might have can be in conflict, such as:

- Performance
- Reliability
- Security
- Scalability
- Censorship Resistance
- Economic Sustainability
- Efficiency (of network, storage, and compute)
- Robustness (in the face of attacks or accidents)
- Simplicity (for users and developers)

This is particularly applicable to Pastel's storage layer, which must balance all of the above objectives to accomplish the goals we have for the project. In short, our design places the most emphasis on reliability, censorship resistance, and simplicity; we then try to do as well on all the other objectives without sacrificing any of these core ones. We believe that the most important promise we have to the user is that their files will be safe and available indefinitely, and that, other than a relatively painless process at the beginning when the data is stored and the user pays for this storage using PSL coins, they will never again have to interact with Pastel in order to preserve access to those files in perpetuity.

Part of our system is based on technical innovations, such as our use of the RaptorQ encoding scheme, which allows for a nearly optimally efficient, low-overhead protocol; the other part is based on "crypto economic" considerations, where we set up the incentives in such a way that SN operators acting in their own selfish interests will keep the network running reliably, even in the case where other SN operators act maliciously[2].

Our design is always motivated and steered by 3 vitally important principles:

1. Network participants, and particularly SNs, must have as little discretion or ability to choose any of the particular actions that they must take on the network to remain compliant with the protocol and to avoid incurring penalties. This applies across the board, including which network requests each SN is responsible for; which file "chunks" the SN is supposed to store; which other SN peers each SN should issue storage challenges to (and which files should be selected for challenging, or the start and end offsets of those files used to generate the challenge); how responsibility for storing each file chunk should dynamically adjust in response to SNs entering and leaving the network, etc. Instead of allowing SNs to choose anything or using some kind of "random" selection process, we take advantage of the emergent complexity of the proof-of-work mining that secures Pastel's blockchain: by referencing the hash of the previous Pastel block, which everyone can agree is impossible to predict or "game" because it's an artifact of the mining process, we are able to arrive at a provably fair rank order of all SNs in each block by using the XOR-distance algorithm introduced in Kademlia (we will explain this in greater detail below). This approach has a number of advantages, the most important of which is that it is completely distributed and decentralized, without any centralized authority or "tracker" responsible for

[2]There is a tendency in many decentralized designs to rely on "game theoretic" ideas about the optimal behavior of participants in maximizing their own utility; while these might have good performance or efficiency properties, they often accomplish this at the expense of security and censorship resistance. In the design of Pastel, we never take for granted that any network participant will always act in their best interest from a purely financial standpoint; instead, we suppose that even SN operators (who by definition have a significant economic stake in the network, given the need for 5 million PSL to collateralize an SN) might act in "economically irrational" ways (i.e., ways that might result in financial penalties or even getting banned by the network) because they are less interested in profit and more motivated by censorship (the suppression of particular data files by a power state-level actor is the most critical threat model in our view), or are simply trying to vandalize and undermine the network.

coordinating activity. Instead, each SN can independently compute not only what it should itself be doing at any given moment on the network, but also what every *other* SN is supposed to be doing, so they can all fairly check each other, and can even "check the checkers" to verify that everyone on the network is doing the right thing.

2. We must always minimize the probability of a permanent, irrecoverable loss of data; we don't mind too much if individual data file chunks are lost in the short term, because we dramatically over-provision the amount of data we need to reconstruct the original file, and as we will see below, our data chunks are not the normal disjoint segments of data that you might be familiar with from BitTorrent, but instead are seemingly magical objects that have the amazing property of being "fungible". This means that there is no such thing as a "rarest chunk" of data that you might not be able to find from another node; instead, as long as a chunk is one you haven't seen before, it will help you in reconstructing the original file. Furthermore, because each of these chunks is stored on multiple machines, it is less likely that all those machines will go down at the same time. But even if they do, and the network loses access to those particular file chunks in the short term, these missing chunks can always be regenerated exactly based on a seed value that is stored in the corresponding ticket that is stored in the Pastel blockchain. Thus as long as we can reconstruct the original file from the remaining available chunks (which we have far in excess of what is required), we can later do "self-healing" of the storing layer to replace the missing files.

3. Any theoretical attack on the system should only ever apply to the system as a whole, and never to a particular set [3] of targeted files, nodes, transactions, etc. This follows the security properties of proof-of-work mining in general, where the only feasible attack (at least for Bitcoin so far) is a "51% attack"; this means that it's not possible for an attacker to target a particular Bitcoin address or transaction without simultaneously attacking the entire network. This principle is particularly important when it comes to censorship resistance in the face of nation-state level actors, which possess both financial and technical resources. When

[3]A corollary to this principle is that we should also aim to treat all the files stored in Cascade in exactly the same way, with no special treatment for any particular file. Because as soon as the treatment can vary, this can be exploited to lead directly to censorship. One exception to this could be caching of the entire file for efficiency reasons for heavily accessed files, since nodes can easily verify that they have received the correct data by checking the file's hash against the corresponding Pastel blockchain ticket.

it is possible to target a particular file or other entity in the system, it's much harder for independent third-party network participants to determine whether observed behavior on the network is "intentional" or "accidental". For instance, the attacker could be doing an amazingly good job at responding correctly and quickly to all network requests other than the select few that it is interested in suppressing, and still potentially maintain a competitive ranking in the network because its "average" performance is still quite good. To use a military analogy, the worst losses in battle are generally suffered when one side is able to focus its forces so that the other side is outmatched by a large multiple. Thus a good defensive posture is one that avoids placing forces in situations where they are vulnerable to such losses, such as by being outflanked or encircled. By allowing no discretion or choice on the part of network participants, we protect the storage layer with a kind of "phalanx" (in the form of the XOR distance procedure) that prevents individual files from being targeted, just as the soldiers in the inside of the phalanx are better protected from attackers.

With all that being said, before we can decide on the best set of engineering compromises, we must first clarify our primary use case. There are many ways one could use a storage layer, but for Pastel's purposes, we are interested in a system with the following properties:

1. Files are *read-only* once created; this is enforced and verified by checking the hash of the file against the hash in the corresponding blockchain ticket associated with that file.

2. Files will generally be at least 100kb or so, as they will mostly consist of media files such as images. Thus we aren't trying to store billions of tiny 1kb files, which has its own set of challenges.

3. File retrieval latency isn't super critical for our application; most file download requests for a given file will be fairly infrequent, as only the current owner of a file is allowed to request it for download from the network, and presumably, they won't need to download the file all that often. If it takes a couple of minutes to stitch together the file in response to a valid request, that is not such a drawback (certainly it's much less of a problem than potentially losing data permanently.)

4. There will be a "Pareto Principle" in the pattern of file access, whereby a small fraction of the files will generate the majority of requests and

activity on the network. We can optimize our design for this situation through the careful use of caching so that SNs are not forced to reconstruct the file from chunks in order to serve the most "popular" files that have recently been requested by other users; instead they will have a separate storage area for keeping the entire files ready for the most recently requested files.

Obviously, if the ability to modify existing files, or to frequently access huge numbers of tiny files were critical to our application, we would have different priorities and would come up with a very different design. We believe that the most useful and valuable storage service is one that emphasizes data robustness and security the most while being as efficient as possible within those constraints, and that's what we attempted to do in the design of Cascade.

3 Background

3.1 XOR Distance

There is basically one enormously useful "trick" that Pastel uses again and again throughout its design to solve the "decentralized coordination problem"; that is, how can a group of nodes on a network determine what each should do based on the behavior of the other nodes without relying on a centralized coordinator that can assign roles or tasks. At its core, this problem reduces to "how can we order a set of objects (i.e., files, SNs, etc.) in such a way that everyone can agree that it is a fair ordering, with no opportunity for clever trickery or other ways of gaming the system to predict or control the result". That's because, if *anyone* could predict or control the ordering, then they could leverage that power to give them disproportionate control over the network without requiring a correspondingly large investment in resources into the attack, since they would be able to focus all their resources narrowly on the attack.

There is a good analogy to the process by which computers generate random numbers. If the number is the result of a completely deterministic process running on the CPU, how can we really say that it's "random"? One way people have solved this issue is to make computer add-on cards that can sample the random thermal fluctuations and use this to seed a random number generator. The Linux kernel has a quite involved procedure that attempts to "gather entropy" from various sources on the machine, such as the exact timing and values of keyboard presses and network packets, all of which is mixed together to do the equivalent of shaking a pair of dice in your hands to generate a "provably fair" random value. The problem with these approaches is that they are all well and good for convincing *yourself* that the number is random, but how would it help you to convince anyone else? When they run the same code on their machine, they will come up with a different number.

What we need is a way to avoid this issue by having everyone reference a single shared source of "trusted entropy" (i.e., unpredictability). What might this look like? To use an old-fashioned paper analogy, suppose that each day, a group of people wanted to establish a rank ordering of their group so that each member of the group is assigned a number from 1 to N, where N is the total number of members. One way they could accomplish this is to choose a source that is unlikely to be easily corruptible, and use that to somehow encode the ordering. More specifically, suppose the group first prepares some simple alphabetical ordering of the N members, so that each member is associated with a number in an initial ordering (since we will re-order them, this initial sorting order is irrelevant). Then, suppose the group comes up with the following plan: every day, they will each check their paper copy of the New York Times, and starting from top to bottom and left to right, they will take the first letters of the words in the first sentences of each article, to arrive at a string of text, such as `tpbhc`. They continue this process until they have generated a certain number of strings of a pre-determined length. Finally, they could take the list of resulting strings, add each of them to the original alphabetical list of member names, and then sort the combined list alphabetically again (or use some other scheme, such as reverse-alphabetical). Then they could rank each member based on how many of the generated strings they were "close to"—say, within 3 places of the member in the final rank ordering.

Why might the group accept this approach as being fair? Well, for one thing, it's pretty hard to influence the New York Times in that way, where you could force a bunch of individual journalists and editors to modify their first sentences. Also, everyone can easily and securely verify what the New York Times says, by purchasing their own copy from an independent source such as a random newsstand. And as long as each member's name is fixed over time, they can't do anything to improve their chances of being one of the top-ranked members. Reasoning from this example, we can try to find something that shares these key characteristics. It turns out that proof-of-work mining, of the type used by Pastel's blockchain (specifically, the Equihash algorithm used by Zcash, which is minable by both GPUs and ASICs), offers a compelling source of shared, trusted entropy that checks all the boxes.

The process of mining a block on Pastel consists of repeatedly trying new random values (called nonces) which, after being appended to the end of the bundle of transaction data to be included in a new block, results in the block hashing to a small value— one with a large number of leading zeros. There is no "shortcut" way to do this; to mine faster, you simply must be able to try more random values faster to see if any of them result in a small hash value. In a network where there is competition among multiple unrelated miners, such as Pastel, it is difficult enough to solve a block with no constraints at all other than the block hashing to a low enough value relative to the current "mining difficulty"; if you also add on to that the additional requirement that the hash value must match, the difficulty of finding such a solution quickly becomes infeasible (i.e., an attacker would need, say, 1000x as much hash power as everyone else in the network). That is, we can rely on the particular block hashes as being a mere coincidental artifact of the mining process. Just like with the first letters of the sentences from the New York Times, we can safely assume that no one is able to manipulate the particular hash in a way that would advantage them somehow.

And just as we can reliably and easily get a copy of the New York Times that hasn't been doctored, we can very easily get a trusted copy of the blockchain data from the network simply by asking other nodes and comparing their results, and also by validating the information they give us independently. So now we have a reliable source of a string of text (the previous block's hash) that we know was generated in a fair way that we can all agree on. How can we go from this to a rank ordering of all the SNs that they will all agree with? This is the "XOR distance trick" that we keep referring to.

In order to make it work, we must have some form of persistent identity on the network. In Bitcoin, users are advised to constantly generate new addresses to maintain privacy; otherwise, anyone using a blockchain explorer website would be able to see the various other addresses you transacted with over time. But such a system makes a fair ordering impossible; instead, we need users to commit to an identifier and then stick with it over time, where there is a significant cost involved in changing the identifier and a minimum waiting time to prevent "ID churn". This is accomplished in Pastel by means of the PastelID registration tickets which are written by users and SN operators directly to the blockchain in the form of special coin transactions that encode this data[4]. These tickets, which anyone on the network can easily and securely retrieve and decode, specify a public key and other information about an SN.

[4]In short, blockchain tickets on Pastel are stored by sending small amounts of PSL to a large number of "artificial" multi-signature addresses. These addresses are artificial (also called "fake" in the literature) in that no one even knows their corresponding private keys; they are instead used as "vessels" for holding small blobs of the data we want to store, while still validating as properly formatted addresses. Thus we are using these transactions not for their economic value but rather as signaling mechanisms to store arbitrary data.

For our purposes here, the main thing is that we have this fixed text string of their public key (e.g., `jXYxRhqvZzcnTaap7VULxvHfo1TPBpTUvs`) that we can use to refer to each SN for purposes of ranking them or sending messages to them.

Now we can explain the "XOR distance trick": it is a way of defining the concept of "distance" in a clever way so that we can talk about the distance between any two strings. In our case, we will be comparing the PastelID identifier string for each active SN to the hash of the previous block. Whichever SN's PastelID is "closest" to this previous block hash using this definition of distance is the "winner" (we actually choose the top 3 SNs in each block as the winners). While it's not necessary to understand all the implementation details to understand the basic idea used, we present here a rough, intuitive sketch of how this XOR distance calculation works. First, we ensure that the two strings we are comparing are of equal length, by padding the shorter one or trimming the longer one (an alternate approach, which we use, is to first compute the SHA3-256 hash of each string, which ensures that the strings are always the same length). Then we convert the text string into a long binary string of just 0s and 1s; this is quite easy to do, as each character is already encoded as a certain bit pattern at a low level.

We then compare the two resulting binary strings by starting with the leftmost digit, and checking the value from each string; treating the 1 as meaning "True" and the 0 as "false", we then apply the logical rule of the exclusive-or (i.e., either A or B is true, but not both A and B). For example, suppose the first number on the first string is 0 and the first number of the second string is 1; then the result would be 1. But if they were both equal to 0 (or both equal to 1), then the result would be 0. We then repeat this simple procedure for each digit in the strings from left to right. At the end of this process, we will have a new binary string of a length equal to the original strings we compared. Now we take advantage of a clever idea, which is to notice that it's easy to represent integers in binary form and vice versa, so that, given a binary string, we can turn that into a large integer in an obvious, consistent way. Now we simply call this resulting integer "the XOR distance" between the original strings.

If you recall the notion of distance from a math class, any valid distance metric must satisfy 3 basic properties. These are that (1) the distance from any object A to itself should be zero; (2) the distance between objects A and B is always the same as the distance between B and A; and (3), the "triangle inequality", which basically says that the distance between objects A and C is always less than or equal to the distance from A to B plus the distance from B to C (i.e., the shortest path is a straight line). If you think about it for a few minutes, it is clear that the XOR distance described here satisfies these 3 requirements, so it makes sense to talk about it as a sensible measure of distance.

In any case, once we have this distance number for each SN relative to the hash of the previous block, all we need to do is sort this distance in descending order, and the SNs with the smallest distance are deemed the "winners" for the current block (lasting 2.5 minutes on average). This gives us the system we need for every SN to be able to easily check and validate the ordering for the current block. Because we know that it is fair by construction, a SN operator doesn't mind if they lose a few blocks in a row; eventually, the law of averages will kick in and they will be among the lucky SNs picked.

In the previous example, we have focused narrowly on the problem of putting all the SNs into a provably fair order during each block. But in fact, we constantly use this idea throughout the entire design of the storage layer in Pastel. For example, in order to determine which SN is responsible for storing which file "chunk", we simply compute the hash of each file chunk, and then compute the XOR distance between these hashes and all of the SN PastelID identifiers; if an SN is in the list of the 5 closest SNs to that particular file chunk, then the SN is responsible for storing it. This also solves the coordination problem of deciding who should store what on the network, since it's provably fair and impossible to "game": there is nothing a malicious SN operator could do to increase the odds that one of their SNs is selected to store a particular file chunk (and even if they could, they would have to be storing *all* of the many file chunks needed to recover the file, and would have to knock out the other "honest" SNs which are also responsible for storing those file chunks).

Also, because each SN can independently compute which files it is supposed to store, and can also do this for every other SN, it is fairly straightforward to design a system of storage challenges where SNs can quiz each other "at random" with spot checks to see whether they are storing the files they are supposed to be by asking for some kind of proof of storage within a short period of time. But in fact, these spot checks are not actually "random" in the usual sense of the word; rather, they are completely deterministic. In fact, the entire process, from start to finish, is based on the application of the XOR distance trick in different ways. As we will see later, every single decision by every SN is pre-determined based on the protocol together with the previous block hash. This process controls the selection of all the various parameters, such as which SNs are supposed to issue storage challenges in each block, which other SNs should be the recipients of these storage challenges, which file chunks should be the subject of a file challenge, etc.

Thus we see that XOR distance, together with the provably random hash of the previous block in the blockchain, provides us with a convenient and universal way for nodes to prove to each other that they don't have "anything up their sleeve" when proposing any kind of rank ordering or correspondence between various objects in the network, such as files, blocks, and SNs. As we will discuss in more detail later, taking away any potential for choice or discretion by network participants and replacing this with a totally scripted and automated protocol is absolutely critical hardening the system against attackers. While attacks can never be prevented entirely, through good protocol design we can make these attacks provably difficult to pull off without already controlling a majority of the network through mining hash rate and a majority of the SNs.

3.2 RaptorQ Encoding

In the previous section, we explained how the XOR distance idea works to provide us with a secure and efficient means of coordinating the distribution of responsibilities in the network— particularly in determining which SNs are responsible for storing which files. The other major component in the design of a storage layer is to figure out what it is exactly that the nodes should be storing. In the most naive type of implementation, this might be the actual (entire) file itself, with the whole file mirrored by multiple nodes. While this is certainly simple, it is also the riskiest way to store the data. That's because, if all the nodes storing a given file were to leave the network at once— something that would not be such a rare occurrence if the number of nodes storing each file is relatively low (which would need to be the case in order for the network to scale efficiently)— then there is a reasonably high probability for permanent data loss of *some* files. Even if a randomly selected file has on average a fairly low chance of being lost, we want a system that makes it as unlikely as possible to permanently lose *any* complete file.

The next step up in complexity might be to take the original file and split it up into hundreds or even thousands of tiny disjoint (i.e., non-overlapping) chunks, such that you need to collect all of the chunks before you can reconstruct the original file. This is the approach pioneered by the BitTorrent protocol, and it generally works fairly well. A nice property of this approach is that you first compute the hash of each of the disjoint chunks and send that in advance (this is stored in the ".torrent file" itself) to anyone trying to download the file. This allows the downloader to constantly check that each chunk has been downloaded correctly without any corruption. However, this approach has a big drawback, which is that there is no built-in redundancy in the protocol; that is, one has to download *all* of the file chunks correctly to get back the original file, and if any of these chunks are missing in the network for whatever reason, then you won't be able to accurately reconstruct the file (you would get a corrupted version missing some data). Thus there is always the potential for there to be a "rarest" chunk that is only stored by a few of the nodes. This is why it's a common occurrence with "under-seeded" BitTorrent links to get the file to 99% completion but then get stuck because you can't find any nodes that will send you those last few chunks that you need.

Clearly, we need some kind of redundancy built in to avoid the problem just described. But how should we do this? There are lots of approaches we might try. The simplest might be to just store the chunks on more machines, so that the chances of ever completely losing any single chunk are very low. But this is terribly inefficient, since in order to minimize the chance of complete loss, we end up requiring so many nodes to store each file chunk that we quickly run into scalability challenges (each individual node can't be required to store too much of the total data, or it raises the requirements to participate in the network to such a high level that it would make the network centralized). A smart approach might be to make use of so-called "parity files", an approach that was popular years ago for storing files in the Usenet binary newsgroup system. In short, parity files allow you to supplement a set of disjoint chunk files with additional files that can be used to reconstruct missing chunks from the chunks that you do have (together with the parity files). This makes the system a lot more robust in dealing with some number of missing file chunks. The issue with it is that the parity files take up a lot of space. For example, it's typical to allocate 10% of the size of the original file to the parity files. But this only lets you handle 10% of the chunks going missing; if you want protection against 20% of the chunks disappearing, you need to allocate 20% of the space— a large amount of overhead. Still, this has some nice properties in that you can use the parity files to replace *any* of the missing file chunks; this lets us "spread the risk out" so that we can have fewer nodes storing each file chunk while still being protected against permanent loss of data to a reasonable degree.

It turns out that we can do better— much, much better. The basic idea is to use a fountain code to generate our file chunks. In brief, a fountain code has this name because of the analogy to a drinking glass being filled up by a water fountain. If our goal is simply to fill up our glass, we don't care which particular droplets of water go into our glass. We stick the glass into the stream of water, and any droplets are good for us if they aren't already sitting in our glass; when the glass is full, we can remove it from the stream of water and we are done. The reason this works is because all of the droplets are "fungible"; any of them can be replaced with any other one and it doesn't matter, so long as the incremental droplet is new to us (i.e., not already in our glass). Fountain codes offer a way to do something similar with a file: we can take the file, and instead of generating disjoint chunks, where each chunk has no overlap with any other chunk, we can instead create special chunks (which we refer to as *symbol files*) that have this fungibility property. Although the math involved is complicated, you can think of each symbol file as being like a bag filled with randomly sized "shards" of the original file data. Each new symbol file thus gives you important clues on how to reconstruct the entire file. As long as you haven't seen that symbol file before, it will certainly help you in getting closer to being able to reconstruct the original file.

The trick, then, is to initially over-provision our storage to create far more symbol files that would be strictly needed in order to reconstruct the file. In Cascade, a 1mb image file would first be transformed into a collection of 12mb worth of small 50kb symbol files. That way, we can afford to "lose" a huge percentage of the total symbol files at random and we will still be able to get back the original file reliably from the symbol files that are still available. This is already a much better state of affairs, but then the natural question arises: how much will this cost us in efficiency? If we need to download 2mb worth of the 50kb symbol files in order to reconstruct our original 1mb file, then our overall efficiency immediately plummets by 50%. The reason why we have selected RaptorQ as our fountain coding scheme is that it is truly the "latest and greatest" fountain code that achieves almost unbelievable efficiency. That is, the protocol overhead for RaptorQ is so low (indeed, it asymptotically approaches the maximum theoretically possible efficiency) that we only need to download roughly 1mb worth of the symbol files to get to over a 99% probability of reconstructing the original file exactly. And with just 1 additional 50kb symbol file (i.e., 1mb worth of the 50kb symbol files plus one additional 50kb symbol file, or 1.05mb of symbol files in total), we are essentially guaranteed to recover the original file.

This means that we get all of the advantages of such a system with none of the drawbacks. We can handle losing a huge portion of the total symbol files at random, and we don't need to waste a lot of space storing additional parity files that only provide partial protection. Not only that, but by storing each symbol file by some modest number of SNs (in our case, 5), we dramatically reduce the chances of losing any of the individual symbol files (even though we already have far more than we need because of over-provisioning). Furthermore, as we will see later in more detail when we discuss the "self-healing" capabilities of Cascade, even if we do get "unlucky" and all 5 SNs storing a given symbol file leave the network before they can be replaced by other SNs, we can precisely replace any missing symbol files later as long as we have access to the original file, which we can get from using the remaining symbol files that we do have.

It's hard to overstate just how ingenious RaptorQ is. It represents the culmination of a refining process that began years ago with simpler fountain codes such as LT Codes. In each iteration, efficiency has been improved until it essentially achieved the maximum possible efficiency given the constraints of information theory. The system is heavily patented and controlled by the massive company Qualcomm, and RaptorQ is now heavily used in the ubiquitous 5G cell phone radio standard. Luckily, Qualcomm has stated that it is only interested in going after infringing uses of RaptorQ that are in the domain of radio transmission. Because Pastel is an open-source project that uses RaptorQ in a non-commercial way, and also in a very different capacity than what Qualcomm uses it (i.e., we pre-generate more symbol files than we need at the beginning and store them, whereas RaptorQ is generally used in radios in a "streaming" capacity, where fresh symbol files are generated by the sender of the data in real-time as they are sent to the receiver), the risk is quite low that Pastel will be subject to legal challenges for using RaptorQ in this way.

3.3 Existing Work

The utility of decentralized storage has been evident for years, and various projects have been created to solve the problem using different approaches. Some of these are open-source software projects such as IPFS, which are not directly linked to any particular cryptocurrency projects, but are commonly used to store files related to NFTs and other kinds of data. Others, such as Arweave and Filecoin, are part of integrated cryptocurrency projects, and feature native currency tokens which can be used to buy or sell access to storage in various ways. We believe that both classes of existing technologies have fundamental flaws that make them unsuitable for the sort of very long-term, archival quality, decentralized and censorship-resistant storage that Pastel is attempting to provide with Cascade.

To start with IPFS, which is the largest and most popular storage system in common use as of 2022, while the project has various technical merits, it suffers from a fatal flaw, which is that it's a purely technical solution with no economic model for ensuring that there are durable incentives for providers of storage and bandwidth to continue providing those services indefinitely. For example, suppose a new NFT project is launched with a lot of hype and excitement, and IPFS is used for storing the image files (with the metadata stored on Ethereum or other blockchains).

In the beginning, when interest is high, there may be many parties interested in promoting the project and "pinning" the files on IPFS. If buyers of the NFTs are lucky, the creators will have entered into a long-term agreement with some kind of pinning service that can provide good up-time guarantees, so long as the payments are made regularly to cover the hosting costs (most of which seem to be done on AWS and other large cloud services providers, ironically undermining much of the supposed purpose of a decentralized system).

But what happens if the project gets "rugged" and the insiders all abandon it to work on their new scheme (or to return to their old non-crypto jobs!). One by one, the number of systems pinning the files on IPFS would likely drop. Eventually, the credit card hooked up to the 3rd party pinning service provider will expire or get canceled, and then what? When no one is left to care about whether a file stays up, history has shown that the files tend to vanish.

This concept of "link rot" is well known; for example, huge portions of reference links in Wikipedia are broken, and the chances that a URL from 2007 will work (let alone a BitTorrent link from that year, which is more relevant to IPFS) are fairly low if it isn't from a highly stable and well-managed site. By the time the buyers of those NFTs would even realize that the original files were no longer accessible using the links in the NFT metadata, it could be too late, and unless someone has created their own offline backup and is able to restore the pinning, the file would become unavailable to users in the future. What would that imply to the long-term value of the NFT if it devolves into a "dead link"? Obviously, in this scenario, such an NFT is not likely to have much value anyway. And for a famous NFT series like the Bored Apes, the files will be so widely replicated that they will never disappear. But the main point here is that, without an iron-clad, long-term viable economic incentive to continue hosting a file, there are many cases in which stored files will "fall through the cracks" and be lost to time.

Compare this lack of incentives to the Pastel Network model, where SNs must provide hosting and bandwidth (and must respond to storage challenges) in order to maintain their share of the income from the network. In addition to this, the top-ranked SNs in any given block also receive storage fees in PSL that are proportionate to the total amount of data stored in that block. Although only those top-ranked SNs receive the corresponding storage fee income for the block, the actual responsibility for storing each of the resulting chunk files arising from the original file is distributed across all SNs in the network using the XOR distance method.

In addition to the lack of economic incentives, the rise of popular IPFS pinning service providers such as Pinata has introduced a large amount of centralization and created an attractive target for entities that might want to engage in censorship, such as nation-states. This is true not only at the level of the pinning service, but also at the infrastructure level, since Pinata runs almost entirely on Amazon's AWS cloud. When a large portion of the content in a storage layer is hosted by a single entity that is easily identifiable in the physical world— and also subject to any local laws and regulations— it creates a large class of vulnerabilities that are simply not present in a decentralized network.

In the end, decentralization really comes down to what we might call the "telephone test": namely, if someone high up in some real-world powerful organization (such as a government official or technology company CEO) can pick up the phone and call a decision maker at the hosting firm and threaten, cajole, or otherwise induce the host to remove the data or block access to it, then the network fails the test. Because anyone can set up a Pastel SN without permission, this puts Pastel on a long-term path where it can easily pass the telephone test. Just as how you can't censor a transaction out of a Bitcoin block by calling up and strong-arming the Bitcoin miner submitting the block (there is no one to call, since the whole process is pseudonymous, and a block could be discovered by any miner in the network, with new miners able to join at any point without permission), you won't be able to take down a file from Cascade because you wouldn't be able to identify and coerce the various pseudonymous SN operators into cooperating. And even if some could be identified and persuaded to block the file, they would be replaced automatically with other non-compliant SNs.

So what about the other class of existing projects that contain native currency tokens, so they can at least provide the essential long-term incentive structure to have any chance of achieving robust archival storage? We believe that these projects have optimized for the wrong things. They are generally too complicated for users to easily think about and plan for. SiaCoin, for example, requires a complex storage contract to be negotiated with particular hosts, and these agreements are for a limited duration and thus be "topped up" by the user storing data when the existing agreements expire. This introduces a lot of potential risk to "3rd party users" that might be relying on the file being available, but not in a position to influence the storage of the file in any way. For example, people get distracted and move on to new things. What happens if they forget to renew the storage contract and the data is lost? Or what if they decide that it no longer makes financial sense to continue to incur the expenses of keeping the data available? In Pastel's system, users pay a single fee once and the data is stored in perpetuity. This fee is set at a level that is sustainable over the long term.

It's important to again point out that designing a decentralized storage system involves a series of trade-offs. It appears that SiaCoin's designers were more interested in bringing down the cost of storage well below the prices offered by traditional cloud hosting services such as Dropbox, and this influenced how they designed their protocol. We view Pastel's approach as being much more focused on protecting data against permanent loss under adverse scenarios, with pricing and cost competitiveness to centralized alternatives a much less important priority. After all, what good is it to save some money on storage, but then have the file end up lost in 10 years for some reason or another?

That being said, cost is of course an important factor, and we aim to be as efficient as possible in the Cascade protocol to bring down the cost of providing very robust storage. Particularly because the cost of storage in Cascade is priced in terms of the native currency, PSL, we must be careful to avoid price shocks that could occur in either direction as a result of volatility in the price of PSL. We handle this in Cascade (and in the rest of Pastel's design) by setting nominal prices based on the amount of space used to store a file, and then applying a "price deflator factor" to this nominal price to arrive at the final price to the user in PSL terms. In brief, this works by tracking the total mining hash power of the network (something that all nodes can easily and securely do without referencing any external price feed or "oracle") and adjusting the cost of storage based on the proportional increase in hash power. That's because, in a PoW-based project, the mining difficulty over the long term tends to correlate with the value of the coin; as the price of the coin increases, more miners join the network, and existing miners often add capacity or invest in more efficient mining equipment. For example, if the mining hash power were to increase by a factor of 10x, then the final cost to store a 1mb file in PSL terms would decline by 90%. Over time, as the cost of storage declines, we may add additional adjustment factors to the protocol, with the goal of making the cost of storage to end users consistent with the cost to provide that storage, with a reasonable profit margin on top to further incentivize SN operators.

Arweave is perhaps the decentralized storage project that has made the most headway in the NFT space for storing image data. As we will discuss in more detail below, Arweave has made a core component of its design in direct violation of the first of the three principles we explained previously: "network participants must have as little discretion or ability to choose any of the particular actions that they must take on the network to remain compliant with the protocol." Instead, Arweave has relied on game theoretic arguments to justify their decision to *allow node operators to store whichever files and* *blocks they choose to store.*

The reasoning given is that the financial benefits to a node of storing a given file on Arweave are inversely proportionate to how redundant the file is in the Arweave network. That is, files that are "under-hosted" in the network offer high financial returns, which gives nodes an incentive to seek out less redundantly stored files that are at greater risk of data loss and store these themselves. Conversely, if a particular file is excessively popular among nodes, then it will offer below-average financial returns, which might cause the node to replace that file with a "rarer" file for which they can generate a higher income. Thus, nodes should be left to their own devices to choose and optimize their returns with the most efficient allocation of files. While this approach has a certain elegance and is fairly efficient in a storage/bandwidth sense because it relies on distributed incentives to work, it is a total nightmare from a security and censorship-resistance standpoint. We will explain the issues with this approach later, describing a specific scenario that could lead to unchecked *targeted* censorship.

4 Detailed Description

4.1 Overview

We already explained in the background section the two key ideas that serve as the foundation for Cascade: XOR distance and RaptorQ. In this section, we will put all the pieces together and explain in detail how Cascade works for an example file. To keep the math simple, let's again assume that we want to store a 1mb image file as our original file. So that means we need to generate 12mb worth of 50kb RaptorQ symbol files(resulting in 240 symbol files). The raw data for this operation is provided by the user in their request, and this data is signed by the user's PastelID private key to ensure that it hasn't been modified by any of the SNs involved in adding that data to the Pastel storage layer. Those involved SNs are referred to as the *registering SNs*, and they are simply the top 3 highest ranked SNs in the current block (i.e., the "closest" SNs to the previous block hash using XOR distance).

Each of the 3 registering SNs then calls the `raptorq` service that is part of the Cascade software. All 3 SNs supply the same parameters and data to the `raptorq` service and thus all of them generate the same exact list of symbol files. This list of symbol files is compiled into what we call an *inventory* file, which enumerates all the symbol files and includes the SHA3-256 hash of each. These inventory files are treated in a special way compared to the standard RaptorQ symbol files, with 10x the redundancy in the storage layer as compared to a RaptorQ symbol file. Then, the symbol files are distributed to the responsible SNs for each symbol file. This distribution process is of course simply the XOR distance method. The inventory file is also distributed to the responsible SNs, which are the 50 “closest” SNs to the file hash of the inventory file (compared to just the 5 closest used for RaptorQ symbol files).

Once all 3 of the registering SNs determine the same list of SNs that are ultimately responsible for storing each of the newly generated symbol files, they cause the relevant files to be transferred to the responsible SNs. That is essentially the entire process at a high level. But how is this all implemented in concrete terms? We elected a very simple architecture that leaves room for changes in the future.

The most basic implementation question is how the RaptorQ files (and inventory files, containing the lists of symbol file hashes corresponding to an original file) should be saved on disk. While we originally planned to use some kind of high-performance but low-level key-value store such as RocksDB or Badger, we ultimately concluded that SQLite offered the best combination of performance, reliability, and development speed. The next big implementation question is how messages and data should be sent between SNs on the network. This is all done in Cascade using custom GRPC methods which are generated using the Goa Framework for Golang. However, we do have longer-term plans to integrate a more robust and generic messaging layer that can be used instead.

Finally, each SN exposes a relatively simple API through which other SNs can make requests related to the storage layer, such as requesting a symbol file. An important part of Pastel’s design is that end users never directly interact with the storage layer. All requests must instead be routed to the top 3 registering SNs in the current block, which then all verify the requests to ensure they are valid. For example, if a user is requesting to download an original file from Cascade, they must prove that they have access to that file by signing the request with their PastelID private key; then the responding SN can check that this PastelID is a current owner of that file and thus authorized to request it.

Similarly, new files cannot be written to the storage layer without first passing through a strict process; that is, only the current registering SNs are allowed to add new files. Furthermore, each of these new files, in addition to coming from the registering SNs [5]This means all 3 of the registering SNs— if any of them disagree with each other, for any reason, we deem this a failure and

[5](

retry in the next block with a new set of SNs.), must also have a corresponding blockchain ticket, and the files in the storage layer (stored on the machine as binary blobs in an SQLite database) must have hashes which match the hashes in the corresponding inventory file, which must, in turn, match the hash in the corresponding blockchain ticket for the original file.

4.2 Storage Fees

As mentioned in the previous section on existing work, a big issue with systems such as IPFS is creating the right set of economic incentives to ensure that files are stored reliably over the long term. Pastel approaches this issue in two main ways. The first is to require that SNs follow the protocol and store the files they are supposed to store in order for them to remain in good standing in the network and thus continue to receive their share of the block rewards for helping to run the network. The second way is that SNs receive additional *storage fees* (paid in PSL by the user who is storing a file) which are proportionate to the size of the file stored. These fees are paid to the 3 top-ranked SNs that are selected to handle the file storage request for a particular user, despite the fact that these SNs might not ultimately be responsible for ultimately storing the RaptorQ symbol files that are associated with that particular file. Over time, these fees will average out, although certain SNs may get "lucky" and receive more storage fees than the average SN over a shorter time period. Since SNs can't influence when they will be selected as a top-ranked SN, they can't manipulate the system to generate higher than average fees.

Beyond creating another incentive for SNs to store files reliably, storage fees have the further advantage that they are linked to the consumption of individual users, so that users who place higher demands on the resources of the network must pay a commensurately higher amount to defray the cost of provisioning those resources. Obviously, data storage isn't free; in particular, highly redundant storage split across multiple machines that are always connected to the internet is fairly expensive. Thus it is essential that storage fees be high enough to reflect these costs. In a theoretical sense, we want the storage fees for a particular file to approximate the discounted present value of the cost of providing that storage in perpetuity. Below, we outline an example to show what such a calculation might look like.

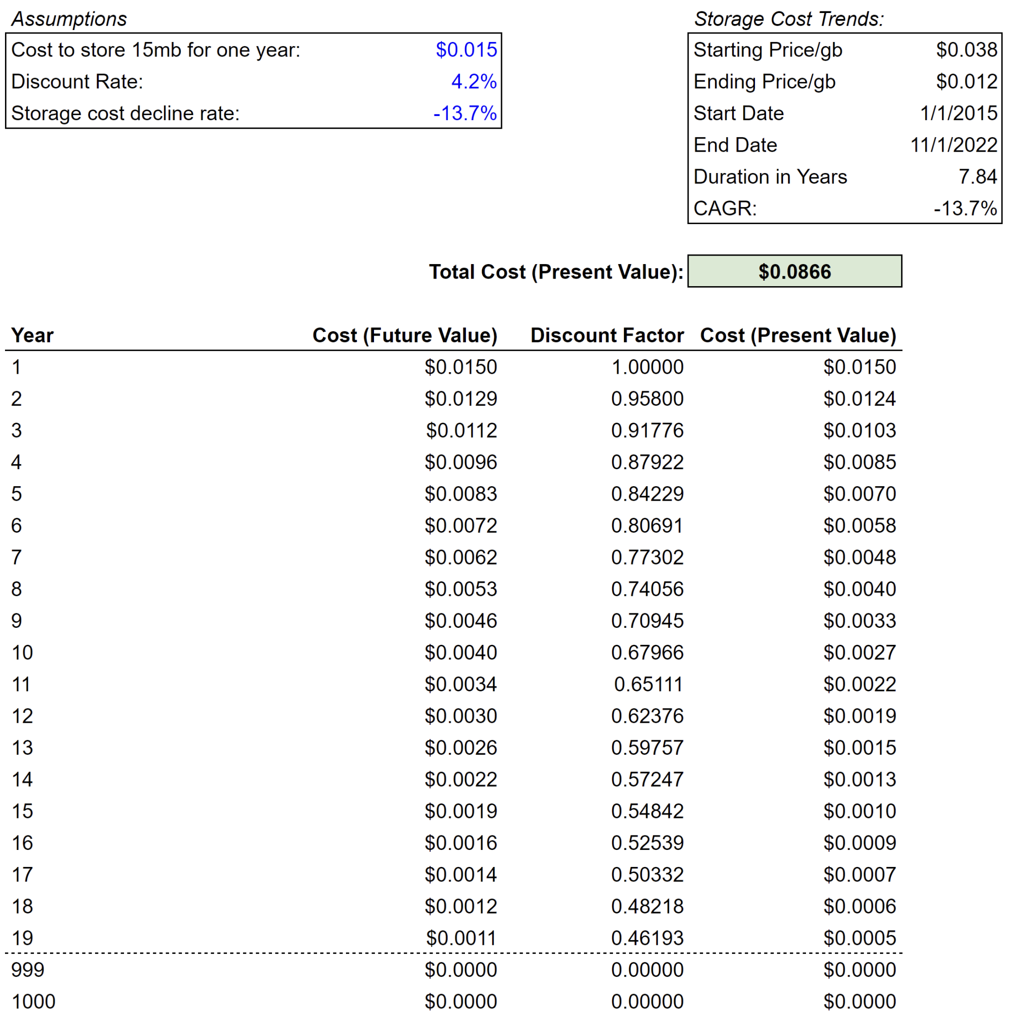
As of 2022, Apple charges $10 per month for 2 terabytes (i.e., 2 million megabytes) of cloud storage. Thus we can estimate that the cost to store, say, a 15mb image file for one year to be (10/2,000,000)*15*12 = $0.0009 per year. But this is not a great estimate to use for our purposes, for a few reasons: one is that Apple is able to purchase storage at incredibly low prices because it is one of the largest buyers in the world. The other is that, although iCloud data is certainly redundantly backed up across multiple data centers, it doesn't require nearly the level of redundancy as a decentralized network like Pastel would require because Apple is centralized and can control whether or not it keeps its servers running. In Pastel, of course, any SN operator is free to disconnect their machine without any prior warning.

So what is the right multiplier to use to get to a reasonable cost for Pastel? One approach is to see how much total space would be required across all the SNs on the network to store our illustrative 15mb file. First, that 15mb file would turn into 150mb worth of RaptorQ symbol files; then, each of those symbol files would be stored by 5 different machines, bringing the total to 750mb, or a factor of 50x higher than the original file size. But we should give Apple some credit for also storing the file redundantly; if we assume that Apple stores 3 copies of every file, then the effective increase in redundancy is more like (50/3) = 16.6x.

Using Apple's pricing, that would work out to \$0.0009*(50/3) = \$0.015 (one and a half cents) per year for the 15mb file. Of course, we are not storing the file for just a single year. Indeed, this is a key point of differentiation for Cascade versus other storage networks such as SiaCoin that require a set term negotiated in advance— the user can pay a single, known amount up-front and then never has to worry about it again, and can rely on their file remaining available "in perpetuity". It's relatively simple to estimate the present value of this cost in perpetuity using a spreadsheet, as shown in the figure below. If we assume that storage costs are flat over time to be conservative, and use a 4.2% discount rate (the current 30-year treasury bond rate as of November 2022), we come up with a present value of approximately \$0.36 to store the 15mb file "forever". But if we suppose more realistically that storage costs will decline at their historical decline rate of 13.7% per year since 2015 (Source: BackBlaze, Diskprices.com) as a result of greater data density from more advanced technologies, then the present value goes down to just \$0.087. Thus, for the file storage system to be economically sustainable for SNs purely from the storage fees, we must effectively charge the user at least 9 cents to store a 15mb file. However, since the SN operator also derives income from simply running an SN separately from any storage fees, the price can be a bit below this without significantly impacting the incentive structure.

Now, it's not enough to charge users a simple variable ("per megabyte") fee. There is also a degree of fixed overhead associated with any file storage operation in Cascade in terms of network and compute. While this fixed overhead is negligible when it is amortized over a "large" file (say, over 10mb),

Figure 1: Spreadsheet for estimating the cost of providing storage services in Cascade in perpetuity.



Assumptions

Cost to store 15mb for one year:	$0.015
Discount Rate:	4.2%
Storage cost decline rate:	-13.7%

Storage Cost Trends:

Starting Price/gb	$0.038
Ending Price/gb	$0.012
Start Date	1/1/2015
End Date	11/1/2022
Duration in Years	7.84
CAGR:	-13.7%

Total Cost (Present Value): $0.0866

Year	Cost (Future Value)	Discount Factor	Cost (Present Value)
1	$0.0150	1.00000	$0.0150
2	$0.0129	0.95800	$0.0124
3	$0.0112	0.91776	$0.0103
4	$0.0096	0.87922	$0.0085
5	$0.0083	0.84229	$0.0070
6	$0.0072	0.80691	$0.0058
7	$0.0062	0.77302	$0.0048
8	$0.0053	0.74056	$0.0040
9	$0.0046	0.70945	$0.0033
10	$0.0040	0.67966	$0.0027
11	$0.0034	0.65111	$0.0022
12	$0.0030	0.62376	$0.0019
13	$0.0026	0.59757	$0.0015
14	$0.0022	0.57247	$0.0013
15	$0.0019	0.54842	$0.0010
16	$0.0016	0.52539	$0.0009
17	$0.0014	0.50332	$0.0007
18	$0.0012	0.48218	$0.0006
19	$0.0011	0.46193	$0.0005
999	$0.0000	0.00000	$0.0000
1000	$0.0000	0.00000	$0.0000

it starts to be an important consideration when the file size is much smaller (say, 10kb). This has important security implications, since a malicious actor could flood the network with requests to store tiny files in Cascade and potentially prevent legitimate users from effectively using the network— all without incurring large costs. In order to deal with this, we set a "minimum file size" of 1mb. While users are welcome to store files that are smaller than 1mb, they will need to pay the same fee as they would pay to store a 1mb file. This increases the costs of an attack while still leaving the system affordable for valid users.

With all of this taken into consideration, the initial fee schedule for Cascade is simple: 50 PSL per megabyte (with a minimum of 50 PSL). At the average price of PSL during October of 2022 of $0.0001 (i.e., one-tenth of a penny), this works out to be around $0.0075 (7.5 cents) for a 15mb file. Because the storage fees are denominated and paid in PSL coins, which have a volatile exchange rate relative to the dollar, we also need some way of ensuring that the fees won't diverge too far from the target level over time as the price of PSL fluctuates. We address this issue in the next section, where we introduce the *Fee Deflator Factor*.

4.3 Fee Deflator Factor

If we want to keep the cost of storing files in Cascade reasonable over time, so that it never becomes excessively expensive relative to other options, we need a way to adjust the PSL-denominated fees to account for potential large increases in the price of PSL. Although anyone can look up the price of PSL on sites such as , this is not an option for a decentralized, trustless system like Pastel. Sure, you could have the SNs separately check the price on various sites and compare the results, but it would introduce a variety of risks. The problem is that anything not internal to the state of the blockchain itself is not reliably observable. The source itself could be corrupted— say, a bug or hack on coinmarketcap.com, or it could be misreported by malicious SNs.

Because Pastel is a proof-of-work based blockchain, it turns out that there is a natural and secure way we can get a fairly reliable estimate of the value of PSL over time, assuming that we take a longer-term perspective. The total amount spent on mining for a proof-of-work chain in a competitive market is generally related to the value of the mined coins if they are immediately sold on an exchange. This amount is closely related to the hash power of the overall network, although this can be distorted in certain cases, such as when Ethereum completed the merge and created a lot of "stranded" mining equipment that switched to other projects using the same hash function, such as Ethereum Classic. But over the long term, and in general, the hash power of the network tends to move in response to changes in the value of the coin. We can use this by measuring the *difficulty* level (essentially, the number of leading zeros we require of the block hash), which automatically adjusts periodically based on the recent block times— if the blocks are being added faster than the targeted 2.5 minutes per block, then the difficulty goes up.

By following the difficulty rate over time, we can construct what we call the *fee deflator factor*, and multiply it by the originally calculated fee (50 PSL per mb). For example, suppose the price of PSL increases 5x from the current price. Then we would expect the difficulty rate to also increase by around 5x. In that case, the fee deflator factor would be 1/5 = 0.2; thus the adjusted storage fee would be 50*0.2 = 10 PSL per mb. Thus we can keep the price of storage on Cascade to the end user somewhat stable in dollar terms despite very large changes in the price of PSL. Most importantly, we can do this in a totally secure and reliable way by observing an internal value that is built directly into the blockchain itself— we don't need to trust any entities to be reporting the value correctly.

In order for this scheme to work in practice, we need a baseline period. Pastel selected a 10,000 block period beginning on block 150,000, and we took the average difficulty during this period, which was representative of recent periods in terms of network hashrate. We then compute a rolling average going forward of the difficulty over the preceding 10,000 blocks and divide this by the baseline difficulty to arrive at the fee deflator factor. This gives us a smoother, more stable adjustment that is more predictable by users.

4.4 Storage Challenges

We have alluded several times already to the central role that storage challenges play in Cascade. In this section, we will flesh out the specifics of how storage challenges are implemented in Pastel and see why they are so essential to the reliability and security of the system. The concepts motivating storage challenges apply to many parts of Pastel's design:

1. Trust no one on the network except yourself; Assume that other network participants may be unreliable or even malicious— particularly when there is a financial incentive for bad behavior (i.e., if we didn't have storage challenges in Cascade, "bad" SN operators could claim to be storing files in fulfillment of their obligations to the network, but then not actually store them to save money on hosting costs).

2. Always enlist multiple "fairly selected" SNs to perform any task; then, have them compare their results with each other for consistency. All messages between SNs are signed using the corresponding private key associated with the PastelID of the SN.

3. "Check the checkers"; to avoid situations where an SN operator might have a financial incentive to falsely accuse another SN of failing a challenge (since the SN share of the mining block reward is fixed, it's a "zero-sum game"), we should verify challenge failures by having other

independent SNs check all the data associated with a challenge. This depends heavily on the next point.

4. Do everything "out in the open"; instead of SNs sending each other direct messages to issue and respond to storage challenges, we instead have all SNs broadcast their challenges to the whole network, so that every node is aware of what is happening. One benefit of this approach is that we can easily implement a measure of "cheat prevention" by having other SNs that also store the same chunk file that a given SN is being challenged on automatically withhold that chunk for the duration of the challenge (similar to how a schoolteacher might instruct students to be careful not to allow other students to see their test paper).

5. As we keep stressing throughout our design, do not allow any choice or discretion at any stage in the entire process: every aspect of the storage challenges, down to the most specific details on each challenge, such as which SNs should be selected as the challenge issuer and challenge recipient in each block, or which files should be challenges be issued for, etc, must be explicitly proscribed by the protocol, where the XOR distance trick and the entropy from the mining proof-of-work keeps everything secure.

We now describe how storage challenges are implemented in Pastel, and how this design encapsulates the key concepts listed above. Think of the storage challenge process restarting for each new block in the blockchain. Then the first step is in selecting the "who" of the challenges: which SNs should be selected to issue new storage challenges during this new block? We again apply the XOR distance trick, comparing the PastelID public key of each SN to the preceding block hash and rank ordering them. We can then take the top-ranked one-third of the SNs and designate these as the *Storage Challengers* for the current block.

As a general efficiency measure, it doesn't make sense to have SNs challenge other SNs on files that the challenging SN is not already responsible for storing. That's because, in order to issue the storage challenge, the challenger would first need to request the file from another SN that has it, wasting a huge amount of bandwidth and nullifying the major benefit of the clever way in which we construct challenges to minimize bandwidth while maintaining security (we will explain this in more detail below). Thus, we require Storage Challenger SNs to stick to the files they are already storing when generating new challenges. The next problems we need to solve are how the challenger should select the specific files it should issue challenges on, and which other SN should receive that challenge (obviously, the challenge recipient SN is one that is also supposed to be storing the file in question according to the protocol).

But before we do that, we can first simplify things by generalizing our XOR distance technique slightly. We do this by introducing the idea of a *comparison string*, which we can use as one "leg" in the XOR distance calculation. In the previous examples of the XOR distance method we reviewed, we were always comparing a single attribute (e.g., "SN PastelID PubKey") with another single attribute (e.g., "preceding block hash"). But if we can glue together multiple attributes into a single string, we can add additional levels of "deterministic entropy" in a simple way. For example, to select which files the Challenger SN should pick, we can create a comparison string consisting of the preceding block hash concatenated with the PastelID of the Challenger SN. We can then compute the hash of this comparison string and use it for one leg of the XOR distance calculation, with the other leg being the file hash of each of the files that the Challenger SN is storing locally. The Challenger can then sort all these and select the N files that are "closest" to the hash of the comparison string, where N is the number of storage challenges that should be issued by each Challenger SN in each block. This way, each SN will select different files based on the different PastelIDs.

We can apply this idea again to choose which of the other SNs that are supposed to be storing the particular file being challenged should be selected to receive a given challenge. This time we make a more complex comparison string that integrates more distinct pieces of information: we first take the preceding block hash and concatenate it with the hash of the file on which we are challenging. We also concatenate the PastelID of the challenging SN. Finally, since this code iterates over the set of SNs in a standardized sort order, we also concatenate the index number, which yields the final comparison string. We then find the "closest" SN to the final comparison string among the list of SNs that are also supposed to be storing that particular file, and this SN then receives the storage challenge. One nice thing about this system is that any SN can easily verify and independently compute this same result in a trustless manner using publicly available data on the network, which means that each SN can check that all the other SNs are following the protocol and issuing the right challenges on the right files to the right recipients (and then also that these are responded to in time with the correct response).

At this point, we have determined most of the challenge details— which SNs should issue challenges in the current block, which files they should issue challenges for, and which SNs should receive those challenges. The final step is to determine what we refer to as the *byte offsets* for each challenge. Recall that the RaptorQ symbol files are 50kb in size, or 50,000 bytes. Instead of asking the challenge recipient to send over the full file in order to prove that they are storing it correctly, we can save a huge amount of bandwidth by instead asking a question with a short answer that the challenge recipient could only answer correctly if they had access to that RaptorQ symbol file locally. A nice way to do this is to somehow generate two integer values between 0 and 50,000; we can take the smaller of the two integers and use it as the starting offset, and the larger of the two becomes the ending offset.

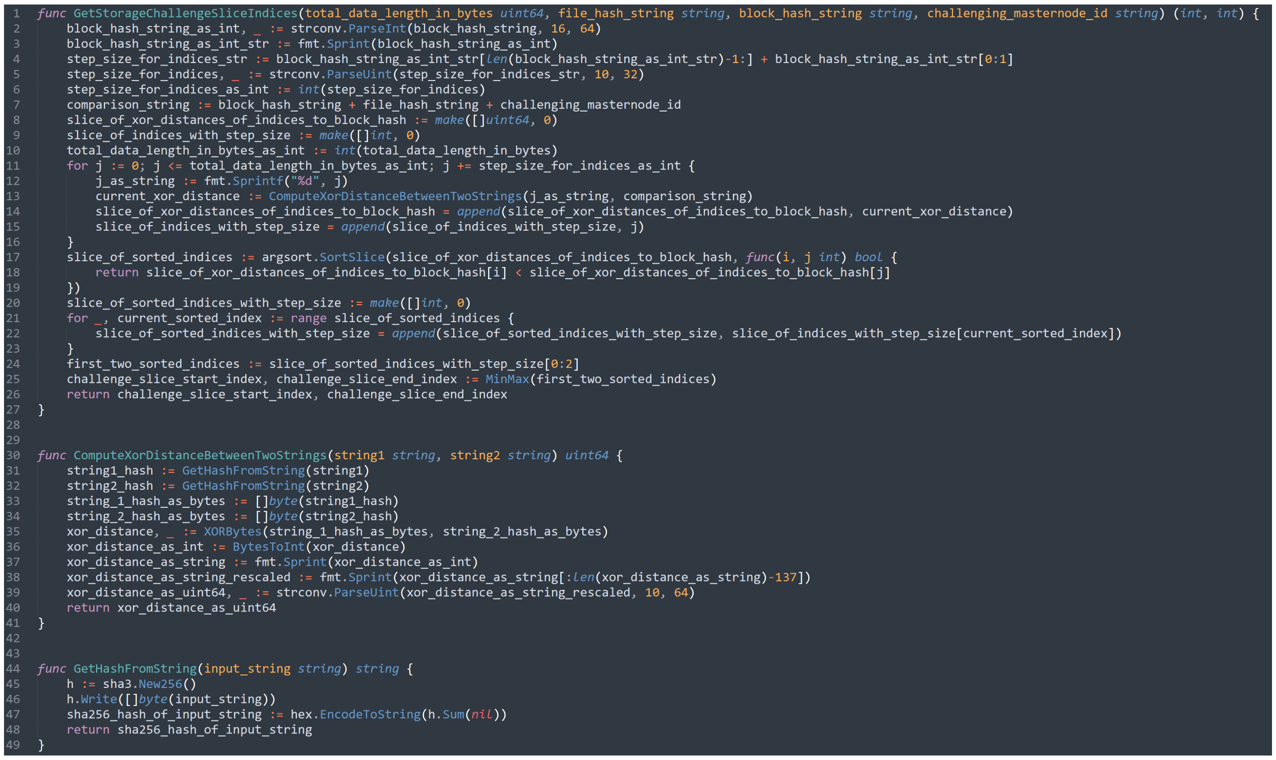
For example, suppose we somehow generate the numbers 5,014 and 31,059 as our values. We include these values along with the rest of the details of the storage challenge; to answer the challenge correctly, the responding SN must then take the subset of data in that particular RaptorQ symbol file between bytes 5,014 and 31,059 and computes the SHA3-256 hash of this subset. This hash is very small, at only 32 bytes— 99.999% smaller than the corresponding symbol file (i.e., 1 - (32/50000) = 0.99936)— and so the responding SN can prove that it had access to the whole file without using much bandwidth. It's essential that the start and end offsets be "random" from the perspective of the challenge recipient, because otherwise, they could prepare by computing a bunch of subset hashes and storing the hashes instead of the actual file, saving a lot of storage space. But because there are so many possible hashes that could be produced using our approach, it would be easier to simply store the actual file and compute the subset hash as required than it would be to try to cheat the system by caching a huge number of pre-computed hashes and discarding the actual file.

In the preceding paragraph, we skipped over how to actually generate the start and end offsets. So how do we do this? The process is similar in that we first construct a comparison string by concatenating the previous block hash, the file hash, and the challenging SN's PastelID. After that the process is a bit more involved, but still not too conceptually difficult: we iterate over the number of bytes in the file. That is, we loop from 1 to 50,000, and for each loop iteration— say, 12,412, we express that number as a string and then compute the XOR distance from that string to the comparison string we just prepared. We then take the set of computed distances and choose the 2 loop iterations that had the smallest XOR distance values. The smaller of these is used as the start offset, and the larger as the end offset.

Of course, this is quite inefficient because we need to compute so many XOR distances in order to do this calculation. In order to enhance performance, instead of counting up by 1 each time, we can skip every K indices. That is, we could start with index 7 and then go to 14, 21, etc. until we are about to cross over 50,000, the maximum value. This means that we can do the calculation K times faster than if we tried every index. So how do we arrive at the value of K to use? We can generate a two-digit integer in the following way: the first digit is the last number in the previous block hash (where we interpret this block hash as an integer, similar to how the XOR distance computation works), and the second digit is the first number in this "previous block hash as a number". This means that we don't need to use a fixed K for every challenge, but instead, this can dynamically adjust for every challenge based on the particulars of the challenge (i.e., on the preceding block hash, the file hash, and the challenger's PastelID).

To make this more concrete, the figure below shows an implementation of a function in the Golang programming language that performs the procedure just outlined for computing the start and end byte offsets. The function makes use of two helper functions that we define below the main function, called *GetHashFromString* and *ComputeXorDistanceBetweenTwoStrings*.

Figure 2: Code for generating the byte offsets used in storage challenges.



```go
func GetStorageChallengeSliceIndices(total_data_length_in_bytes uint64, file_hash_string string, block_hash_string string, challenging_masternode_id string) (int, int) {
    block_hash_string_as_int, _ := strconv.ParseInt(block_hash_string, 16, 64)
    block_hash_string_as_int_str := fmt.Sprint(block_hash_string_as_int)
    step_size_for_indices_str := block_hash_string_as_int_str[len(block_hash_string_as_int_str)-1:] + block_hash_string_as_int_str[0:1]
    step_size_for_indices, _ := strconv.ParseUint(step_size_for_indices_str, 10, 32)
    step_size_for_indices_as_int := int(step_size_for_indices)
    comparison_string := block_hash_string + file_hash_string + challenging_masternode_id
    slice_of_xor_distances_of_indices_to_block_hash := make([]uint64, 0)
    slice_of_indices_with_step_size := make([]int, 0)
    total_data_length_in_bytes_as_int := int(total_data_length_in_bytes)
    for j := 0; j <= total_data_length_in_bytes_as_int; j += step_size_for_indices_as_int {
        j_as_string := fmt.Sprintf("%d", j)
        current_xor_distance := ComputeXorDistanceBetweenTwoStrings(j_as_string, comparison_string)
        slice_of_xor_distances_of_indices_to_block_hash = append(slice_of_xor_distances_of_indices_to_block_hash, current_xor_distance)
        slice_of_indices_with_step_size = append(slice_of_indices_with_step_size, j)
    }
    slice_of_sorted_indices := argsort.SortSlice(slice_of_xor_distances_of_indices_to_block_hash, func(i, j int) bool {
        return slice_of_xor_distances_of_indices_to_block_hash[i] < slice_of_xor_distances_of_indices_to_block_hash[j]
    })
    slice_of_sorted_indices_with_step_size := make([]int, 0)
    for _, current_sorted_index := range slice_of_sorted_indices {
        slice_of_sorted_indices_with_step_size = append(slice_of_sorted_indices_with_step_size, slice_of_indices_with_step_size[current_sorted_index])
    }
    first_two_sorted_indices := slice_of_sorted_indices_with_step_size[0:2]
    challenge_slice_start_index, challenge_slice_end_index := MinMax(first_two_sorted_indices)
    return challenge_slice_start_index, challenge_slice_end_index
}


func ComputeXorDistanceBetweenTwoStrings(string1 string, string2 string) uint64 {
    string1_hash := GetHashFromString(string1)
    string2_hash := GetHashFromString(string2)
    string_1_hash_as_bytes := []byte(string1_hash)
    string_2_hash_as_bytes := []byte(string2_hash)
    xor_distance, _ := XORBytes(string_1_hash_as_bytes, string_2_hash_as_bytes)
    xor_distance_as_int := BytesToInt(xor_distance)
    xor_distance_as_string := fmt.Sprint(xor_distance_as_int)
    xor_distance_as_string_rescaled := fmt.Sprint(xor_distance_as_string[:len(xor_distance_as_string)-137])
    xor_distance_as_uint64, _ := strconv.ParseUint(xor_distance_as_string_rescaled, 10, 64)
    return xor_distance_as_uint64
}


func GetHashFromString(input_string string) string {
    h := sha3.New256()
    h.Write([]byte(input_string))
    sha256_hash_of_input_string := hex.EncodeToString(h.Sum(nil))
    return sha256_hash_of_input_string
}
```

At this point, we have described the major components of the storage challenges in Cascade. The most important thing to keep in mind is that this entire process is transparent on the network: all actions taken by SNs are broadcasted to the rest of the network in the form of messages that detail all the particulars of each challenge. These can all be verified by the other SNs in the network to ensure that everyone is following the protocol correctly. The result of all of this is that we get a robust mechanism for constantly verifying and validating the contents of the storage layer and for ensuring that SN operators either follow the rules or bear the consequences of violating the protocol, which include financial penalties (i.e., the withholding of violating SN's share of the mining block rewards for a specified period of blocks) and eventual banishment from the network.

4.5 Self-Healing

In the previous section about RaptorQ, we explained how, when a file is added to Cascade, we generate many more symbol files than would be strictly required to reconstruct that file, and then further reduce risk by requiring each of the symbol files to be stored by the 5 SNs that are closest to it using XOR distance. But what happens if we get unlucky, and all 5 of the SNs that are storing a set of symbol files suddenly leave the network at the same time, so that there is no chance to conduct a "handoff"[6] of responsibility to other SNs that remain in the network?

This is where another special property of RaptorQ comes into play: there is a specific "seed" value that we can store in the Pastel blockchain itself in the form of coin transactions. This seed allows us to, after the fact, use the complete copy of the file (which we can verify by comparing it to the file's hash, which is also stored directly in the Pastel blockchain) to reconstruct any of the files from the list of RaptorQ symbol files that we originally created. This means that we never truly lose any of the symbol files "forever" so long we can always reconstruct the original file correctly; if we can do that, we can get back any that we are missing. Once we regenerate the missing symbol files, they are "distributed" to the network of SNs in the same way as all other symbol files: the 5 closest SNs to each symbol file must store each of the regenerated files.

Based on the above, the path to implementing "self-healing" in Cascade is fairly clear. **While the self-healing functionality is not yet implemented in the Pastel software, we believe this will not prove difficult.** But our description still leaves out important details, such as which SNs should actually perform the self-healing function? Which files should they target for verification? We can flesh out some of these details below by using the

[6]Under most common scenarios, SN attrition will be infrequent and low enough that there will be plenty of time for the "next closest" SN to get a copy of the symbol file.

same ideas and approaches used in the design of Pastel's storage challenges. For example, in any given block, a certain percentage of all SNs could be designated as "file reconstruction workers"; which SNs? The ones that are closest to the previous block hash for example.

Then, which files should SNs choose to check for completeness? Note that here we are talking about *original* files, not RaptorQ symbol files. We know the hash of any of these original files, since this information is included in the corresponding blockchain ticket for that file. So each SN could select **K** original files to verify and potentially reconstruct if necessary. One way to select these K files would be to use a comparison string consisting of each SN's PastelID public key concatenated with the previous block's hash; then the SN could choose the K "closest" original files, defined using the XOR distance between the comparison string and each original file hash. Now the "file reconstruction worker" (henceforth, "the worker") SN must go through each of these K original files. For each one, it must first look up the corresponding blockchain ticket to get the required information to look up the RaptorQ symbol file "inventory" file in Cascade. It must then request this inventory file from one of the many[7] SNs that store it. The inventory file contains the list of all the RaptorQ symbol files that we originally created for that original file, including the hash of each of these symbol files.

Finally, the worker must issue some kind of storage challenge for each of these symbol files and verify that all symbol files listed in the inventory file can be successfully retrieved. Now, we can't use the exact same scheme as in normal storage challenges, because that relies on the challenger having access to the file in advance. Instead, the only way to do it is to have the challenge recipient send over the actual file, which must have the correct size (50kb for RaptorQ symbol files) and the correct SHA3-256 file hash. We can think of this as a "brute force" storage challenge.

But what if one of these referenced symbol files can't be retrieved? That is, none of the responsible SNs correctly respond to a brute force storage challenge for a particular symbol file during the course of this verification process. Now, there are multiple reasons why a single SN might fail a challenge like this for ancillary reasons, such as a poor internet connection. But the chance of all of the responsible SNs failing at the same time is much lower, since these are effectively selected from the network in a "random" way. Thus, a likely cause for this scenario would be that the RaptorQ symbol file in question has been "lost" temporarily by the network and must be replaced so that

[7]Inventory files are stored at a much higher level of redundancy than RaptorQ symbol files in Cascade— 50 nodes are responsible for each inventory file.

the responsible SNs can get copies of the missing symbol file. We deem this condition, where all responsible SNs fail the brute force storage challenge, as grounds for starting a “file reconstruction process”.

The worker then requests enough of the symbol files corresponding to that original file until it can locally reconstruct the original file (which it can verify by comparing the hash to the corresponding blockchain ticket for that file). Then, using the seed value for that original file (also contained in the blockchain ticket), the worker generates a stream of RQ symbol files, discarding unneeded ones until it generates a symbol file with a file hash that matches one of the “lost” symbol file hashes. Finally, the worker sends the reconstructed symbol file to the 5 closest SNs to it in the network using XOR distance, just as with any other symbol file.

For security reasons, the network can’t simply “trust” that the worker has performed its task correctly. Rather, some portion of SNs (selected using XOR distance to the previous block hash) would be designated as reconstruction verifiers, and independently check that every part of the file reconstruction process was done “on the level”. That is, they could verify that in fact all of the responsible SNs did fail a brute force storage challenge at the specified time (since this entire self-healing process would be conducted using messages broadcast across the whole network to all SNs), and could also check that the purported replacement symbol file provided by the worker did match the right file hash as specified in the corresponding inventory file (thus, each verifier would also have to request this inventory file from the storage layer). After the verifiers finish their work, they sign the results using their PastelID private keys and share the signatures with the whole network for verification by all SNs.

We must be careful in balancing the power of positive financial reinforcement for “pro-social” behavior on the network— things like successfully reconstructing lost symbol files in Cascade— with the security risks that this can introduce by creating perverse incentives for malicious behavior. For example, suppose that, whenever an SN completes a correct file reconstruction process that is verified by the network, then they are awarded a meaningful number of PSL coins as an incentive, since they have demonstrably helped the network to be more robust. But that only looks at the benefits and not the potential costs. The real issue is what is known as the “cobra effect”, which references a famous story from India under British Colonial rule, where the government offered a bounty for each dead snake in an effort to rid the country of this dangerous scourge. While at first very successful in reducing the number of snakes, eventually many individuals began breeding snakes for the sole purpose of collecting the bounty, leading to more snakes than ever and a lot of wasted money.

In Pastel terms, we wouldn't want a malicious SN to make super-normal returns by DDOS attacking all the SNs responsible for a given symbol file in Cascade at the exact moment that they are being challenged. Instead, we simply require that all SNs do their share of file reconstruction duties in order to continue qualifying to receive their share of the mining block reward— just as how they must also perform their storage challenge responsibilities.

5 Security and Censorship Resistance

5.1 A Comparison of Censorship Resistance

Suppose that there are a total of 100 SNs, and that an attacker controls 10 of them. Assume that the attacker wants to disrupt the network's operations, so that files can't be reliably stored and served by users, but that they don't even have a profit incentive in doing so. That is, they won't make any more PSL by their actions (say, by causing other SNs to receive less, as in the example above), but are acting maliciously purely out of "spite". This isn't so hard to believe; imagine that someone has stored a file that is objectionable to the Chinese government, which wants to suppress it. Or even that a competing crypto project to Pastel wants to hurt us to beat the competition.

Because our hypothetical attacker only controls 10% of the SNs, given the various controls and procedures described in the previous section, they have virtually no chance of being able to really hurt the network. Because if a user requests access to a file, the other 90% of the SNs should already have between them more than enough symbol files to reconstruct the file. The attacker can't do anything to increase their odds of getting just their SNs "closer" to all of the relevant symbol files that are related to the particular file they want to suppress.

Remember, this is all based on the XOR distance of the SN's PastelID, which is constant and doesn't change over the life of the Supernode. In addition, there is a minimum "startup time" associated with changing everything by transferring the collateral PSL coins to a new Supernode in order to generate a new PastelID which might be "closer" to any of the symbol files of interest to the attacker. This all means that, even if the attacker acts maliciously to refuse to send any symbol files that are requested from it (they can't get away with sending corrupted or modified versions of the symbol files, since the other SNs know what the file hashes of each symbol file should be), this would still never prevent the user from getting the file from the rest of the "honest" SNs.

So why is this better than Arweave? Well, because Arweave uses an incentive-based "market" system to allocate file storage, where nodes can decide however they want (but presumably based on what will earn them the most money on the network) which files they should store. As described earlier, there is a built-in incentive for nodes in Arweave to replicate the rarest files, since these are the most profitable on average.

As Arweave explains in their Yellow Paper:

"The PoA algorithm also incentives miners to store 'rare' blocks more than it incentivizes them to store well-replicated blocks. This is because when a rare block is chosen, miners with access to it compete amongst a smaller number of miners in the PoW puzzle for the same level of reward. As a consequence of this, miners that prefer to store rarer blocks on average receive a greater reward over time, all else being equal."

But suppose that the Chinese government or similar state actor is trying to suppress a file it doesn't like on Arweave.

One strategy this attacker could take is to create a large set of "sock-puppet" nodes (see Sybill Attack), so that other participants on the network have no way of knowing that a bunch of nodes (which they might assume are independent) are actually all controlled by the attacker. Then, using these sock-puppet nodes, the attacker could store tons of copies of the files in question, thus driving down the attractiveness of storing those files so low that none of the other honest nodes want to store them, since they could do much better for themselves by storing rare files. This process could be done slowly over time until the vast majority of nodes storing the relevant files are under the control of the attacker. Only then, the attacker could spring into action, suppressing the file by refusing to correctly respond with the relevant data when requested by users and nodes on the network.

Of course, Arweave has mechanisms to verify that nodes do what they are supposed to, but in this case, the attacker might be perfectly happy to lose a bunch of AR coins as a penalty or fine for not sending the data when asked (especially if they always promptly respond to every other data request). The network can't be too punitive anyway, because there are always going to be "explainable" service interruptions from broken machines and network connectivity issues, which could lead to too many "false positive" situations. In this way, a powerful attacker could quietly and effectively suppress data on the network. And it all stems from giving the nodes a choice in which files they store— a choice that they can abuse to further their ends if they don't care about maximizing their profit on the network but instead have ulterior motives.

Ironically, Arweave claims in their Yellow Paper that allowing nodes to choose somehow makes their network more censorship-resistant:

"The Arweave protocol avoids making it an obligation to store everything, which in turn allows each node to decide for itself which blocks and transactions to store. This increases the censorship resistance of the network as nodes are not forced to store material they don't want to."

This is a false dichotomy, because there are other better alternatives besides the straw-man "every node stores every file" approach and the "any node can choose to store whatever they want to." A better solution in fact is to have some of the nodes responsible for some of the files, and to choose all this in a provably secure and efficient way that can be validated independently by each node doing local computations in a trustless manner.

As Arweave says in their Yellow Paper:

"Here, you will discover how the blockweave enables miners to democratically decide which storage content they collectively do or do not wish to store inside the network."

The problem is that we do **not** want democracy here, which is extremely vulnerable to all manner of attacks. Instead, we want an all-powerful absolute monarchy, but not where the monarch is a Supernode or other network participant (selected, say, on a round-robin basis, or worst of all, through elections), but where the network protocol itself is the monarch. Because the network protocol will always be fair and impartial, since it is executing deterministic code, and because that code is all ultimately "steered" by the emergent proof-of-work side of the network, which everyone can reasonably agree is a fair system and impossible for anyone to control or predict or otherwise "game".

And like any good monarch, it should have a reliable and efficient "secret police" to ensure that everyone is playing by the rules (by always observing everyone else's behavior and checking for compliance). And that secret police is comprised of whatever portion of the SNs in the network that are truly independent and "honest": they are running the official, unmodified Pastel software, and they are not manipulating anything else to interfere with the ordinary operations of the network, such as DDOS attacking the IP addresses of other SNs.

Even if a relatively large portion of the SNs are controlled by an attacker, the remainder that is honest can still "sound the alarm" about any malicious behavior by any of the attacker's SNs and re-route around the now untrusted SNs by not requesting symbol files from them and adding them to a shared "ban list" to warn other nodes about trusting the malicious nodes.

Importantly, any successful attempts to attack the network would have to be general attacks that would undermine the network as a whole— not attacks against a particular file, which are much easier to undertake while risking less in terms of financial penalties. And the reason for this, again, is that there is no choice available to network participants as to which files they must store to be considered in full compliance with their obligations to the network arising from the network protocol; thus, they have to "attack" all files at the same time, a much harder thing to do.

5.2 Using Tricks to Get Undue Influence Over the Network

One of Arweave's big ideas is to use game theory, where nodes make choices based on what makes the most sense financially. This is exemplified in what Arweave calls the Wildfire scheme, described in the following passage from the Yellow Paper; what sounds like a good metaphor for ensuring efficient network operation suffers from an analogous problem to what we discussed in the previous section:

"In the wildfire mechanic, a form of AIIA game (see section 6), each node in the Arweave network ranks its peers based on two primary factors. Firstly, the peer's generosity - sending new transactions and blocks, secondly, the peer's responsiveness - responding promptly to requests for information, in a similar mechanism to BitTorrent's optimistic tit-for-tat algorithm. The node then gossips preferentially to higher-ranked peers. This allows a node to rationalise its bandwidth allocation. It also has the effect of promoting pro-social behaviour on the part of nodes generally, given the practical implications of how every peer interacts with every other peer."

So they have nodes deciding which other nodes they should most interact with, so they can presumably optimize their profit potential by only communicating with the "good" nodes. That is, the nodes that have been sending them data faster, the ones that are "most generous". The problem is that those "good nodes" might actually be malicious nodes that manipulate this mechanism to gain the upper hand. That is, an attacker could dedicate resources to getting the fastest, lowest latency machines around the world, build up a good reputation in the network by always responding quickly, and then on top of that, run modified software that is "extra generous". That is, the attacker shares more than really makes financial sense, giving them a disproportionate influence on the network. By tricking other nodes into thinking they are good, the attacker lulls the network into a false sense of security.

For example, an attacker could potentially control most or all of the machines that a typical node is checking first to see what's happening on the network. Then at some future point, they can start acting maliciously, by obstructing network activities, selectively withholding messages or data, etc. In contrast, in Pastel, nodes can't manipulate their standing through their own actions (other than avoiding getting banned for violating the rules) to change how much other nodes rely on their messages. A well-intentioned idea for promoting "pro-social behaviour" turns into anti-social attacks by a committed and clever adversary. What is most important in a cryptocurrency project is security, not the "efficiency of markets" or some proof of how the "dominant strategy" is to follow the network protocol. This is a naive view of security. Instead, you have to always assume to worst of all participants— not only that they will attempt to cheat and steal from other network participants, but also that they might just be out for wanton destruction with no financial gain, simply to vandalize and undermine.

This is a very tough standard, but we do have powerful tools to solve the problems. And that is to remove all the discretion from any important actions between nodes, and to systematically monitor all network activity to verify that all nodes are acting in accordance with the protocol. Instead of trying to make the basis of the organization of activity on the network along the lines of maximizing profit, with every interaction based on making more money in the short term, in Pastel the financial incentive for the SNs is much simpler: simply follow the rules in general (so you don't get banned by your peer SNs) and you will make the standard amount for a Supernode— an income which is presumably attractive enough to justify the investment of the 5 million of PSL required to collateralize a Supernode. Then, everyone is incentivized to ensure that everyone else is following the rules, and if not, to ban them (this is possible because all SNs can independently verify everything stored in any part of the system, and they constantly do that on a rolling basis through the storage challenge procedure).

But even if SNs in Pastel are malicious, there is limited damage they can do. They can never pile on various exploits and advantages like nodes in Arweave can, built up over time through "economically irrational behavior" to ensure that they can have an outsized impact on the immediate functioning of the overall network when the time comes— especially if they have a specific target in terms of a set of particularly sensitive files that they want to suppress. This is important, because that is the exact scenario that is most likely to occur; for most data on the network, no one would care enough to do anything 99.999% of the time. But for the high-value target data, the leaked documents, or whatever it is, it might make sense to "call in your chips" and harvest the accumulated goodwill you've earned in the Arweave network by being an "extra good node" that is "super generous and pro-social" for years at a time.

Because you can focus your efforts like a laser beam just on those particular files, you can have a huge impact relative to the risk and cost of the attack.

6 Road Map

6.1 Fine-Grained Access Controls

In the initial version of Cascade, files that are stored will only be accessible[8] to the PastelID of the uploader (i.e., all requests must be signed by the private key associated with that PastelID). This way, the system is secure for the user, and also does not place much load on the network, since a single user would only occasionally request the file (perhaps only once, since they can make a local copy). But it would be useful to allow for additional functionality, as long as it can be done securely and without placing undue strain on the network.

In future version of Cascade, we plan on including the ability to create public, shareable links that can be used by people other than the "owner" of the file on Pastel. Each of these links would be associated with a new blockchain ticket, called an *access control ticket*, which would be used to pick from a large menu of options that give the user control over:

- **Access Control:** Who can access the file? Users can filter on various criteria, such as PastelIDs or IP addresses. Additionally, users can specify a secret password that grants access to the file.
- **Resource Limits:** What are the rules around how and when the file can be accessed? Users can specify how long the public link will be valid by choosing either a date or block height, or set a limit on the total number of views or total megabytes transferred via the public link.

Of course, the network resources to actually serve all these public requests aren't unlimited or free. Thus we must charge a fee in PSL to compensate SNs for the bandwidth cost so that users that share particularly popular links are forced to pay much more than a user who generates a link that is rarely accessed. By setting all the parameters beforehand around the maximum amount of data transferred, SNs can bid to provide that bandwidth for a certain number of PSL per megabyte, and the user's client would automatically select the N lowest-priced SNs. Only after the user has sent the fees to the SNs would the access control ticket become activated on the network.

[8]In the case of image files, lower-resolution thumbnail images will be accessible to all users of the network, but only the NFT owner or file owner can access the full-resolution original file.

6.2 Content Filtering by Nodes

The initial version of Cascade will not have any form of content filtering. However, all images that are processed by Sense (which would constitute the bulk of the data stored in Cascade) have an associated "NSFW" score based on a deep-learning model that quantifies the probability that a given image contains sexual/pornographic content. Thus, it would be possible to allow SN operators the ability to create a new kind of blockchain ticket, the *content filtering ticket*, that specifies the maximum NSFW score (from 0.0 to 1.0) that they are willing to store on their machines.

Obviously, a node that has strict limits on the nature of the files it is willing to store is worth less to the network, because they are contributing fewer resources. This sort of content filtering also complicated the file storage protocol, since we can no longer automatically assume that the "closest" set of SNs to a given file chunk is actually storing the file. There are ways to overcome this in various ways (for example, files that correspond to a highly NSFW image might require twice as many SNs to store the files so that there is room for some of the SNs to refuse without reducing the file availability), but they all add complexity and make the network less robust.

Thus, SNs that elect to impose strict limits on the content they are willing to store must be penalized in some way by the network— otherwise, all SNs would likely avoid NSFW files, reducing the utility of the network. One way to do this would be to reduce the storage fees payable to any SN that sets a limit on the NSFW score, with the reduction to the fee proportional to the strictness of the limit. The end user would still pay the same overall storage fee, but the difference between the full fee and the reduced fee could be burned, thus benefiting all holders of PSL because of the permanently lowered supply.