SECURE PLOT TRANSFER FOR THE CHIA BLOCKCHAIN

SHASHANK AGRAWAL*

ABSTRACT

Chia is a popular cryptocurrency that relies on proofs of space (PoS) for consensus. Plots are the unit of storage on Chia and form the basis of PoS generation. When a proof is found that meets the challenge requirements, farmers on the network compete to create blocks.

Plot generation and farming involve the use of secret information, which makes plot transfer a non-trivial task in Chia. In this short note, we propose a way to transfer Chia plots in a secure manner with the help of zero-knowledge proofs.

1 MOTIVATION

Chia is a popular cryptocurrency right now and among the few ones that are based on proofs of space (PoS). In Chia, farmers contribute capacity to the network by generating *plots*, which are at least 100 GiB each. As of July 3rd, 2022, the netspace was just under 23 EiB [1], so about 250 million plots were being used for farming on the Chia network (assuming that all plots are for $k = 32$ [2]).

Generating plots is a time-consuming and resource-intensive process; it could take several hours to generate a single plot [3]. Many people want to participate in the Chia network but don't have the resources to generate plots on their own. Also, buying expensive hardware for the sole purpose of plotting may not make sense for everyone. As a result, several marketplaces have sprung up online to facilitate the buying and selling of Chia plots like Plotting Cloud, Chia Factory, XCH Cloud Pool, and many others. These marketplaces also enable ex-enthusiasts of Chia to sell their plots to someone else and recover part of their cost.

Storage device manufacturers may also be interested to sell *pre-plotted* drives, i.e., drives that come with one or more plots out of the box. Customers can use the plots to earn rewards on the Chia network right away, so these drives can easily be sold at a premium.

2 BACKGROUND

Plot generation and farming in Chia are cryptographic processes involving the use of secret information, so the trading of plots does *not* just boil down to a simple file transfer. To understand the nuances, let's see how plots are created in Chia and used for proof generation.

* Senior Technologist in the Software Solutions & Algorithms team at **Western Digital Research**

BLS SIGNATURES All keys in Chia are BLS-12-381 keys [4], following the IETF specification [5]. BLS keys and signatures can be *aggregated* in a non-interactive manner [6]. If we have two key-pairs $(\mathsf{sk}_1, \mathsf{pk}_1)$ and $(\mathsf{sk}_2, \mathsf{pk}_2)$ and we would like to produce a signature on a message m under the aggregated public key, then for each pair we need to have either the secret key itself or a signature on m.

MINING ARCHITECTURE A typical mining architecture in Chia consists of the following components: a harvester machine that generates plots; a farming machine that generates blocks; a pool that farmers can participate in; a wallet where rewards can be directed to; and others [7].

PLOT KEY For every plot in Chia, a random independent key called the *local secret key* is generated and stored in the plot itself. A separate key called the *farmer secret key* is associated with the farmer machine. The local secret key of a plot, together with the farmer secret key, define the plot's *plot secret key*. In other words, plot key is an aggregation of local key and farmer key (see the aggregation of BLS keys discussed earlier) [8] .

MINING POOLS Mining is a very resource-intensive and highly competitive process with uneven returns, so miners big and small come together to form mining pools. Almost all of the mining in many cryptocurrencies happens through pools. Unlike other cryptocurrencies though, support for mining is directly built into the Chia protocol [9]. Seven-eighth of block rewards goes to pools and one-eight goes to farmers. Pools generate their own keys which can be incorporated into plots.

PLOT ID & NFT Next comes the important concept of *plot ID*. A plot ID is derived in one of two ways: hash of a pool public key and a plot public key, or hash of something called a `p2_singleton_puzzle_hash` (or `pool_contract_address`) and a plot public key. Either ways, the size of any plot ID is always just 256 bits. Plots are generated in a *deterministic* way from plot IDs, i.e. the plot generation algorithm takes a plot ID as input and generates the same plot every time. Note that plots are much much bigger than plot IDs even though the former is derived deterministically from the latter.

If a plot ID is derived from a pool public key and a plot public key (the first method mentioned above), then the plot is tied to a specific pool *forever*, which may be a problem sometimes. For instance, if a farmer is not happy with the policies of a pool, he/she won't be able to use the plot to farm for a different pool. The second method above is meant to address this problem but official support for it was added to Chia later [10].

To generate plots through the second method, *plot NFTs* need to be set up [11]. A plot NFT is a type of contract on the blockchain which the farmer can control, and in which he/she can set his/her choice of pool. The farmer can also change the pool assigned to the plot NFT at any time. In a bit more detail, a plot NFT is associated with `pool_puzzle_hash`, `p2_singleton_puzzle_hash`, `owner_public_key`, etc. [12]. `p2_singleton_puzzle_hash` is used to derive one or more plot IDs; `pool_puzzle_hash` is the address where the pool rewards would go to; and, `owner_public_key` is the public key that controls the NFT (leaving a pool, joining a new pool, etc.). `pool_puzzle_hash` would typically come from the pool itself and `owner_public_key` could just be the farmer public key (or something else, as long as the farmer has the corresponding private key).

A normal user does not have to be worry about all these things when generating plots through the second method, but some of them will be important in our discussion here. A typical user would just use the UI to set up a plot NFT and the only thing he/she would need to provide is the pool URL [11] (from where `pool_puzzle_hash` would be taken); rest of the things would be taken care of internally. Switching to a new pool is also pretty easy through the UI: just choose

the 'Change Pool' option and enter the URL of the new pool. There is, however, a waiting period associated with changing pools.

BLOCK GENERATION Every time a challenge is available, a farmer would try to find if one of his/her plots could be used to respond to the challenge. The response can be seen as a proof of space. We don't need to get into how plots are used to generated PoS here, but details can be found in the Chia consensus document [13]. When a farmer has a *good quality* proof of space, he/she will put together a block consisting of the proof, some transactions, and a bunch of other things. The file `block_creation.py` in Chia code-base [14] defines all the fields that a block has.

Two important things that go into a block are `farmer_reward_puzzle_hash` and `pool_target`. `farmer_reward_puzzle_hash` is basically the address of the wallet where the farmer would like to receive his/her reward. (A receiving address could be converted into a puzzle hash and vice versa very easily [15].) `pool_target` is a structure that consists of two members: a puzzle hash and a max height. When plots are generated through the first method described above (plot ID being a hash of a pool public key and a plot public key), this puzzle hash is basically the receiving address for the pool rewards. To prevent farmers from specifying an arbitrary `pool_target`, a signature on `pool_target` under pool public key is required [16].

SIGNATURE We just saw what goes inside a block. Let's see what happens if we don't put any cryptographic protections on the contents of a block. When an attacker notices a block B on the network, he can steal the PoS from B and create his own block B′ with a new farmer puzzle hash (i.e., farmer wallet address) and/or a new pool target that he controls. With the same proof quality, B and B′ will compete to be included in the blockchain. If the attacker is better connected to the network than the victim and can act quickly, B′ may have an edge over B. In any case, the victim comes under a real danger of losing the entirety of the block reward to the attacker. This type of attack is usually called *front-running*.

To prevent this attack, Chia requires that a block also contains a signature under the plot public key. To be precise, a block is supposed to contain signatures on several things (reward chain VDF output, challenge chain VDF output, and foliage block data) but they are all supposed to be under the plot public key [14, 16]. Importantly, the foliage block data includes `farmer_reward_puzzle_hash` and `pool_target`, and there is a signature on a hash of all the data in the block. Now, if someone tries to steal the PoS from a block, they would not be able generate a signature on any new farmer/pool puzzle hash because the plot secret key is unknown to them.

CHIA VS BITCOIN It is instructive to compare Bitcoin and Chia at this point. The front-running attack for Chia we described above doesn't actually work for Bitcoin. In Bitcoin, we know that miners are supposed to compute a proof of work (PoW) which essentially amounts to computing a nonce s.t. the contents of a block hash to a certain value. This nonce computation is highly non-trivial: millions and millions of nonces may have to be tried before the right one is found. If a miner prepares a block B with an appropriate nonce value ν and publishes it on the network, an attacker can certainly play around with the contents of the block. However, for any block B′ different from B, the probability that ν or something related to it works out is almost zero. So the attacker would have to restart the nonce computation process for B′, which is no different from generating a block from scratch.

In simpler terms, PoW in Bitcoin depends on the contents of the block itself while PoS in Chia doesn't (it only depends on the challenge value and the plot). As a result, Chia needs a signature to bind the contents of a block while Bitcoin doesn't (of course, signatures are needed for other purposes in Bitcoin).

3 THE PLOT TRANSFER PROBLEM

How do you transfer a plot generated by yourself to someone else? While at first it may seem like a simple file transfer, the presence of plot secret key complicates matters quite a bit.

For this section, suppose Alice wants to transfer a plot P that she generated to Bob. The first thing to consider is how did Alice generate P. Here is one way of doing things:

- Bob generates a farmer secret key.
- He hands over the corresponding public key to Alice.
- Alice generates a local secret key and combines it with the farmer public key to obtain a plot public key.
- She uses the plot public key (with either a pool public key or a pool contract address, which can be provided by Bob too) to define a plot ID.
- She then generates a whole plot from it.
- She transfer the plot to Bob.

Please refer to the paragraphs on 'Plot key' and 'Plot ID & NFT' in the previous section if needed.

This is actually a nice and secure way of generating plots for someone else. We will call this approach the *basic* approach. Some online plot sellers like Plotting Cloud take this approach and storage device manufacturers can take it too. The main problem with the basic approach is that prior to generating a plot, one needs to know who the plot is for so that their farmer public key (and pool public key / puzzle hash) can be used to generate it. In many cases, this is not possible or doesn't make sense. Some examples are:

- Alice generated P for herself to participate in Chia farming but she is not interested in it anymore (farming too costly now, too many plots to manage, etc.). Many online marketplaces have sprung up for people like Alice to sell their plots to whoever's interested like XCH Cloud Pool.
- Alice is a large storage device manufacturer/vendor and wants to sell pre-plotted drives in physical stores like BestBuy, etc. When setting up plots on a drive, she doesn't know who will end up buying it.

In these and many other situations, Alice would have generated or would have to generate plot *secret* keys (and other things perhaps) on her own, so the basic approach doesn't work.

THE PROBLEM Let's consider the situation where Alice generated a plot P *without Bob's involvement* but now wants to transfer the plot to Bob. Note that Alice would have generated any and all the secret material involved in the generation of P *on her own*, so plot transfer won't just be about sending P to Bob as in the basic approach.

Alice could try to cheat Bob in this situation: Alice would transfer P to Bob but keep a copy for herself (this could happen with the basic approach too) and/or Alice would transfer the secret material to Bob (plot secret key, etc.) but keep a copy for herself.

Suppose Alice holds on to P and all the secret material after transferring it to Bob. With the same plot P, both Alice and Bob will now compete to win block rewards; sometimes Alice will win, and sometimes Bob. Alice is actually better off repurposing the storage space she has dedicated to P (at least a 100 GiB) for some other plot P′. Sure, generating a new plot P′ would require some resources, but the

cost of managing P' would be same as P and she wouldn't have to compete with Bob for the block rewards of P'.

Unfortunately, the same argument doesn't extend to the case where Alice just keeps the secret material (but not the actual plot P) because it's much much smaller when compared to P. A plot secret key is merely 32 bytes, so it doesn't hurt Alice to keep a copy of the key after she has transferred it to Bob. This key would enable Alice to carry out the front-running attack we discussed before (see the paragraph on 'Signature' in the previous section). When Bob produces a PoS from P and publishes a signed block B with the PoS, his wallet address, etc., Alice could steal the PoS to create her own block B' with an address that she controls. She wouldn't have any problem generating a signature for B' because she kept a copy of the plot secret key around. Alice would then try to front-run Bob's B with her own B'! If B' is added to the chain before B, of which there is a reasonable chance (proofs are of the exact same quality), then the block rewards would go to Alice's wallet. This is a very serious attack: Alice is able to win block rewards through a plot that she no longer stores. She is able to provide a proof of dedicated space without actually dedicating any space!

POOR BOB As far as we know, the default Chia client does not provide a way to detect such attacks, so an ordinary user like Bob may not even realize that he is under attack. Even if Bob is smarter than an average user and is able to detect the attack manually, or by writing a piece of code, there isn't anything that he can do about it. The best he can do is not do business with Alice again. However, the name Alice may not be much more than a pseudonym. The next time Bob buys a plot from some Carrol, she may as well be Alice :-)

Also observe that it's very difficult for Bob to convince someone else that he is being cheated, after all both his block B and Alice's block B' are valid. In particular, there is no way for Bob to *prove* that he did *not* generate B' himself.

To summarize the section, transferring plots in a secure manner is a big problem if the recipient wasn't involved in plot generation itself.

4 ZERO-KNOWLEDGE PROOFS

In this section and the next, we present zero-knowledge proofs and apply them to the problem of plot transfer. We do not intend to provide a formal treatment of cryptographic proofs here, nor a formal proof of security. We will keep the discussion high-level and accessible.

We will now introduce a cryptographic tool called zero-knowledge proofs that has found many applications in the blockchain world, but have not been considered in the context of Chia before. Zero-knowledge proof (ZKP) is a cryptographic technique that allows to prove statements about secrets *without* revealing them. For example, for some known public key pk, one could prove that two different ciphertexts c_1 and c_2 encrypt the same message without revealing what the message is! One could think of digital signatures as some kind of zero-knowledge proof too: by signing a message, one shows that one knows the private key without actually revealing it.

Let us look at ZKP a bit more formally. ZKP is a protocol between a prover and a verifier that involves one or more rounds of messages. If there is only one round of communication (which has to be from the prover to the verifier), then the ZKP is called *non-interactive*. There are at least three components to a ZKP system: some public *parameters* that everyone has access to, a *statement* that needs to be proven, and a *witness* that helps to prove the truth of the statement. While the parameters may stay the same across many sessions, statements and witnesses typically change from session to session. In a session, the prover tries to convince the verifier that

a certain statement is true using the witness that he/she has. At the end of the session, the verifier either accepts the proof or rejects it.

Let us go back to the encryption example. In this example, we could set up a ZKP system where the public key would be one of the parameters, a statement would consist of two ciphertexts, and the corresponding witness would consist of the underlying message and the two random values used to generate the ciphertexts. The ZKP system would then provide a way for the prover to convince the verifier that the two ciphertexts encrypt the same message. Observe that the prover could just send the witness to the verifier and have the verifier check for himself/herself that the message when encrypted with the two random values indeed produces the ciphertexts in the statement. However, this would *not* be zero-knowledge because the verifier learns a lot more than he/she needs to. A ZKP will make sure that the verifier learns no more than the fact that the two ciphertexts encrypt the same message. In particular, the verifier will not learn "which" message the ciphertexts encrypt.

We would also like to point out a few more things here. First, not every statement is true and a malicious prover may try to prove false statements. For instance, a statement in the example above could consist of two ciphertexts that actually encrypt different messages. For such statements, no prover should be able to show that they are true (in other words, verifiers should always output reject in such cases). Second, a true statement may have several witnesses that can attest to its truthfulness. The knowledge of any of them should be good enough to convince the verifier. Third, proofs are not always about showing if something is true or not; they could also be about the knowledge of something that is hard to find. For instance, let H be a hash function that is pre-image resistant (like SHA-256), i.e., given a hash value h, it is infeasible to find any m s.t. $H(m) = h$, even though most likely one would exist. We could have a proof system for a prover to show that he/she knows the pre-image of a hash value h without revealing what it is.

This brings us to the properties of a zero-knowledge proof system. Somewhat formally, a ZKP is supposed to have the following cryptographic properties:

- *Completeness.* If a statement is true, then a prover must be able to convince a verifier (e.g., through the knowledge of one of witnesses).
- *Soundness.* If a statement is false, then no prover should be able to convince a verifier. Such malicious provers need not follow the steps of the protocol.
- *Zero-knowledge.* During an interaction with a prover, a verifier does not learn anything beyond the validity of the statement. Here, verifiers may not follow the steps of the protocol.

The most interesting type of ZKPs for blockchains are the non-interactive ones where provers send just one message to verifiers and the latter immediately output accept or reject. The great thing about these proofs is that they can be stored on the blockchain and anybody could verify them (they are not verifier-specific; there is nothing that "comes" from a verifier anyway). For these proof systems, both the prover and verifier can be represented by 'simple' algorithms (as opposed to interactive algorithms). We can represent the prover algorithm by P, which takes the parameters, a statement, and a witness as inputs, and outputs a proof. We can represent the verifier algorithm by V, which takes the parameters, the same statement as before, and the proof as inputs, and outputs accept or reject.

One particular type of non-interactive ZKP used in some crypto-currencies is zk-SNARK, short for zero-knowledge succinct non-interactive argument of knowledge. In addition to the properties that any non-interactive ZKP has, zk-SNARKs have a few others:

- *Succinctness.* Proofs are short, i.e., the bit-length of a proof is small, and they can be verified quickly. For some systems, proofs can be as small as a few

kilobytes and verified in a few milliseconds, no matter how big/complex statements are.

- *Argument.* Soundness holds against computationally bounded provers only, i.e., a prover with unlimited computational power could prove false statements too. This is less of a feature, more of a limitation. However, in practice, this is nothing to be worried about.
- *Proof of knowledge.* Prover actually *knows* the witness. Mere existence of witness doesn't suffice. To elaborate a little: A regular proof convinces a verifier that a statement is true, i.e. a witness exists for the statement, but an argument of knowledge provides a stronger guarantee—that the prover actually knows the witness.

Perhaps the first major use of zk-SNARKs in the real world was to build Zcash [17], widely considered as one of the most private cryptocurrencies in the world. The largest known deployment of zk-SNARK is on the Filecoin network [18], a decentralized storage-based cryptocurrency.

To wrap up this section, we point out that zk-SNARKs are not particularly efficient for provers. Depending on the statement to be proved and the computational capability of a prover, it could take anywhere between a few seconds to a few minutes to generate a proof. However, irrespective of the statement size, the proofs are always short and easily verifiable (succinctness). For proof systems like Groth16 (used in Zcash [17] and Filecoin [18]), the proof size is just 0.2 - 0.3 KB and it only takes a few milliseconds to verify a proof [19].

5 SECURE PLOT TRANSFER

We propose to use zero-knowledge proofs (zk-SNARKs in particular) to address the plot transfer problem in Chia, which comes up when the recipient is either not involved in the plot generation process or not even known at the time of generation (see Section 3).

Going back to the Alice and Bob example, we know that Alice doesn't gain much from keeping a copy of the plot P itself but she can easily hold on to the secret material involved in the generation of P—specifically, the plot secret key. She can steal Bob's proofs of space (PoS) and use the secret key to generate valid blocks. This leads us to following question:

> *Could there be a way for Bob to show that he has a valid PoS without actually revealing it, so that Alice can't steal it?*

Indeed, ZKPs could actually enable us to do such a thing!

Just throwing the zero-knowledge idea at the problem wouldn't solve it though. We have to be very careful in how we apply it, making sure we don't introduce new attack vectors. Moreover, introducing something like this to Chia would mean modifying the Chia blockchain in a crucial way (a hard-fork would be needed; more on this later).

POS VERIFICATION First let us see how a PoS π generated from a plot P is handled by the network. π needs to be verified against the plot public key and the challenge, and its quality has to be computed. If π is valid and of "good" quality, then it could be included in the blockchain. If we look at the Chia code-base, then `block_header_validation.py` invokes a function `verify_and_get_quality_string` on π [16], which is defined in `proof_of_space.py` [20]. This function does a few different things including checking whether π passes the plot filter. It calls `get_quality_string` at the end, which in turn calls `validate_proof`. To go further, we need to switch to a different part of the code-base under `chiapos` (Chia Proof of Space

library). While `chia-blockchain` is mostly in Python, `chiapos` is implemented in C/C++. Under `python_bindings` in `chiapos`, we find that `validate_proof` is bound to `ValidateProof` [21]. We finally find the actual implementation of proof verification under `ValidateProof` in `verifier.hpp` [22]. If a proof verifies, then `ValidateProof` calls `GetQualityString` to compute the quality of the proof.

A ZKP SYSTEM To apply ZKPs in this context, we will specify a statement with a plot ID and a challenge value whose witness will be a PoS. A prover would like to show that his/her PoS is valid w.r.t. the plot ID and the challenge, without revealing what exactly the PoS is. In other words, the prover needs to provide a proof (this proof is different from PoS of course) that he/she knows "something" that would make `ValidateProof` return successfully. From now on, we will denote this proof with κ. The reader should not confuse κ with π, the notation for proofs of space.

Making a few things concrete, we can summarize the discussion so far as follows. When a farmer generates a block, instead of providing a plot ID and a PoS π in the block, he/she will provide the plot ID and a ZKP κ. Any verifier/validator on the Chia network will derive the current challenge value from the publicly available information (see `calculate_pos_challenge` in proof_of_space.py [20]) and use the ZKP verifier to test whether κ is a valid proof w.r.t. to the plot ID and the challenge. The ZKP system will convince the verifier that the farmer indeed knows a valid PoS, but the verifier will not learn what it is during the process. From now on, we will use `ZKValidateProof` to denote the ZKP verifier. Note that `ValidateProof` needs π, the actual PoS, while `ZKValidateProof` can work with κ.

We defer the discussion of how replacing π with κ affects Chia until we address some other important issues related to the use of ZKPs .

PROOF QUALITY We have skipped over one important detail: `ValidateProof` does not just validate PoS; it also returns a quality value. With the PoS now hidden through a ZKP system, how can the verifier compute the proof quality anymore? There is a simple way of dealing with this problem: along with a zero-knowledge proof κ, a farmer would also provide a quality value v. We alter `ValidateProof` so that it also takes v as an input and compares v with the quality that it computes internally. We will now redefine `ZKValidateProof` to be the ZK version of the new `ValidateProof`. `ZKValidateProof` outputs accept if and only if the PoS is valid *and* the comparison succeeds.

STEALING ZKP ITSELF One might wonder what happens if Alice steals the ZKP κ itself. Until now we were worried about Alice stealing Bob's PoS π, so we replaced it with a 'blinded' proof κ, which doesn't reveal anything about π. Also, the blockchain logic is modified so that the verifiers check for the validity of κ instead of π. However, notice that Alice doesn't really care about π itself; she just needs to present 'something' that a verifier will accept. Therefore, instead of stealing π, she would now steal κ. With κ and a wallet address that she controls, Alice will create her own block and sign off on it with the plot secret key. So we are now back to where we started: Alice can still front-run Bob :-(

Not all is lost however. We can actually redeem ourselves by making a small tweak to the zero-knowledge statement that Bob or someone else would prove. Note that the statement is *only* about the PoS itself and not tied to anything else in the block. In the least, we need to tie PoS and wallet address together so that Alice cannot replace the latter with something of her choice. Therefore, instead of showing, "I know a PoS π that satisfies `ValidateProof` ...", Bob shows that "I know a PoS π that satisfies `ValidateProof` ... *and* the hash of π and my wallet address (concatenated together) is h." (Bob reveals h as part of the proof; it is added to the block along with other items).

What happens when Alice steals κ now and tries to create her own block? If she replaces Bob's wallet address with her own, she would have to reveal a hash of π

and her own address, which she can't because she doesn't know π. She can still replace other stuff in Bob's block with her own but, if the block is valid, Bob will get the rewards and not Alice. We could actually go one step further and prevent Alice from modifying anything in Bob's block. In the original Chia protocol, let $h^\star$ be the hash of everything in a block except the plot ID and the PoS. In particular, $h^\star$ will have the wallet address and all the transactions hashed in. Bob could show that "I know a PoS π that satisfies `ValidateProof` ... *and* the hash of π and $h^\star$ (concatenated together) is h." We have now tied the entire block together!

As discussed before, in the current Chia protocol, a block's contents need to be signed with the appropriate plot public key. If block generators produce ZKP for the statement described above, then such signatures would be unnecessary! More on this in the following section.

5.1 Pooling

An important aspect of the plot transfer problem that we have ignored so far is that plots are associated with pools. In fact, seven-eighth of block rewards goes to pools and only one-eighth goes to farmers. A farmer could also choose to create his/her own pool and associate all of his/her plots to this pool. While self-pooling ensures that the entire reward goes *directly* to the farmer, it also means that he/she may have to wait for a long time to get any reward. If a farmer chooses to participate in some large external pool, he/she would usually farm at a lower difficulty level and receive smaller rewards—but on a more consistent basis. In the long run, the amount of reward obtained would roughly be the same either way (slightly lower in the latter case because the pool operator would generally take a small cut).

Now recall that plot IDs are generated in one of two different ways. A plot ID could be a hash of a pool public key and a plot public key (the old way), or hash of a `p2_singleton_puzzle_hash` and a plot public key (the new way). Let us consider the former simpler approach first.

5.1.1 Plot generation from pool public key

This is not the recommended choice for plot generation anymore. However, we cannot ignore it because tens of millions of plots have already been generated through this method. In the Alice-Bob situation, Alice may be self-pooling or participating in an external pool, say ePool. In the latter case, when Alice transfers a plot P to Bob, she would have to share ePool's public key too. Bob would need to make sure that ePool is a legitimate mining pool (in particular, not controlled by Alice herself). Bob will register with ePool if he is not already registered with them. He would share lower difficulty PoS with ePool privately, so we don't have to worry about Alice stealing anything there. Additionally, Bob would need a signature on `pool_target` from ePool. Recall that `pool_target` consists of a puzzle hash and a max height, where puzzle hash is the receiving address for pool rewards.

The other case is when Alice self-pools, i.e., Alice knows the pool secret key. In this case, Alice would have to provide Bob a signature on a `pool_target` of his choice (after all, we want Bob to receive the entirety of block reward just like Alice did in the past). If Bob needs a signature on a different `pool_target` in the future, then there may be a problem (Alice may not still be around). Alice could potentially transfer the pool secret key itself to Bob but she may have used the same pool for many of her plots and not sold all of them to Bob.

5.1.2 Plot generation through NFTs

The other method of generating plots is through NFTs. The exact mechanics of this is a bit complex but the UI provides an easy of setting up NFTs, changing pools, etc.

Please refer to the paragraph 'Plot ID & NFT' in the background section for some details.

Here too we can run into the same issue we saw before. Many different plots could be associated with a single plot NFT by using the NFT's `p2_singleton_puzzle_hash` to derive IDs for the plots. So if Alice generates her plots by setting up an NFT first, she couldn't just share the secret key corresponding to the `owner_public_key` (which controls the NFT) with Bob.

However, if there is a set of NFTs exclusive to the plots Alice sells to Bob, then Alice could hand over the secret keys corresponding to the `owner_public_keys` for those NFTs. Bob could then set the pool in those NFTs to whatever he likes. There is of course the problem that Alice may keep a copy of the secret keys with her and try to mess around with the NFTs later. However, Alice may not gain much out of this beyond creating some nuisance for Bob because he can always 'undo' her changes.

6 CHANGES IN CHIA

One downside to our solution to the plot transfer problem in Chia is that it requires a hard-fork. We do not see a way of securely transferring plots in Chia without modifying it in a significant way.

PROOFS AND VERIFICATION The primary change we bring to Chia is a new type of proof and its verification. If we use a zk-SNARK proof system, then the new proofs are about the same size as the old ones (200-300 bytes) and the new proof verification is quick too (but not as quick as the old one). The time to generate a proof, however, will go up significantly: a regular PoS is just a quick look up of some values from the storage medium whereas a ZKP would likely involve hundreds of thousands of public-key operations.

CO-EXISTENCE OF PROOFS One interesting thing to observe is that both the old and the new proof system can co-exist on the Chia network! If a farmer generates plots on his/her own and protects the secret keys involved, then he/she can just use a regular PoS. Even if someone steals the proof, they don't have the secret key to generate new signatures. On the other hand, if the farmer has obtained a plot from someone else and is worried that they may still have a copy of the key, he/she can use the ZKP system. We can designate a special bit in the blocks for farmers to indicate which kind of proof they are going with. A worried farmer will spend more time generating a proof but for the blockchain and the verifiers it wouldn't make much difference.

ELIMINATING PLOT SECRET KEYS If we do away with regular PoS and just have ZKPs on the Chia network, then the plot secret keys don't have a role really. Anyone could have the secret key of a plot but they won't be able to do anything unless they have the plot itself. Indeed, as we mentioned before, if we have a ZKP tying an entire block together, then there is no need to have a signature on the block.

Therefore, we can actually simplify the plot generation process itself by eliminating plot secret keys altogether and picking arbitrary random values for the plot public keys. These values could be combined with either a pool public key or a `p2_singleton_puzzle_hash` to derive plot IDs.

REFERENCES

[1] Chia Blockchain Explorer. https://xchscan.com/.

[2] Chia Plotting Basics. https://www.chia.net/2021/02/22/plotting-basics.html.

[3] Reference Plotting Hardware. https://github.com/Chia-Network/chia-blockchain/wiki/Reference-Plotting-Hardware.

[4] BLS Keys. https://docs.chia.net/docs/09keys/keys-and-signatures/.

[5] Dan Boneh, Sergey Gorbunov, Riad S. Wahby, Hoeteck Wee, and Zhenfei Zhang. BLS Signatures. Internet-Draft draft-irtf-cfrg-bls-signature-04, Internet Engineering Task Force, September 2020. Work in Progress.

[6] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 435–464, Cham, 2018.

[7] Peer to Peer system. https://docs.chia.net/docs/02architecture/p2p-system.

[8] Plot Public Keys. https://docs.chia.net/docs/09keys/plot_public_keys.

[9] Bram Cohen. Pools in Chia. https://www.chia.net/2020/11/10/pools-in-chia.html.

[10] Bram Cohen. Official Pooling Protocol Launched. https://www.chia.net/2021/07/07/official-pooling-launched.en.html.

[11] Pooling User Guide. https://github.com/Chia-Network/chia-blockchain/wiki/Pooling-User-Guide.

[12] Pooling | Chialisp. https://chialisp.com/docs/puzzles/pooling.

[13] Chia Consensus Document. https://www.chia.net/assets/Chia-New-Consensus-0.9.pdf.

[14] block_creation.py. https://github.com/Chia-Network/chia-blockchain/blob/main/chia/consensus/block_creation.py.

[15] Address and puzzle hash converter. https://www.chia.tt/convert.

[16] block_header_validation.py. https://github.com/Chia-Network/chia-blockchain/blob/main/chia/consensus/block_header_validation.py.

[17] What are zk-SNARKs? https://z.cash/technology/zksnarks/.

[18] zk-SNARKs for the world. https://research.protocol.ai/sites/snarks/.

[19] zk-SNARKs vs. Zk-STARKs vs. BulletProofs? https://ethereum.stackexchange.com/questions/59145/zk-snarks-vs-zk-starks-vs-bulletproofs-updated/63778.

[20] proof_of_space.py. https://github.com/Chia-Network/chia-blockchain/blob/main/chia/types/blockchain_format/proof_of_space.py.

[21] chiapos.cpp. https://github.com/Chia-Network/chiapos/blob/main/python-bindings/chiapos.cpp.

[22] verifier.hpp. https://github.com/Chia-Network/chiapos/blob/main/src/verifier.hpp.