Whitepaper: Gjallarbro

HashMapsData2Value.algo

October 2022

Contents

1 Acknowledgements

Thanks to barnjamin and noot for sharing their respective codebases with me to serve as references. Thanks to kayabaNerve for the suggestion on how to use Ed25519Verify. Thanks to fabrice102 for pointing out the necessity of proving knowledge of the secret spend key. Credits to h4sh3d for writing the paper *Bitcoin Monero Cross-chain Atomic Swap*, which inspired this work.

2 Etymology

In Norse mythology the Gjallarbrú was a gilded bridge over the river Gjöll, guarded by the maiden giant Modgunn. The Æsir Hermod rides over it in his quest to recover his brother Balder from the kingdom of the dead, Helheim.

3 Introduction

This whitepaper introduces a cross-chain atomic-swaps protocol between Algorand and Monero. Of note is that Monero lacks a scripting language, making bridging assets using e.g. state proofs not possible. However, by relying on the smart contract capabilities of the Algorand blockchain and some simple elliptic curve cryptography, a protocol can be devised that allows for the peer-to-peer trustless cross-chain swap of *Algo* and *XMR*.

The Github repository for Gjallarbro can be found under HashMapsData2Value/gjallarbro.

4 Theoretical Background

This section includes information pertinent to understanding the operations or actions available with Algorand and Monero used in the protocol. For more comprehensive information on Algorand and Monero, refer to the Algorand Developer Portal and Monero Docs respectively. Zero to Monero: Second Edition by Koe, Alonso and Noether is also an excellent and comprehensive textbook.

4.1 Monero's Addresses

Algorand, being a public blockchain, only has one set of keys. Monero on the other hand has two sets of key pairs: view keys and spend keys. Monero is private by default but this split allows for a Monero investor to voluntarily divulge their holdings with their view keys without giving permission to spend them.

A Monero address is composed of the following:

$$address = prefix||spend_{pk}||view_{pk}||checksum \tag{1}$$

where the prefix is a network byte corresponding to the network (*Main Chain* or *Test Chain*) and checksum is the first four bytes of the *Keccak-256* hash of the prefix, public spend key and public view key. || refers to the byte array concatenation operation.

Once *XMR* has been sent to an address the secret key $view_{sk}$ is required to "decrypt" all of the incoming transactions. In this way, Monero is optionally transparent. Note however that outbound transactions cannot be ascertained this way, and as such the balance at a given time cannot solely be determined with the $view_{sk}$ unless you have some guarantee that no transactions could have been made from it.

Possessing the $spend_{sk}$ allows the owner to actually make new transactions and spend the *XMR* sent to the address[1].

4.2 Secret Sharing

Both Monero and Algorand are based on ed25519 elliptic curve cryptography. For a key pair, the secret/private key represents a very large (scalar) number. The corresponding public key is a point on a Twisted Edwards curve:

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2 \tag{2}$$

and going

$$K = k \cdot G \tag{3}$$

where K is the public key (curve point), k is the private key (scalar) and G is a generator point provided by the standard.

Curve points have their own form of arithmetic allowing two points on a curve to create a third point on the curve. Any time the $+$ is used between two points it is curve point addition that is being referred to. To take the $\cdot$ multiplication operator means to recursively perform curve point addition of a point with itself a *scalar* number of times.

Points are pairs (x, y), but in the world of cryptography points are commonly compressed using point compression techniques, allowing for representing them with one value.

Note that scalars exist within the interval $(0, L)$, where

[1]Keep in mind that Monero is still an UTXO blockchain, even if the language in this whitepaper might suggest it is balance-based.

$$L = 2^{252} + 27742317777372353535851937790883648493 \tag{4}$$

The arithmetic that follows is modular arithmetic, also known as finite field arithmetic. Values outside of $(0, L)$ are wrapped around, and hence $\equiv$ (mod L) is used instead of $=$.

Assume that Alice and Bob possess scalars a and b. Then, the following holds:

$$\begin{aligned} A &= a \cdot G \\ B &= b \cdot G \\ A + B &\equiv (a + b) \cdot G \end{aligned} \tag{5}$$

The combination results in a point C from the scalar c:

$$\begin{aligned} C &= A + B \\ c &\equiv a + b \end{aligned} \tag{6}$$

Since A and B are public keys, it is no problem for Alice and Bob to share their public information with the other. It is only by sharing a or b respectively to the other that it is possible to recreate the secret c, with which it is possible to create signatures of messages, or crucially in a blockchain context, sign off on transactions.

Assume now that Alice has a $spend_{pk}^{Alice}$ public spend key, $spend_{sk}^{Alice}$ private spend key, $view_{pk}^{Alice}$ public view key and $view_{sk}^{Alice}$ private view key. Bob is in possession of an equivalent set of keys.

The two share their public keys with each other (both of which are contained in their Monero addresses) as well as their private view keys. They can now create the combined keys:

$$\begin{aligned} spend_{pk}^{Combined} &= spend_{pk}^{Alice} + spend_{pk}^{Bob} \\ view_{pk}^{Combined} &= view_{pk}^{Alice} + view_{pk}^{Bob} \\ view_{sk}^{Combined} &\equiv view_{sk}^{Alice} + view_{sk}^{Bob} \end{aligned} \tag{7}$$

as well as the $address^{Combined}$, using equation 1.

Alice, Bob or anyone else is free to transfer value into $address^{Combined}$. Alice and Bob would also have a transparent view of the balance of that address, since no one[2] would be able to spend the funds. It is only if Alice or Bob could avail on the other to hand over their $spend^{*}_{sk}$ private spend key that the $spend^{Combined}_{sk}$ could be reconstructed and allow that person to take control of any funds inside.

4.3 Leaky Operation

Assume that $address^{Combined}$ has been seeded with some *XMR* by Bob. Alice wants access to the funds and needs to convince Bob to hand over his $spend^{Bob}_{sk}$. She can do so by setting up a situation where Bob can divulge $spend^{Bob}_{sk}$ with the world and be guaranteed to receive something of equal or greater value, e.g. some Algo.

In fact, there would be no reason for Bob to lock up his precious *XMR* into the $address^{Combined}$ and be put at Alice's mercy unless she had already arranged for such a situation in the first place!

Alice creates a *stateful* smart contract on Algorand. On Algorand, stateful smart contracts can hold their own balance, offer APIs that can be called from outside, and create their own transactions.

Algorithm 1 Bob's "Leaky Claim" Call

```
1: $i \leftarrow$ input from outside smart contract
2: if ($fooScalarMult(i) == spend^{Bob}_{pk}$) && $scRightState()$ then
3:     return $closeToBob()$
```

Algorithm 1 illustrates a simplified version of an operation that could be called upon. An input i is passed along, ideally $spend^{Bob}_{sk}$. The hypothetical function *fooScalarMult* converts the input using equation 3 and if the results match up i must have been the corresponding $spend^{Bob}_{sk}$. *closeToBob()* empties out all the *Algo* held in the smart contract's account to Bob's Algorand address (which was already communicated and incorporated into the smart contract). *scRightState()* represents other things we want to be true before allowing this to go through, e.g. that the smart contract itself has been primed by Alice or primed itself following the expiration of a timestamp. More information on the details of that in Section 6.

Currently there is no equivalent to *fooScalarMult*(i) among Algorand's opcodes. Instead, *ed25519verify* can be used to get the same effect. Subsection 5 goes into a deeper explanation.

The key thing to keep in mind is that a blockchain operation like Algorithm 1 on Algorand is completely public. Specifically, the input i is visible to others in the blockchain history. By calling on it through a blockchain transaction, Bob is "leaking" his $spend^{Bob}_{sk}$ to the world and, more importantly, Alice.

[2]The odds of anyone accidentally generating that exact scalar is of course astronomically low.

In exchange for the *Algo* she locked up in the smart contract she gains the secret $spend_{sk}^{Bob}$ necessary to gain control over the *XMR* funds locked up in $address^{Combined}$.

Note that Alice, after having exchanged information with Bob, might also regret and ask for a refund from the smart contract. There are two such calls, as seen in Algorithms 2 and 3.

Algorithm 2 Alice's "Leaky Refund" Call

```
1: $i \leftarrow$ input from outside smart contract
2: if ($fooScalarMult(i) == spend_{pk}^{Alice}$) && $scRightState()$ then
3:     return $closeToAlice()$
```

Algorithm 2 is a leaky call Alice has access to at first. Since Bob might have already locked up his *XMR* in $address^{Combined}$, Alice must leak her own $spend_{sk}^{Alice}$ to him, such that he can recover his funds.

Algorithm 3 Alice's "Punish Refund" Call

```
1: if $scRightState()$ then
2:     return $closeToAlice()$
```

Algorithm 3 is a call that is available much later, if Bob is too slow, has timed out completely or is doing a denial of funds attack on Alice by not leaking his $spend_{sk}^{Bob}$ at an appropriate time. This call allows Alice to recover her funds without having to leak her own secret. In doing so Bob is "punished" by not being able to access the *XMR* funds he locked up.

5 Regarding ed25519verify

Algorand is a public blockchain with smart contract capabilities. Its consensus mechanism is called *Pure Proof of Stake*. Two types of smart contracts exist: stateless and stateful. A stateful smart contract acts like its own account, capable of holding its own balance of *Algo* and *ASA* (Algorand Standard Assets - arbitrary tokens).

A smart contract lives "on the blockchain" and can be interacted with, as well as having the ability to issue its own transactions. Algorand has its own stack-based language called TEAL, resembling an assembly language. However, other more developer-friendly implementations exist that compile down into TEAL bytecode, such as PyTeal and Reach.

The smart contract can call on certain operations, or opcodes. Of note is the *ed25519verify* opcode, which will be used until *bn256_scalar_mul* is introduced in a later version of TEAL. *bn256_scalar_mul* will allow for the direct calculation of equation 3 in the smart contract. *ed25519verify* is instead a signature verification opcode and what needs to be relied upon until then. For a given message *msg* signed with public key key_{pk} producing signature *sig*, *ed25519verify* verifies that signature *sig* is correct.

Note that *ed25519verify* requires that *msg* contains a concatenation of the string "ProgData", the smart contract hash and the actual message. In version 7 of TEAL, *ed25519verify_bare* was introduced, ignoring this entirely and only requiring the message to be the message. But for the sake of brevity we will ignore this and only refer to *ed25519verify*.

The *Schnorr* signature *sig* is in actuality the byte array concatenation of two values:

$$sig = s||R \tag{8}$$

where s is a scalar and R a point, calculated from the random nonce scalar r in accordance with equation 3. The values conform to the following equations:

$$s \equiv r + hash(R||K||msg) \cdot k \tag{9}$$

$$s \cdot G \equiv R + hash(R||K||msg) \cdot K \tag{10}$$

where k represents the private key, K the corresponding public key, G the generator point and *hash* the SHA512 hash function. $hash(R||K||msg$ is also called the *challenge*). Both s and R are presented publicly as the *signature* and by taking the scalar multiplication of s and comparing it with the right-hand side (all of which contains public information) in equation 10, it can be concluded that indeed only someone knowing k would have, with astronomically high probability, been able to produce s.

For this method to work safely however, r needs to have been truly randomly generated. If r was not randomly generated, or if r has been reused across different messages, an adversary could crack equation 9 as there would be one equation and only one unknown.

Suppose that r was intentionally set to a value and that value was known to the world. E.g., $r = 1$. The private key k could be calculated as follows:

$$\begin{aligned} s &\equiv 1 + hash(R||C||msg) \cdot k \\ s - 1 &\equiv hash(R||C||msg) \cdot k \\ \frac{(s-1)}{hash(R||C||msg)} &\equiv k. \end{aligned} \tag{11}$$

Note that the division operation in equation 11 is not actually division, as this is modular arithmetic. In actuality $(s - 1)$ is being multiplied by the scalar inversion of the challenge.

This is very useful. By putting in a clause in an escrow smart contract forcing the first 32 bytes of the 64 byte signature *sig* to correspond to the byte representation of the point R[3] calculated from $r = 1$, Alice/Bob can be sure that whatever signature the other provides as an input to be checked against *ed25519_verify* must have been "sabotaged". Sabotaged in the sense that anyone observing the blockchain and its history of transactions can extract the private key corresponding to the public key hardcoded into the smart contract at creation. And by intentionally providing the malformed signature, they are choosing to leak their Monero $spend^*_{sk}$.

6 Protocol

While Alice and Bob might be two friends, it is more likely that they are two strangers - or computers - separated across the Internet with little to no information about the other beyond that one has Algo, the other *XMR*, and they have a stated desire to exchange the two. The protocol presented here does not require them to put their trust into the other but rather into the protocol and that the blockchains involve maintain their integrity. Once they have committed to a trade at best they should always be able to recover their funds sans blockchain transaction fees. Note that for the purposes of this protocol, one actor accidentally "logging off" or suffering a loss of connectivity is considered malicious behavior.

6.1 Happy Path

The Happy Path refers to the scenario where both Alice and Bob act non-maliciously. Both get what they want, with minimal transactions.

The figure 1 illustrates the "happy" path the protocol follows when there is no malicious behavior, with blockchain-specific color coding.

Both Alice and Bob start off by generating the keys for the respective blockchains and exchanging public information. To ensure visibility both also share their secret view keys $view^*_{sk}$, before calculating the combined view key $view^C_{sk}$. Besides providing the aforementioned keys, each one uses their respective $spend^*_{sk}$ and signs off on the information, which the other person verifes using $spend^*_{pk}$. By providing the signature the party proves to the counter-party that they actually know $spend^*_{sk}$. The reason for why this is important is explained in 6.4. The signature should be on a message composed of a concatenation not only the keys but also some unique metadata made for the procedure. It should also have a prefix that makes it impossible to be transaction signature.

[3]PyTEAL representation: Bytes("Xffffffffffffffffffffffffff")

Alice		**Bob**
Generate Monero Keys		Generate Algorand Account
	$view_{sk}^{A}$, $view_{pk}^{A}$, $spend_{pk}^{A}$, $Addr_{Algo}^{A}$, Sig(...)^{A} $\longrightarrow$	
	$\longleftarrow$ $view_{sk}^{B}$, $view_{pk}^{B}$, $spend_{pk}^{B}$, $Addr_{Algo}^{B}$, Sig(...)^{B}	
Verify Sig(...)^{B} with $spend_{pk}^{B}$		Verify Sig(...)^{A} with $spend_{pk}^{A}$
Calculate $view_{sk}^{C}$		Calculate $view_{sk}^{C}$
Generate Smart Contract (SC)		Generate Smart Contract (SC)
Deploy SC		
		Verify SC
Fund SC		
		Fund $Addr_{XMR}^{C}$
Confirm $Addr_{XMR}^{C}$ funded		
Set SC "Ready"		
		Do "Leaky Claim" SC
		Receive Algo
Calculate $spend_{sk}^{C}$		
Receive XMR		
Delete SC		

Figure 1: The protocol's "Happy Path". Left side shows Alice's actions, the right side shows Bob's. Note the Monero orange and Algorand teal specific color-coding, for actions and data associated with the respective blockchain. Sig(...)* refers to a signature on the message (the collection of keys + some unique metadata) signed with $spend_{sk}^{*}$.

They both generate the escrow smart contract which will hold Alice's *Algo*, Bob checking to make sure they're both on the same page. Alice proceeds to fund it, and Bob seeing that follows by locking his *XMR* up in the multi-sig combined account $Addr^C$.

Once Alice is sure that Bob has funded the combined account, she signals to the smart contract that she is ready. Before this, Alice was in a position to refund the *Algo* to herself by leaking $spend^A_{sk}$. In return, Bob would have been able to calculate $spend^C_{sk}$ and take control over the *XMR* in $Addr^C$.

In the Ready State Bob can claim the *Algo* to his account by leaking his secret spend key $spend^B_{sk}$. Doing so gives Alice the ability to calculate $spend^C_{sk}$ and take control over the *XMR*.

The exchange is complete and the now empty smart contract is deleted.

6.2 Bad Scenario - Alice Disappears

After Bob has funded the combined account there is a risk that Alice simply disappears, rather than setting the smart contract into Ready State. Whether due to a bad disconnection, accident or simply malice this scenario deprives Bob of his funds. While Alice would also be unable to touch her funds, it might be worth it for her to lock away Bob's *XMR*.

In order to deal with this we introduce a timestamp: t_0. The zero refers to zero indexing, not that it starts at time 0 seconds.

t_0 is a timestamp agreed upon by Bob and Alice. Simply put, the smart contract will enter into the Ready State after t_0, regardless of whether Alice wanted it or not. Just like if Alice herself put the smart contract into Ready State, after t_0 Bob will be free to Leaky Claim the *Algo* owed to him.

The introduction of t_0, a clock, has some additional consequences for Alice. It would be entirely possible for Bob to not lock up his *XMR*, instead waiting for the clock to run down. Hence, if Alice notices that Bob is taking an awfully long time sending his *XMR*, she needs to Leaky Refund the *Algo* back to her.

She will need a safety margin as well, perhaps once 50%-75% of time has passed between start and t_0. Her Leaky Refund transaction will be visible to the world as it is gossiped by the Algorand relay nodes and as it enters the mempools of Algorand's participation nodes but before it has been included in a block. There is a danger that if the transaction is sent very close to t_0 and is somehow not included in a block in time, Bob will be able to not only re-take control over his *XMR* and send it elsewhere but also be allowed to do a Leaky Claim after t_0. Alice needs enough safety margin to ensure that her Leaky Refund gets into a block before t_0.

Algorand has a block time under 4-5 seconds and enjoys instant finality. Monero has a block time of roughly 2 minutes per block and needs 10 blocks of confirmation (i.e. 20 minutes finality). Of course, this assumes the blocks are not congested, or that fees do not make the transactions and the entire atomic swap infeasible. If Alice or Bob are in a hurry (or are simply trying to maximise the number of trades per day) they will want a short t_0. t_0 should be set accordingly to the appetite for risk.

Another alternative is to use a block height b_0 instead of time t_0. In case the Algorand blockchain suffers a stall, the block height will stop too.

6.3 Bad Scenario - Bob Disappears After Ready State

After the smart contract enters the Ready State (either due to Alice activating it or t_0 passing), the ball is in Bob's court. It is up to Bob to Leaky Claim the *Algo* owed to him, giving Alice control over the *XMR* funds in the process. However, it is possible for Bob to deny Alice access to her money by not doing the Leaky Claim in the first place, despite acting non-maliciously up to this point.

To deal with this, another timestamp t_1 is introduced. After t_1, the smart contract will enter a new state wherein Alice can "Punish Refund" Bob. The smart contract will, after t_1, allow her to claim her *Algo* again without having to provide any "Leaky" information. By refunding her money this way Bob never gets $spend^A_{sk}$, is never able to calculate $spend^C_{sk}$ and is thus unable to get access to his *XMR* funds again, unless he could somehow convince Alice to part with her own secret.

As before, there is a risk of Bob submitting the Leaky Claim operation very close to t_1 but failing to have the transaction included in time, getting it stuck in a mempool or in the gossip. Alice would be able to calculate $spend^C_{sk}$ and take control of the funds, and then post-t_1 also be able to Punish Refund Bob and receive her *Algo* again. In any case there is no reason, conceivable to the author, for Bob to hold off on doing a Leaky Claim and risk passing t_1 anyway, making this scenario quite unlikely.

6.4 Why prove knowledge of $spend^*_{sk}$?

It is important that each party proves that they have knowledge of the $spend^*_{sk}$ they used to generate $spend^*_{pk}$. Otherwise, one of two scenarios is possible, depending on who communicates their keys first.

6.4.1 Alice Shares First

Consider if Alice shares her keys first: $view^A_{sk}$, $view^A_{pk}$, $spend^A_{pk}$ and $Addr^A_{Algo}$.

It is then possible for Bob to calculate $spend^B_{pk}$ in order to manipulate $spend^C_{sk}$ in the following way:

$$\begin{aligned} spend^{'}_{pk} &= spend^{'}_{sk} \cdot G \\ spend^{B}_{pk} &= spend^{'}_{pk} - spend^{A}_{pk} \end{aligned} \tag{12}$$

where $spend^{'}_{sk}$ is a scalar Bob is generated himself, and the minus refers to point subtraction. Equation 7 showed how the combined Monero account is generated, the one holding Bob's *XMR*.

$$\begin{aligned} spend^{C}_{pk} &= spend^{A}_{pk} + spend^{B}_{pk} = spend^{A}_{pk} + spend^{'}_{pk} - spend^{A}_{pk} = spend^{'}_{pk} \\ spend^{C}_{sk} &= spend^{A}_{sk} + spend^{B}_{sk} = spend^{A}_{sk} + spend^{'}_{sk} - spend^{A}_{sk} = spend^{'}_{sk} \end{aligned} \tag{13}$$

Because Alice shared her keys first, Bob was able to calculate $spend^{C}_{sk}$! He now has access to the *XMR*... which he himself locked up. He can remove them if he wants.

However, note that he still does not actually know $spend^{A}_{sk}$. As such, he actually does not know $spend^{B}_{sk}$ either! As such he cannot claim the *Algo* Alice locked up in her smart contract.

Looking back at our protocol, because Bob is unable to claim the Algo, the time runs out and the smart contract enters $> t_1$. Alice simply refunds her *Algo* but in a non-leaky way. In the end both participants have their crypto back (minus transaction fees), however Bob has wasted Alice's time, significantly so depending on t_1 and without the punishment of losing his funds.

6.4.2 Bob Shares First

In this scenario, Bob shares his keys first: $view^{B}_{sk}$, $view^{B}_{pk}$, $spend^{B}_{pk}$ and $Addr^{B}_{Algo}$.

Similarly, Alice can calculate $spend^{A}_{pk}$ in order to manipulate $spend^{C}_{sk}$ in the following way:

$$\begin{aligned} spend^{'}_{pk} &= spend^{'}_{sk} \cdot G \\ spend^{A}_{pk} &= spend^{'}_{pk} - spend^{B}_{pk} \end{aligned} \tag{14}$$

$$\begin{aligned} spend^{C}_{pk} &= spend^{A}_{pk} + spend^{B}_{pk} = spend^{B}_{pk} + spend^{'}_{pk} - spend^{B}_{pk} = spend^{'}_{pk} \\ spend^{C}_{sk} &= spend^{A}_{sk} + spend^{B}_{sk} = spend^{B}_{sk} + spend^{'}_{sk} - spend^{B}_{sk} = spend^{'}_{sk} \end{aligned} \tag{15}$$

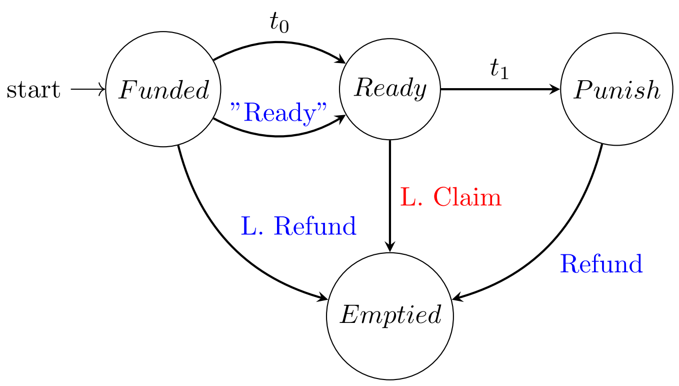

Figure 2: State transition diagram for the smart contract. Alice actions are colored in blue and Bob's in red.

As soon as Bob shares his keys Alice is in possession of the combined account. However, since they follow the protocol, Bob will not fund it with his *XMR* until after Alice has deployed her smart contract and locked up her own *Algo* in it. Alice does not know $spend^A_{sk}$ (since it is a function of $spend^B_s k$) so she will be unable to claim the *XMR* and then leakily refund herself her *Algo*. Instead, she could grab the *XMR* and then maliciously choose not to prime the smart contract into Ready mode. However, the smart contract will simply enter $> t_0$ mode whereby Bob will claim the *Algo* to himself. Despite leaking $spend^B_{sk}$ in the process it will be too late for Alice to leaky refund the *Algo* back to herself.

6.4.3 Proof-of-Knowledge of $spend^*_{sk}$

The solution is simply for both Alice and Bob to prove to the other that they know their respective $spend^*_{sk}$ secret spend keys. They can do so by actually generating $spend^*_{sk}$ and signing off on a message. The other party could then verify this signature using the public spend key $spend^*_{pk}$ equivalent that they received. Refer to section 5 for more information on how Ed25519 signatures (Schnorr signatures) are validated.

If we assume that there is no possibility of either party knowing the counterparty's secret spend key[4], it is assumed that having proof-of-knowledge of the secret private spend key is enough to be confident that the counterparty's public spend key was not used to create the party's public spend key. Of course, in this case, we truly want a random nonce.

6.5 Smart Contract - State Diagram

The state diagram in Figure 2 shows how the contract would go from being funded (and in a state to be re-funded to Alice), readied for Bob to claim the *Algo* and finally in a state for Alice to punish Bob. There are time/block height constraints that force the contract along the states, as well as leaky (L) actions Refund and Claim.

7 Future Work

7.1 Algorand Standard Assets

Algorand treats *ASA* as first-class citizens, meaning that they can be pretty much transacted just in the same way the native *Algo* can, with the caveat that you still need to use *Algo* to cover the transaction costs and minimum balance. This means that it is easy to adapt the same contract that facilitates *Algo/XMR* swaps to be able to handle *ASA/XMR*. People on Monero would thus be able to buy NFT Art, stablecoins, DAO tokens, etc in Algorand directly with *XMR*.

7.2 Adaptor Signatures

A possible evolution to this protocol would be to use so called Adaptor Signatures. Adaptor Signatures would be more discrete on the blockchain as some of the logic would be handled by Alice and Bob passing signed transactions off-chain and then choosing to spend them on the Algorand chain. Instead of a smart contract deployed by Alice with its own account, a standalone Algorand account would directly be used instead. In order to implement the bad scenario where Bob fails to claim his *Algo* and instead holds it hostage, a stateless contract would still be required, giving Alice permission to transact the *Algo* back after a certain period has passed.

Since the transactions are commitments, the very act of making the transaction would be enough to leak information to the counter-part. This method could be less "conspicuous" for others as there would be no *stateful* smart contract deployed. It also comes with less on-chain work, reducing the minimum balance required for Alice to lock up in the smart contract, and the *ed25519verify* on-chain operation that requires the opcode budget from three transactions. For the latter point, Algorand efficiency and low fees mean that it is currently not a big imposition.

For a first version however that is arguably easier to understand (more so once *bn256_scalar_mul* arrives), the protocol detailed previously in this document is enough to reliably and securely serve Algorand and Monero investors, and Adaptor Signatures can be implemented afterwards.

[4] If they did there would be no need for this protocol.

8 Conclusion

In this paper a protocol for trustless, peer-to-peer cross-chain atomic swaps has been described, with Algorand and Monero particularly in mind.