vacuumlabs

Programmable tokens

Whitepaper v0.1

June 12, 2024

Contents

1 Introduction

Motivation for programmable tokens

Cardano tokens are unique by being a direct part of users' balances. They enjoy native support within the Cardano ledger. The full custody belongs to the users. The tokens already have programmable token creation (minting) and token destruction (burning). They are used massively on the Cardano blockchain to both represent value in the form of different asset classes including stablecoins, and for purely technical reasons.

This contrasts with Ethereum's token standards such as ERC-20. There, tokens of a particular type are tracked in a single smart contract that essentially keeps track of its balances for all users. While this approach has its drawbacks such as troublesome recognition of which tokens a user holds and no self-custody of the tokens, it also provides a significant advantage of programmable transfers. Since a token transfer is a smart contract execution itself, any additional smart contract logic can be included there by the programmer of the contract; to be executed during token transfers. This feature is used extensively within the Ethereum ecosystem.

Currently, integrating the functionalities of Ethereum's ERC-20 tokens into Cardano's native tokens presents a significant challenge, partly because of a lack of a design to achieve programmable transfer functionality. The token's code can not control transfers, the code is not invoked. This lack of transfer programmability prevents multiple functionalities that are vital for compliance and sophisticated token management, such as blacklisting specific addresses, enforcing mandatory fees on token transfers, the ability to temporarily freeze tokens during critical situations, and much more.

While this could be solved with developing an *ERC-20*-like smart contract on Cardano, a place where the balances would be tracked, we believe that native tokens provide their own security and performance benefits which we do not want to compromise on. Our goal is rather to develop an extension over native tokens as they are. An extension on how to add and enforce programmable checks on tokens. This missing piece would bridge the gap between the capabilities of ERC-20 tokens and Cardano's native tokens.

For a deeper dive into how tokens are currently used and what potential vulnerabilities need to be accounted for in the smart contracts, we recommend checking out our two blog posts: Token Security, Token Security II.

Current state of research

We know of at least one other significant effort in this field, summarized under the Cardano Improvement Proposal CIP-0113? by Michele Nuzzi and Matteo Coppola. Since the design there has been evolving as well, we will try to summarize the ideas contained within very broadly, one at a time.

ERC-20 smart contract

The general proposed approach of the open CIP suggests creating a Cardano smart contract that would simulate the ERC-20's interface in a validator on Cardano. They created a modifiable smart contract which tracks user balances in its contract data (datum) and allows certain operations such as transferring the virtual tokens. While such smart contract has certain benefits over ours (see pros and cons of our design in Conclusion), it does not use Cardano native tokens.

As a direct consequence, it does not enjoy any of their benefits. Any solution that does not utilize Cardano native tokens would be very difficult to integrate into existing dApps as well. Decentralized applications often block other script inputs for security reasons, i.e. to prevent double satisfaction vulnerabilities. The only way to integrate those virtual tokens would therefore be to change all the scripts such that they allow those specific script inputs. Such a change would need to be re-audited as well, as it is not a simple change from the security perspective.

The scripts would also have to parse the datum of each virtual token UTxO on input and possibly on output as well. This means that e.g. a lending protocol would not hold a collateral value in its smart contract, but rather would be listed as an owner of a particular amount of the collateral's virtual tokens in the tokens' smart contract. This changes the design of existing protocols a lot.

Moreover, the implementation of the idea has been evolving also due to an innate problems with contention of such an approach. The simple reason for this is that even receiving tokens might require spending of an existing recipient's UTxO. This blocks his UTxO and could lead to denial of service. An interesting thing to note here, is that Ethereum's ERC-20 tokens do not suffer from this.

Another difficulty is that while the basic checks for native tokens such as the check that the sum of tokens on inputs and in the minted value is equal to the number of tokens on the output are performed by the ledger, in a solution utilizing such virtual tokens, all such checks would need to be in the smart contract itself. That brings performance issues as all this counts towards a common strict execution memory and cpu limit imposed on a transaction.

The single biggest advantage of this approach is the undeniable smart contract validation all the way from the virtual tokens' inception, through every transfer, to its potential burn. As the number of tokens are noted only in a UTxO, as long as there is no security issue with the smart contract, there is no way to not run the validation and change anything about the tokens – including their owner.

Bind tokens to its owner

The idea is to use native tokens but use the token name part of the tokens to encode the last valid owner of those tokens. This was our initial idea. We also saw that it was mentioned in a comment under the CIP by Las Safin.

The implementation of the idea means that for every valid transfer when the owner is changed to a new one, the old tokens are burned and new are minted. The programmable checks can be enforced in the minting policy. The burn and mint is necessary for the token name change to happen. Only tokens with a valid name represent any real value as dApps and users can (and should) start enforcing that tokens they receive are correctly set to belong to them. Even though there would not be a way to prevent invalid transfers, meaning transfers without invoking the validation logic, they would discredit the tokens. Ultimately, there would be no incentive to do this.

An integration with dApps would be simple. First of all, the tokens would still be represented as native tokens. Secondly, an owner of the tokens could be set to the validator hash. Finally, smart contracts would need to allow for a token name change of programmable tokens if the minting policy is run.

We researched this idea and came to a conclusion that while this would work for token transfers between users, there are security issues when we take smart contracts into account. The biggest issue is that tokens can become invalid and after a long time, they could be validated again. What's worse, it could all happen at a smart contract address. Imagine tokens that are set to belong to e.g. a very simple peer-to-peer lending smart contract. They could be lent to anybody using invalid transfers and become validated again by being moved to the same smart contract. That provides a recipe on how to avoid the programmable checks, avoid a blacklist or paying fees on transfer. What's more, such tokens would be on-chain indistinguishable from other valid tokens. We thought we could do better. The following design builds on this idea, but bulletproofs it.

Our approach

We do not like that there are functionalities that exist in other ecosystems (in this case Ethereum) but are not available on Cardano. Especially, if the reason is likely the technical complexity. We would like to push the technological limits of what can be delivered on Cardano. What's more, we want to empower projects regardless of their size. The programmable tokens' functionalities would provide huge benefits to Cardano if they were available to every project regardless of their budget or influence.

With our design, we do not want to go against Cardano's philosophy but rather complement it. Let projects choose if they need the ability to enforce programmable checks or not. If they do, provide an easy design they could use. We want to make it easy for decentralized applications to recognize and securely integrate programmable tokens.

2 Basic protocol idea

Objective

We can formulate the basic informal requirements on programmable tokens as follows:

- **Req. 1 — Enforced transfer programmability**. The developer of the programmable token can create a programmable check that is validated on each programmable token transfer.
- **Req. 2 — Composability**. The tokens must be usable with Cardano smart contracts similarly to native tokens, with minimal modifications. The verification should be as minimalistic as possible, so that even execution heavy smart contracts can use programmable tokens.
- **Req. 3 — Extension of native tokens**. The ledger provides us with the native tokens, does a lot of legwork there and the whole Cardano ecosystem works with them. We do not want to introduce something completely new. We want to build on top of it.

There is immediately a conflict in these informal requirements – native tokens can be freely transferred. There is no way to prevent it without modifications to the ledger. How can we enforce transfer programmability then? Our solution is to define two types of transfer for programmable tokens:

- **Valid transfer**. Tokens are transferred and the programmable check is invoked in that transaction.
- **Invalid transfer**. The tokens are transferred without invoking the programmable check. As they are just native tokens, this can always happen. There could be malicious intent behind such transaction such as sending tokens to a blacklisted address, but it could also be a simple mistake of an honest user.

Further, based on the whole transfer chain of a particular bag of tokens, we can also define two states programmable tokens can be in:

- **Valid tokens**. Tokens that were moved only using valid transfers throughout their whole history on chain.
- **Invalid tokens**. Tokens that were part of at least one invalid transfer in their history.

It is important to note that there is no way to prevent an invalid transfer of Cardano native tokens. There are also very good reasons to not try to fight this. We will present

them later. Therefore, we need modify the first requirement to be more realistic and to take into account the *validity* of tokens:

Req. 1 — Enforced transfer programmability. The developer of the programmable token can create a programmable check that is validated on each *valid* programmable token transfer. Further, a valid transfer of a token can *never* follow an invalid transfer of the same token.

It is easy to see that invalid tokens can never become valid directly from their definition. Users, wallets and dApps should take special care in checking that they receive only valid tokens. If that happens, even progressively, the value of invalid tokens goes to zero. They are unusable in the end. Note that this touches the fungibility of those tokens. They are still fungible among their respective class – valid tokens are fungible between each other and invalid tokens are also fungible between each other. However, valid tokens and invalid tokens are not fungible. It is possible to find out the type of the tokens. How exactly it is possible is very important. Let's now describe the technical details.

Transaction proofs

To facilitate this requirement, we need to have a way of knowing that the whole chain of transfers for specific tokens were valid. This is difficult for two reasons. Firstly, no validator is usually run when transferring tokens. Secondly, even if a validator was run, it would not see the past transactions.

Let us introduce *transaction proofs*. Transaction proofs are UTxOs created on every *valid* transfer of programmable tokens. They are evidence that the programmable check was invoked in the very transaction they were created in. Their creation is a necessary condition for the transfer to be valid. Even if the programmable check was invoked but the proof was not created, the transfer is invalid.

Since a UTxO creation is not generally controlled by any validation, we introduce the *proof's validity token*. For a proof to be valid, it needs to contain this validity token. The minting policy enforces all the programmable token requirements mentioned before.

Informally, that first and foremost includes two very important properties. It checks that all programmable tokens are still valid by requiring reference inputs of the proofs created in the transaction they were last transferred in. This can be checked by comparing the respective unspent transaction outputs' (UTxO) transaction references. Secondly, it checks that the programmable check is satisfied.

For simplicity, let's assume that we have only one programmable token in a transaction that we are validating. Let us describe the building blocks again:

- **Programmable token** A. The programmable token itself. This can be any Cardano native token created with a programmable on-transfer check in mind.
- **Proof UTxO**. A UTxO that proves a valid transfer for the transaction it is created at. For simplicity, let's assume that the smart contract governing these proofs is unsatisfiable and thus the UTxOs are unspendable.
- **Proof validity token** PVT. A token that is minted into the proof UTxO. Only proofs with this token are valid and can be used as proofs. The minting policy checks that all the input tokens were valid, and that the programmable check was invoked. These checks are generic for any programmable token.
- **Programmable check token** PCT_A. The minting policy checks the transaction and enforces the programmable check as defined by the programmable token A's developers, such as checking that no token is transferred into a blacklisted address, etc. It is also minted into the proof UTxO. By doing that, the proof contains something A-specific, proving that programmable token A's validation logic was invoked.

Proof validity token

Most of the generic protocol logic is implemented in the proof validity token PVT. Let us repeat that there is a single PVT for all tokens. It is not to be modified by programmable token's developers. The PVT checks all the important properties related to valid transfers of programmable tokens contained within the validated transaction. Let's start with a single programmable token type:

1. Exactly one new PVT is minted.
2. Exactly one new PCT_A is minted. This means the programmable check is enforced as it is part of the minting policy of the PCT_A token.
3. Both PVT and PCT_A end up in the same proof UTxO with the correct proof validator hash.
4. No other tokens other than the two tokens and Ada are present in the proof UTxO.
5. For each input containing any amount of programmable tokens A, there exists a valid proof UTxO among the reference inputs. Such referenced proof is valid iff it was created in the same transaction as the input it proves and contains both PVT and PCT_A tokens.

The 2nd check enforces the programmable check. The checks 1, 3 and 4 check a valid creation of a new proof. Finally, the last 5th check requires all the programmable tokens

A on input to be provably valid. If there is any single invalid token part of the transaction, the proof can not be created.

In the beginning of this section, we defined three requirements we want from our solution. We will now go through them and show that each of them is satisfied by the introduced protocol.

- **Req. 1 — Enforced transfer programmability**. If someone sent tokens A without invoking the programmable check, the PCT_A was not minted in the transaction. Assuming that no PCT_A tokens can ever leave proof UTxOs, there is no way to construct a valid proof in that transaction as a valid proof for tokens A needs to contain PCT_A. Once a proof is missing, tokens A are invalid. Recall that the proof needs to be created at the same transaction where tokens A are transferred. Further, those tokens can not become valid again thanks to the 5th check.
- **Req. 2 — Composability**. All that is required on top of non-programmable native tokens' is a creation of a single additional output UTxO (the proof UTxO), inclusion of possibly several reference inputs and a mint of 2 tokens. None of these are something that dApps commonly forbid as a new output and referencing other outputs does not influence the security of their protocol.
- **Req. 3 — Extension of native tokens**. Programmable tokens are indeed Cardano native tokens. They enjoy all benefits of the fact. The protocol distinguishes between valid and invalid programmable tokens. This distinction is recognizable by whoever needs to know. From the ledger point of view, they are all the same native tokens.

Different programmable tokens

Transferring different programmable tokens A and B in a single transaction is simple. Firstly, the PVT just needs to check that both PCT_A and PCT_B are minted and included in the newly created proof which can be the same UTxO. Secondly, it is worth repeating that a proof is proving a transfer of programmable tokens X iff it contains PVT and PCT_X. That describes a subtle but important note. Having invoked programmable token's A check in a transaction does not mean it proves a proper transfer of tokens B in that transaction. The PVT needs to handle that properly and check that for all inputs containing A, a valid proof containing PCT_A exists ***and*** that for all inputs containing B, a valid proof containing PCT_B exists.

Up until this point, we have thought of only a single type of programmable tokens. Now that we realize there might be multiple different programmable tokens, we might need to

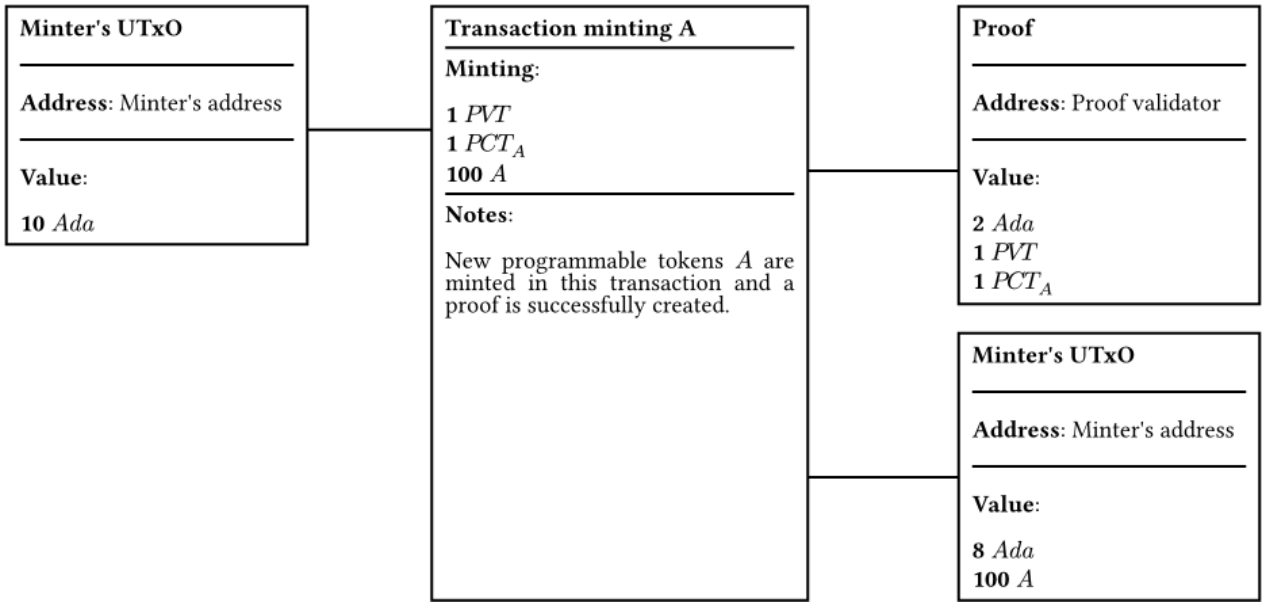

Figure 1: The minting of programmable tokens.

introduce a limit on the number of PCT_X tokens that can be contained in a single proof UTxO. That limit keeps the proofs small, preventing a potential denial of service (DoS) on subsequent transactions that need to reference them.

The fact that the proof validator and the PVT logic are constant across different programmable tokens is a huge advantage. One does not need to create multiple proofs for transferring multiple tokens as a result. The advantage will become even clearer once we introduce proof management of *dangling proofs* in an extension of the protocol covered in subsection 3.

Transaction walk-through

We have now covered the basic building blocks of our protocol. Let us now guide you through the basic transactions involving programmable tokens A.

Mint of programmable tokens

As all tokens, even programmable tokens need to be minted. As newly minted tokens can not be invalid by their history, the 5th check per above is trivially satisfied. The project minting programmable tokens needs to make sure that a correct proof for those tokens is still created, though. The programmable check is invoked as well and so the developers need to think of this case when implementing PCT_A's minting policy. See the transaction diagram in Figure 1.

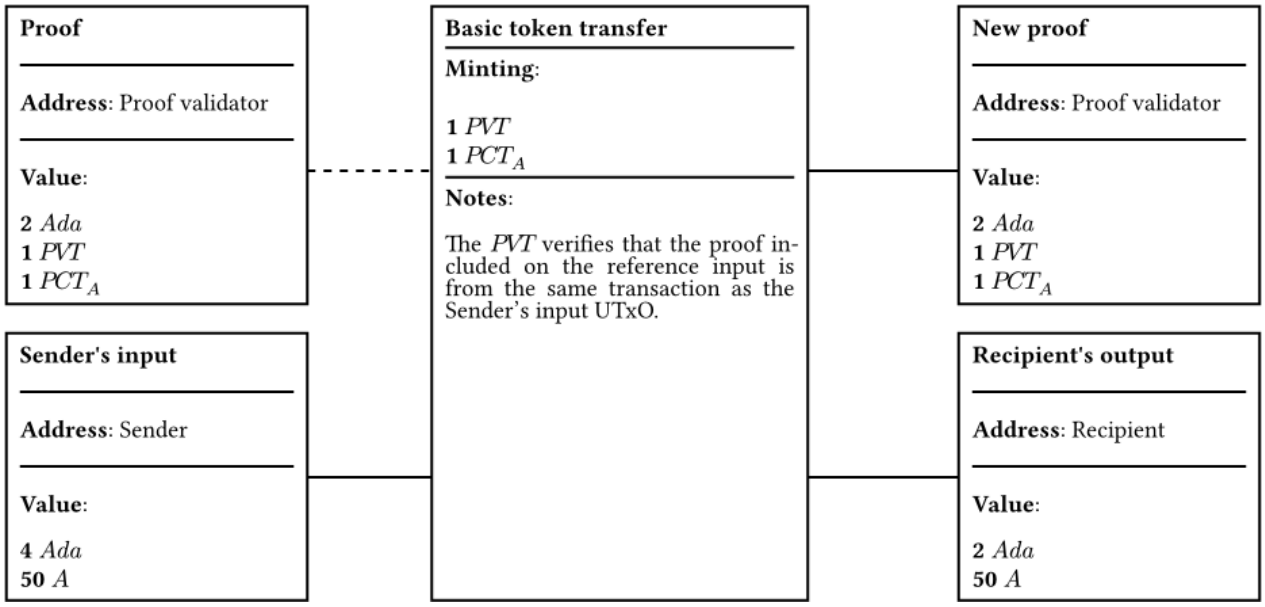

Figure 2: A simple transfer of programmable tokens.

Transfer of programmable tokens

A transfer of programmable tokens additionally requires the previous proofs to be referenced. See a simple transfer in Figure 2.

Figure 3 contains a transaction including two different inputs containing programmable tokens A that are in the same transaction. In our case, the two inputs were created in different transactions. Hence, we need to reference two different proofs in order for PVT to verify that all tokens are valid.

Finally, Figure 4 describes a transaction including a single input containing two different programmable tokens A and B. A single proof is created that contains both PCT_A and PCT_B. In case they would not come from the same transaction, but would instead come from two different outputs created across different transactions, multiple proofs could be referenced, containing at least PVT and PCT_A or PCT_B, respectively.

Basic use cases

The whole programmable functionality of tokens X boils down to the minting policy of PCT_X, the tokens' programmable check token. There are no restrictions on what can be checked in the policy. The minting policies have access to the whole transaction being validated, including all its inputs, outputs, datums, signatories, transaction's validity range, etc.

In this subsection, we will describe basic programmable functionalities that are heavily

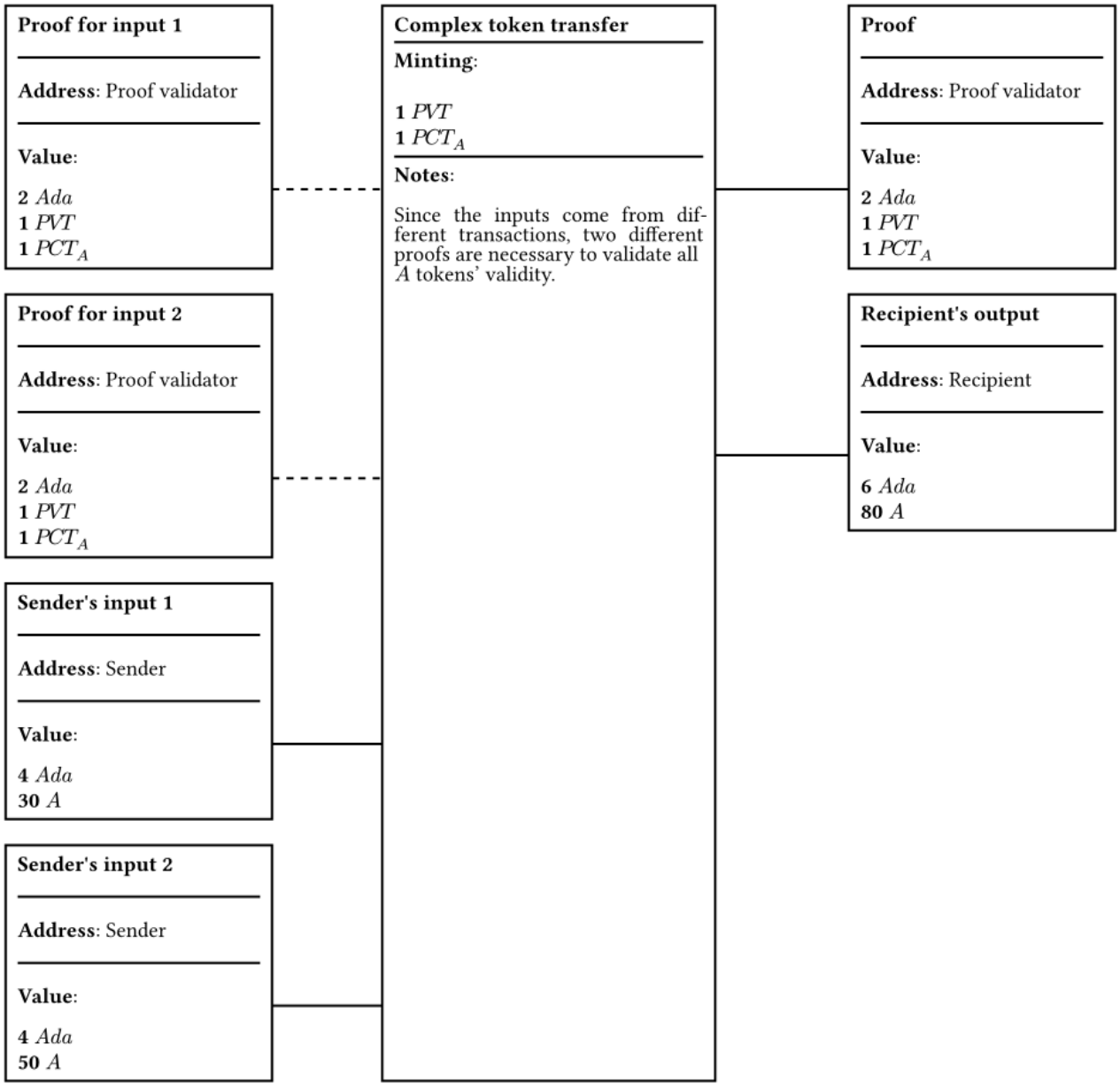

Figure 3: A more complex transfer involving two different inputs with A.

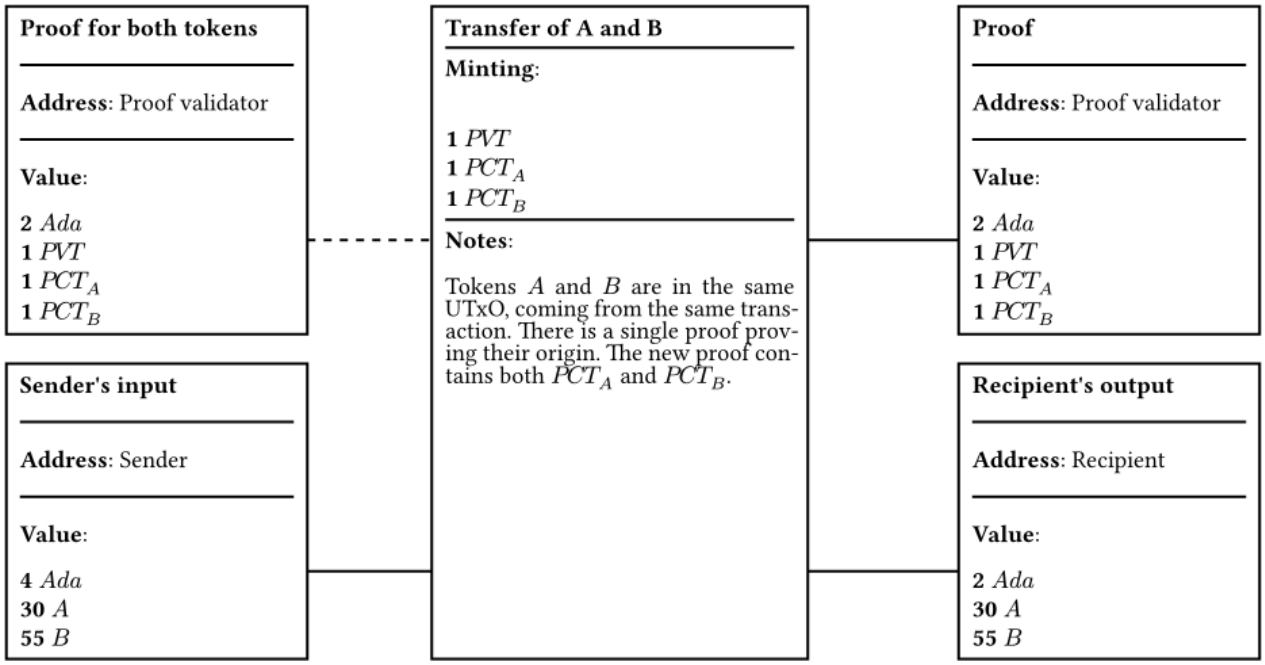

Figure 4: A more complex transfer involving two different inputs with A.

utilized on other chains and show that it is possible to implement them using our design.

Fee on transfer

Fee on transfer tokens require a fee payment to be made on every transfer of those tokens. In our case, we will enforce them to be paid on every valid transfer. There are a lot of options on how the actual fee should be set, in what currency it should be taken or when and if to waive the fee.

Let's take the simplest design possible, a constant Ada fee for every valid transfer. To achieve this, the PCT_X minting policy can be parametrized with a fixed project address dedicated to collect the fees. The minting policy would then check that an appropriate amount of Ada was deposited onto the address. The biggest security concern of this approach is a risk of a double satisfaction vulnerability. To avoid it, the fixed fee address can not be used for anything else and needs to be for this use case only!

As mentioned, this very simple design can be tweaked in a number of different ways. The fee can be computed by the amount of tokens transferred. It could even make use of oracles determining the current value of the tokens and base the computation on that. The oracles can be included in the reference inputs. Additionally, the fee can be taken in the tokens X. Every UTxO needs to contain Ada as well, though. As a result, there needs to be a solution on how to handle that requirement. One way to achieve this could be to set the fee collection address to a smart contract address that would verify that the leftover

Ada is returned to the owner as soon as the tokens are unlocked by the project owner.

Let us reiterate that anything is possible as the minting policy sees the whole transaction. To demonstrate a bit more exotic use case, you could introduce certain tokens whose presence would lower or waive the fees completely. What's more, NFTs are also Cardano native tokens and there is no reason why NFTs could not be programmable tokens. The programmable check token could recognize selling of the NFTs in common martketplaces, parse their datums, find the price and enforce that the royalty is really paid.

Token freezing

It is common for tokens such as stablecoins to implement a temporary emergency freezing functionality. It might be invoked in situations involving a major hack. The introduced design allows for a straightforward implementation of such token freezing.

The token issuer can create a single unique UTxO that would carry a datum with a simple boolean: ***frozen***. To make the UTxO clearly identifiable, a single unique validity token would be put into it. The PCT_X's minting policy would require a reference input of that UTxO for every valid transfer and it would validate that it carries the token and that the tokens are not frozen. That means that if the frozen flag is set to ***true*** by whomever is allowed to modify the datum, no valid transfer can happen. Note that there can be any kind of a robust access control mechanism implemented in the smart contract where the UTxO is locked.

Whitelisting

Whitelisting stands for specifying a set of addresses that are the only possible beneficiaries of the tokens. The whitelist can be partial, e.g. you could allow all public key addresses but maintain a whitelist of smart contract addresses.

Let's start with a simplified whitelisting. Say there is a fixed small-enough set of addresses that are the only possible beneficiaries of valid tokens X. The PCT_X minting policy could be parametrized with the set of addresses. During the minting, it could check that all transaction outputs' addresses are whitelisted or they do not contain X. In other words, no X can stay on an address which is not whitelisted.

To allow for a flexible whitelisting solution, the set of addresses might be part of the datums of a few UTxOs. The UTxOs would be protected by validity tokens recognized by the PCT_X policy. In order for the PCT_X to validate, for every output containing tokens X, a corresponding valid whitelisting reference input would need to be included in the transaction, proving the whitelisted property. Again, the smart contract governing the set of whitelist UTxOs could be arbitrarily complex.

Blacklisting

Blacklisting stands for maintaining a set of addresses that can not receive valid tokens. All addresses are allowed, except for those that are specifically blacklisted.

It can be achieved in a somewhat similar manner to whitelisting. A fixed small blacklist can be again part of the PCT_X minting policy. However, to maintain longer blacklists, a more complex trick is required. The minting policy needs to check that for every output containing X, the address is not blacklisted. It needs to distinguish between the address really not being part of the blacklist and the transaction creator not including it properly in the transaction. That is more complex than proving the whitelist property.

Let's store the blacklist in an on-chain sorted linked list. Each node of this linked list would be represented by a UTxO containing one blacklisted address. Further, it would keep track of the next node's address. The list is sorted by the address. Such linked list can be easily modified – elements can be freely added or removed by the authorized parties. Every operation impacts only a small number of nodes. Note that the blacklist UTxOs would again need to be protected by a validity token.

This linked list can now be used to prove that $address$ is not blacklisted using only a single node $node_i$ that can be in the reference inputs. To prove an absence of $address$ in the blacklist, the $node_i$ needs to satisfy the following: $node_i^{this} < address < node_i^{next}$. In other words, if the $address$ was blacklisted, that node's next blacklisted address would be different – the $address$ would be there. Note that there is at most one node which can prove a particular address' legitimacy.

The design of our programmable tokens also allows a more nuanced blacklisting. For instance, the minting policy can check the inputs containing the programmable tokens as well. That means that if an address is later blacklisted but it already contains valid tokens, those tokens could be virtually invalidated by disallowing the followup transfer from the address. What's more, the blacklisting could disable specific UTxOs instead of addresses. However, the tokens might be moved just before the blacklisting takes place.

3 Extensions of the protocol

Although our solution described in the previous sections fulfills all the properties we aimed for, it also features a few hurdles that we have not discussed so far. In this section, we will explain them and show potential extensions of the protocol that mitigate them.

Token invalidation

If a user spends a UTxO containing programmable tokens but does not create a proof UTxO, the tokens become invalid. As shown before, invalid tokens can not become valid again. The thing is, it could happen to an honest party by accident. That's too bad. Examples of when this could happen include:

- Using a wallet or a dApp that does not recognize programmable tokens and forgets to create a proof.
- Manually constructing a transfer transaction and forgetting about the proof.
- Using a wallet that does not realize that it needs to modify its UTxO management handling as well or it contains a mistake in the handling. The thing is, it can not simply rebalance UTxOs containing programmable tokens without creating a proof. Even a simple transfer of programmable tokens between the same address needs the proof to be constructed.
- Using dApps that do not protect UTxOs containing programmable tokens from a random spend to e.g. cover transaction fees. This is a similar point to the wallet's UTxO management, but on the dApp's side. As they construct transactions on behalf of the user, they also need to take special care to protect UTxOs containing programmable tokens or create a proof for the transfer.
- Using a smart contract that does not protect the user's tokens against a new attack vector related to programmable tokens, namely the risk of token invalidation by another party.

We realize that most of these issues will happen every now and then, but more frequently in the conception phase of programmable tokens. It should happen less often once programmable tokens are the norm. In the end, not many users construct their transactions manually.

Figure 5: A user creates a token repair request.

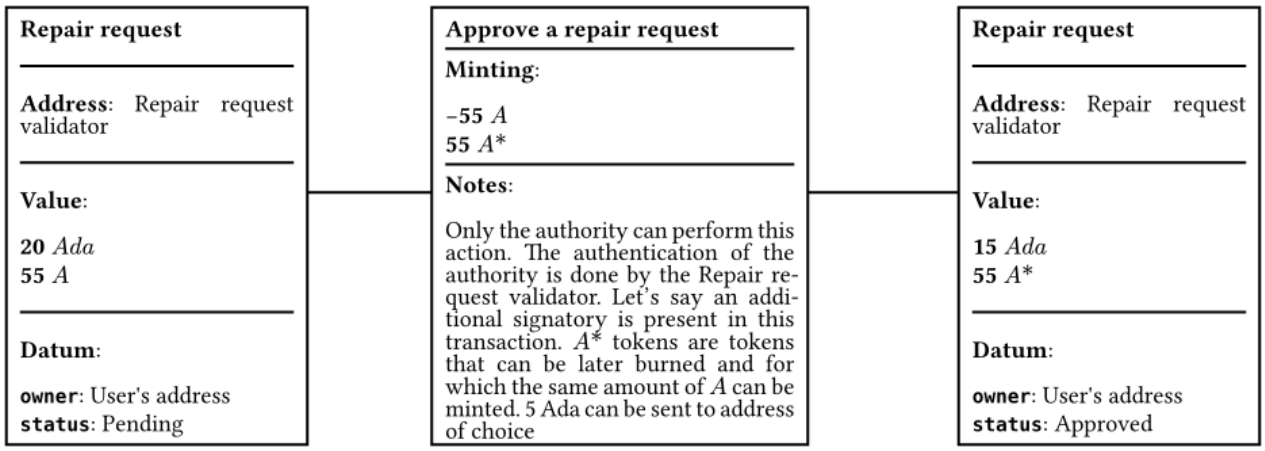

Figure 6: A repair request is approved by the authority.

To mitigate the issue, the tokens can be extended to be ***repairable*** by an authority. The authority can be centralized in the case of centralized tokens, s.a. stablecoins, utlize a DAO governance or be fully decentralized using robust on-chain oracles. A user realizing they accidentally invalidated their programmable tokens A can lock them into a ***repair request***. The authority's role would then be to monitor the repair requests, check the on-chain history of the tokens to see if they are eligible for a repair – namely if they were really invalidated by a mistake and could have been validated if it was not forgotten as opposed to them being involved in a malicious transaction. The authority can accept honest requests, burn the old A locked in the repair request and mint new A while constructing a proof for the new A. The whole process is shown in Figures 5, 6 and 7.

There's a reason why there are three steps in the process and not just two. To keep the PVT logic intact, generic and token independent, we can not both mint and burn tokens

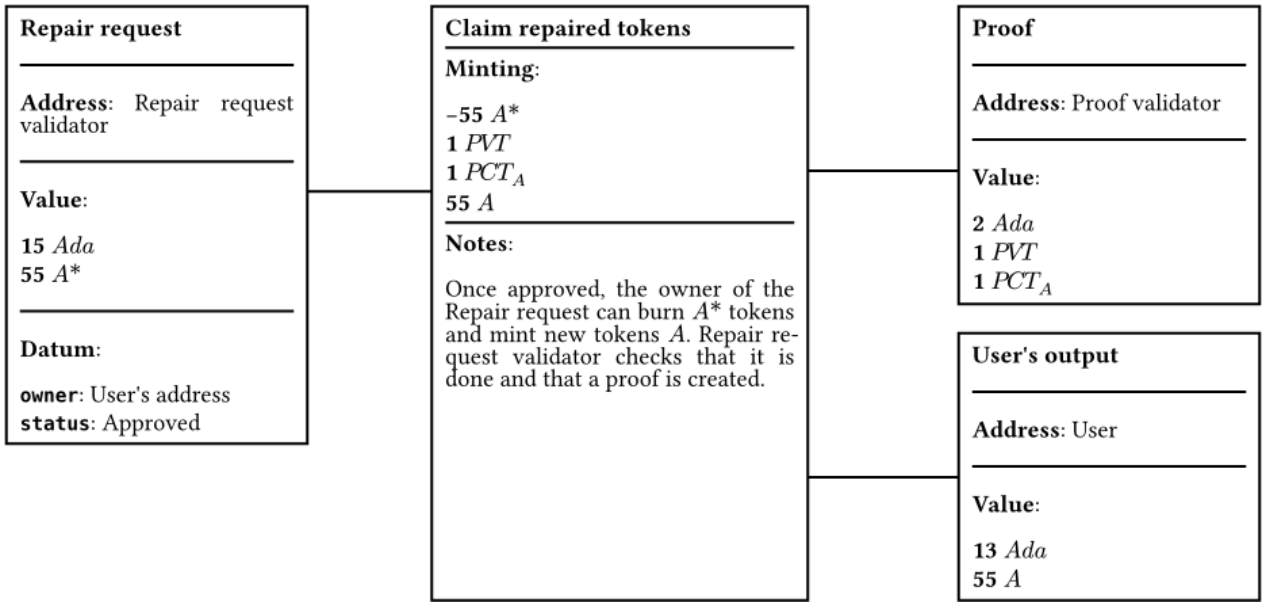

Figure 7: An approved repair request is claimed by the user. Tokens are repaired.

in the same transaction. That would equal to a transfer and the proof would be necessary for PVT to validate. The token issuer can, however, specify any repairable logic in the minting policy of the programmable token itself. To implement this, we introduce tokens $A*$ that are minted by the authority when tokens A are burned and that can be exchanged back to tokens A when the user claims an approved request, see Figure 7 again. From the protocol's perspective, a new proof can always be created for newly minted tokens. The mint of new tokens A is of course strictly controlled by the token issuer's A minting policy. This repair process needs to be explicitly allowed and thoroughly secured. The whole process can be built in a trust-minimizing way by utilizing smart contracts. The user can be sure that his tokens won't be stolen. Naturally, the authority can demand fees for the service to at least cover the incurred fees. A fee of 5 Ada is taken in Figure 6.

Proof management

Probably the biggest drawback of the design is that the proofs are unspendable. Since the ledger enforces every UTxO to contain a small amount of Ada, this makes more and more Ada unspendable forever.

The fact that the validator is unspendable is crucial for the security of the protocol. If a proof was spent, tokens that had been transferred in that transaction for the last time would become invalid. No further valid transfer would be possible. On the other hand, there's a certain class of proofs that could be destroyed freely. We call them ***dangling proofs***. A dangling proof is a proof such that there is no ***unspent*** programmable token for

which it is a proof anymore. In other words, the proof was already used for all the tokens it proved validity for. The tokens now require a new proof and this dangling proof is useless. Note, that it is not an easy task to decide whether a particular proof is dangling or not on-chain.

Proof archives

Instead of deciding whether a proof is dangling or not, we might reformulate what we are trying to achieve. We really want to just ensure the *availability* of the proofs forever. Instead of spending dangling proofs, it is thus enough for us to *compress* them. This is what *proof archives* are for. Proof archives are simply proofs that prove multiple different transactions, possibly for multiple different tokens, in a single UTxO. They can not be created directly, they need to be a compression product of multiple proofs that can be spent as a result of this merge. This recovers the locked Ada. Note that we allow the compression for all proofs, not just dangling proofs. Later, we describe why that is not a problem.

In order to help with deciding who can take the recovered Ada, we make all proofs and proof archives keep track of their *proof owner*. Only a proof owner can compress their proofs. The owner can be the holder of the tokens doing the transactions. However, it could also be a wallet or a dApp that constructs the transactions for the user. That way, the burden of compressing proofs is abstracted from the user. In either case, it should be the party that paid for the Ada contained in the proof. Further, we maintain that only proofs with the same owner can be merged together. Let us repeat that no proof is actually lost in the merging process. By archiving many proofs into a single proof archive UTxO, we ensure that the Ada can be recovered and that the proofs are still on-chain available for any potential transaction that needs them.

A valid proof archive does not need to hold PCT_X tokens anymore. It must have its own validity token, though. Any proof archive not holding the *proof archive validity token* $PAVT$ is not trusted. A valid archive can be referenced in transfer transactions instead of simple proofs.

Merkle trees

To compress multiple proofs together into a single proof archive and keep its datum simple, Merkle trees are used. That way, the size of the datum is constant for any number of proofs as only the root hash is kept on-chain. The whole tree can be obtained off-chain by checking the on-chain transaction history. Any usage of the proof archive would need to prove that a particular record is indeed present in the tree. Such proof can be validated in $\log n$ steps, assuming a balanced Merkle tree with n being the number of records in the tree. If the tree is not balanced, it might need as many as n steps. To prevent the validation from becoming infeasible, the height is tracked in the datum and there is a maximum

height limit enforced by the protocol.

We recommend building the tree in a smart way by keeping multiple Merkle trees on hand and merging them only once in a while and only if they have similar height. For example, an owner can have a bunch of simple proofs. Once he accumulates 64 simple dangling proofs, he can merge them into a single Merkle tree of height 6 and recover Ada from 63 proof UTxOs. He can repeat the process until he has 64 Merkle trees of height 6. Then he can merge them into a single Merkle tree of height 12. By doing the merging this way, he can have at most 128 proofs until he stores as much as $2^{12} = 4096$ simple proofs. The exact parameters can be changed depending on how often he wants to merge and how much Ada he can afford to have locked. You can see a simple proof archive creation in Figure 8.

Proof copies

A proof copy is really just a copy of any proof or a proof archive that can be referenced instead of the original proofs to validate a transfer. Proof copies serve two main purposes. Firstly, by creating a proof copy, you can set yourself as the proof copy owner. By doing that, you ensure availability of the proof since, as we mentioned earlier, only the proof owner can compress his proofs. This is critical and recommended for time sensitive operations. Secondly, a proof copy creation is a way to extract proofs from complex archives down into very simple proof copies. This might be useful if tokens are about to be used with an execution-heavy dApp but the proof has already been archived, as it brings down the validation necessary for the programmable tokens' logic and leaves more for the dApp. Note that this also assumes that the archive had been created on a proof that is not dangling. It is indeed a possibility and can happen either if the proof's owner needs to quickly recover Ada locked in it or if he is malicious. It is not recommended to archive non-dangling proofs, though.

A proof copy is again a generic protocol script's UTxO, meaning it is the same for any programmable token X. Similar to the proof archive, it does not hold PCT_X tokens but instead has its own ***proof copy validation token*** $PCVT$. $PCVT$'s minting policy checks that the copy is created properly. The datum keeps track of the transaction hash of the original proof and the PCT_X tokens it holds. Further, it also has an owner field which can be set to whomever. One key difference from proofs and proof archives is that the owner can choose to destroy proof copies anytime. It is just a copy, the original proof is still somewhere on-chain and so it does not compromise its eventual availability. See proof creation in Figure 9.

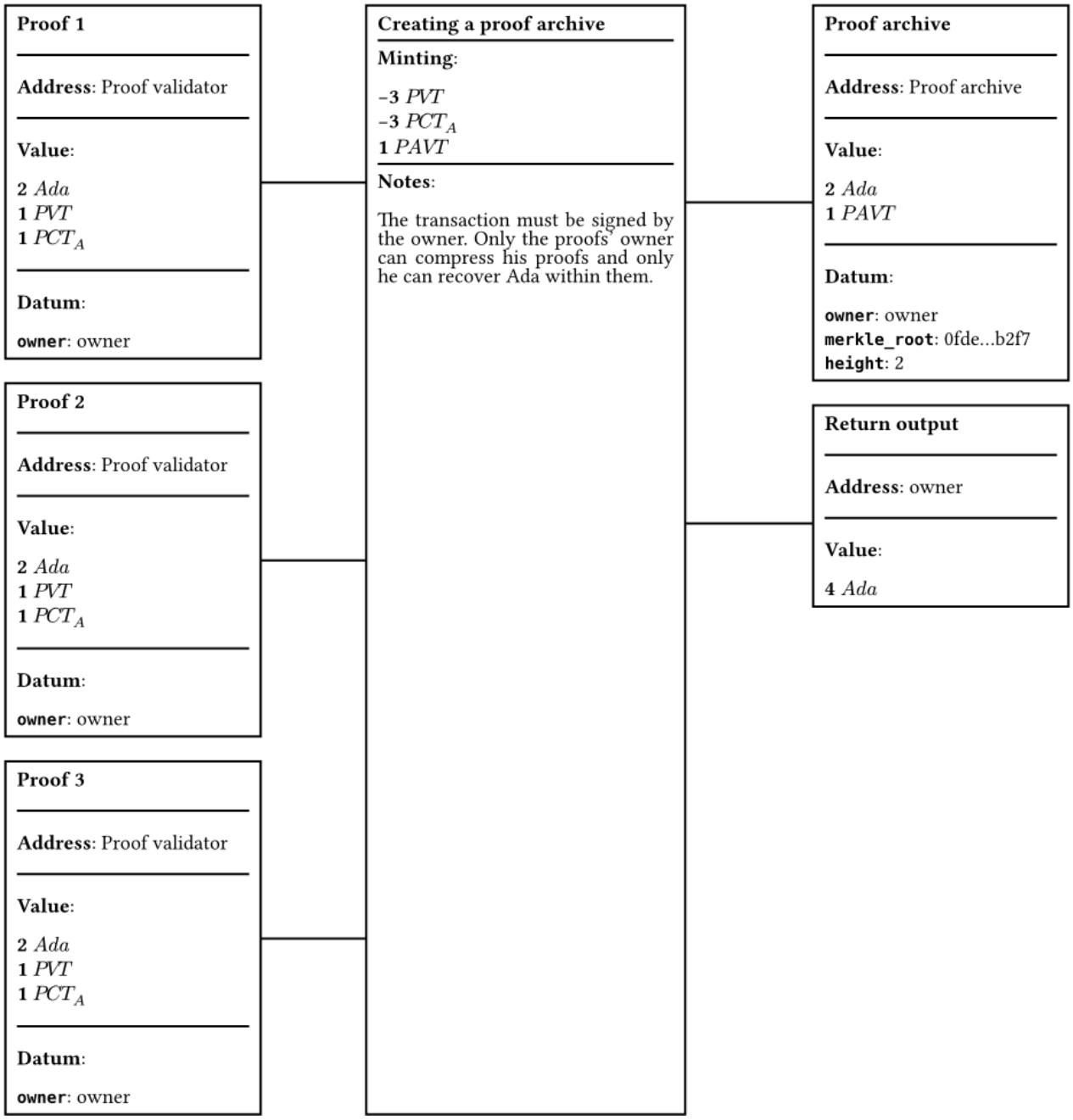

Figure 8: A simple proof archive creation.

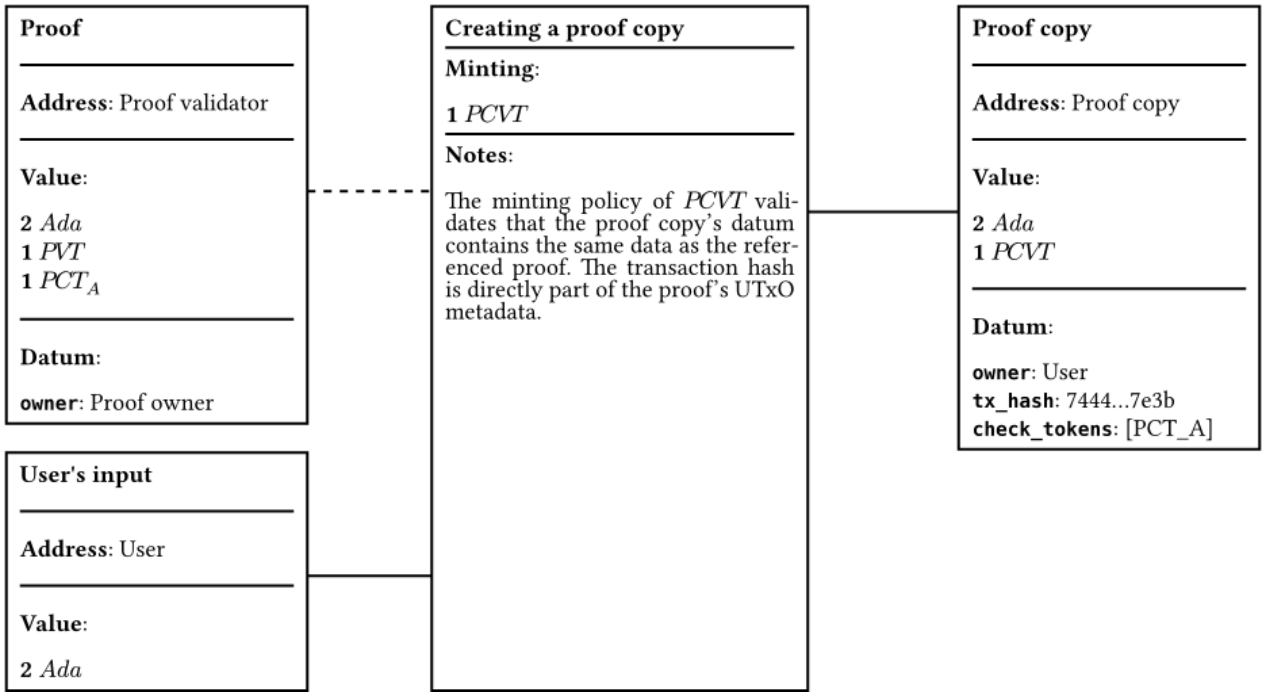

Figure 9: Proof copy creation.

Denial of Service attacks

In this section, we introduced certain ways to spend proofs and recover Ada locked within. However, proofs need to be referenced for any transfer to happen. It is crucial for the proofs to be available to be referenced. For that, we need to control when they can and when they can not be spent.

We have already described a bunch of preventive measures limiting the risk of Denial of Service attacks, such as allowing only the owner of the proof to spend it and compress it into an archive. We have allowed anybody to create a copy of any proof, any proof archive and to extract a simple proof from a proof archive under her own name. Finally, we enforce a maximum height of any Merkle tree referenced in a proof archive, making sure that it is always possible to validate that any proof kept within can be validated.

There's one last missing piece. We want to ensure that there is always a window of opportunity when a receiver of programmable tokens can create a proof copy. Imagine a simple transfer similar to the one mentioned in Figure 2. Tokens are sent from Sender to Recipient. Let's assume that the proof owner is Sender. In the following transactions, however, Recipient needs the proof to be available. Thanks to all the checks described above, he can be sure that the proof will remain on-chain. However, he can not be sure that it will be available in highly time-sensitive environments. Sender may choose to start a complex archiving process exactly when Recipient needs to reference the proof. As

a result, we introduce a small *cool-down period* for all newly created proofs and proof archives. It is a period during which it is impossible to spend them. Naturally, it is still possible to reference them. As a result, it is also possible to create copies.

4 Conclusion

TODO: Once the whitepaper contains all chapters.

A About us

Vacuumlabs has been building crypto projects since the early days.

- We helped create WingRiders, currently the second largest decentralized exchange on Cardano (based on TVL).
- We are behind the popular AdaLite wallet. It was later improved into a multichain wallet NuFi.
- We built the Cardano applications for the hardware wallets Ledger and Trezor.
- We built the first version of the cutting-edge decentralized NFT marketplace Jam On Bread on Cardano with truly unique features and superior speed of both the interface and transactions.

Our auditing team is chosen from the best.

- Talent from esteemed Cardano projects: WingRiders and NuFi
- Rich experience across Google, traditional finance, trading and ethical hacking
- Award-winning programmers from ACM ICPC, TopCoder and International Olympiad in Informatics
- Driven by passion for program correctness, security, game theory and the blockchain technology

B Acknowledgements

We would like to take this space to acknowledge and thank the Project Catalyst and all of you that voted for us in Fund 11. This effort is made real thanks to your support.

Contact us:
audit@vacuumlabs.com