Plasma Leap

A state-enabled computing model for Plasma

Draft v 0.9 - October 9th, 2018
leapdao.org

Abstract

We propose a computing model on Plasma by breaking down smart contracts into smaller programs called spending conditions. EVM-Script, a language enabling developers to write spending conditions is presented and a computation verification game is given to verify any computation executed on Plasma. By encapsulating state into non-fungible tokens a stateful programming model is introduced that enables decentralized application to “leap onto Plasma”. In contrast to other solutions, subjective data availability assumptions do not need to be weakened and only minimal modifications to the Plasma exit game are needed.

Table Of Contents

Introduction

When it comes to scalability, the current capacity of Ethereum network is fairly limited in its transaction throughput, with roughly 15 transactions per second. On August 2017 Vitalik Buterin and Joseph Poon proposed Plasma [1], a novel scaling solution that could enable Ethereum to reach a much higher transaction throughput. Consecutive Plasma proposals have described off-chain venues for simple transfers of fungible and non-fungible tokens. These proposals include Plasma MVP, Plasma Cash and Plasma Debit.

Yet, scaling Ethereum requires not only the transfer, but also the ability to apply rules and conditions to the custody of funds. For any application running on Plasma the developers should be able to define their own rules and conditions. While on Ethereum this task is achieved with smart contracts, on Plasma such functionality is not available at the current state of development.

Despite a number of proposals to integrate the execution of code into Plasma, there are no viable solutions to date. This paper starts by briefly summarizing the difficulties for general computation on Plasma, giving an overview of existing proposals and continues to lay out a potential solution approach. The findings and open issues are discussed in the final section.

Problem Statement

In its essence, Plasma imposes a set of rules on the Plasma operator through crypto-economic incentives. First the operator is bonded with a substantial stake, and then participants are able to submit proofs about invalid state transitions, slashing the operator's stake. The sum of these incentives make the operator a restricted authority, discouraged to create invalid blocks.

Proofs for simple transfers of funds can be generated using Merkle-proofs of transaction inclusion and signature checking. Such proofs indicate the binary attribute of funds being spent or unspent. Arbitrary rules and conditions on transaction spending are significantly harder to proof. Such proofs require the verification of the correct execution of the code that defines the rules and conditions for spending. This becomes the **first part of our problem statement: the introduction of a computing model for rules and conditions governing funds on the Plasma chain as well as the enforcement of correct execution of such code.** Participants need to be able to prove execution of code and penalize the operator for including transaction with such incorrect transitions. The verification needs to be economically feasible inside the Ethereum Virtual Machine (EVM) and should not exceed the capacity of the Ethereum network at any time.

Plasma is based on the assumption of subjective data availability. A subset of the participants can find themselves in a situation where they don't have access to the latest block data. When we introduce contracts into Plasma, funds can have more complex ownership structures, like complex rules and conditions for spending with multiple owners or even no specific owners at all. Exiting such funds requires increased coordination, as their owners might disagree on data availability or there might be no-one with the authority to exit the specific funds. This defines the **second part of the problem statement: Exits can be hard or impossible to coordinate if funds have multiple or undefined ownership.**

To successfully complete a Plasma exit, a transfer should remain unspent for the duration of the challenge period. **The third part of the problem statement describes the coordination problem arising for funds that entered the exit queue**, but are protected with rules and conditions similar to anyone-can-spend. The possibility for a grieving attack emerges where anyone can spend such funds and cancel the exit.

Related work

In this section a short recap of Plasma is given before different approaches to general computation on Plasma are introduced.

Plasma is a technique to do off-chain transactions, in which the transactions are conducted in an external chain environment. The transactions do not need to be replicated across to the Ethereum blockchain, instead, only the resulting balances are exited. Initially, the bridge contract for Plasma is created on Ethereum. It contains the basic rules of the child chain and allows users to move assets between Ethereum and Plasma. The Plasma operator periodically records state hashes claiming the current state of the child chain as valid.

The security of Plasma is assured by the exit procedure, a challenge game through which Plasma guarantees that every party can exit their funds back to the main chain at any given time. The exit game is initiated by any user through submission of an exit transaction. The exit transaction contains a merkle proof of the user's transaction proving ownership of a certain amount of funds [20]. The exiting user also has to attach a small amount of stake, enabling other participants to challenge the user for a certain time, the "challenge period". The challenge mechanism assures that only unspent transfers can exit. After the challenge period is passed, the funds get transferred to the user on the main-net. In case the exit submission is successfully challenged, the exit transaction gets cancelled and the exitors stake is slashed [21].

All states within the Plasma chain are enforced via fraud proofs, which allows for any party to challenge invalid exits, presuming block data availability. However, there are no explicit guarantees around data availability, which causes a significant vulnerability [2]. As missing data can not be used to prove a fraud, the operator can use this security gap to attack the system by creating an invalid block and not publishing the block data. This way the operator can obstruct all other users from fully calculating the state. An approach to circumvent the data withholding attack is the priority queue, prioritizing the exit claims by the age of funds, which ensures transferring of the funds to the correct owner, before the fraudulent access happens [3].

Onther Inc's Approach Of Plasma EVM

Onther Inc develops a solution called Plasma EVM [4], which proposes a state-enforceable Plasma construction to guarantee submission of only the valid state to the root chain. This provides a way of entering and exiting the contract state between Plasma and Ethereum. Plasma EVM uses two types of blocks. RequestBlock manages enter and exit requests and nonRequestBlock (NRB) contains all the other transactions.
The fraud proof is accomplished by a Truebit-like verification game, to resolve validity of the computation by use of solEVM. Onther approaches block withholding attacks by enforcing the submission of block data by the operator to reveal all transactions [5].

As a solution approach to the data unavailability Onther introduces an exit model called userRequestBlock (URB) [6]. Unlike Onther's existing requestBlock, URB contains only transactions that reflect the Exit Request for URB of submitter or other user. This exit model enables a valid exit for users in case of data withholding and makes the individual users do the judgement on data availability. The model estimates the probability of a data unavailability by considering the amount of claimed data availability issues. Additionally a dynamic fee model acts as a security mechanism against the infinite loop attacks with false data availability claims. The fee model adjusts the fees dynamically for the URB and the exit request for the URB through collective judgement data of individual users, i.e. the number of exit requests using URB [7].

Matic’s Generalized State Scaling Approach

For identifying the data involved in fraud challenges, Matic suggests using security deposits as an incentivization mechanism, constituting a mechanism to estimate the side-chain security more effectively, motivating the stakeholders to frequently commit the side-chain blocks to the main chain.

To keep the stakers honest, Matic offers a set of validations, “Insurance contracts”, which in combination should represent a complete set of consensus validation rules on the main chain [8]. These rules are,

- Withholding challenges
- Parsing challenges, i.e. submission of an invalid block structure
- Transaction censorship, which should enforce the inclusion of a main chain transaction in the side-chain within a certain timeframe.
- Invalid block signature
- Invalid previous block hash
- Invalid transaction execution

Matic currently doesn’t offer a solution to potentially costly block withholding challenges, where the operator is forced to reveal the block data.

For enforcing the contract execution, Matic suggests to use a TrueBit-like verification game, to verify EVM state transitions.

Plasmabits’ Approach To Run EVM-based Smart Contracts

Plasmabits is a side-chain implementation that can run EVM-based smart contracts. A centralized operator is responsible of keeping the data availability and correctness of side chain. The validation is enforced using a system of incentivized challenges, and Truebit-like verification for the EVM execution. Using a TrueBit-like binary search, a PASITO contract (Plasma Arbitration Stepping Instruction Test Operator) computes one EVM state transition to verify transaction executions [9].
The solution to data withholding attack is achieved using preimage challenges, in which a validator can request the operator to submit a preimage of the hash of the block. At any successful challenge the Plasma contract tracking the status of the sidechain will halt and stop accepting any further updates. This starts a challenge period in which the invalid or withheld

block is identified. Even in the case of an on-chain challenge, validators can decide to continue to operation.

Our Solution

The section “problem statement” has defined the spending authorization problem, a coordination problem arising from funds held under programmable rules and conditions of spending. If the operator is able to spend the latest state of a contract during or after data withholding, it will be able to push invalid exits before the exit of such funds, opening the chance for theft.

All related work on smart contracts-enabled Plasma chains addresses this spending authorization problem by proposing schemes that alleviate the data availability problem in the hope to challenge invalid transitions by the operator. These approaches often introduce additional actors or costly ways to force the operator to reveal block data without realizing the hopeless nature of the effort. During the data-withholding period the operator can introduce a vast amount of data, forcing the prover into a sisyphus task.

Our solution contrasts with these approaches, by maintaining the assumption of subjective data availability. The solution breaks down smart contracts into smaller programs called spending conditions and into single state objects with clearly defined owners. State objects can be exited with minimal modifications to the exit game. Hence, the solution does not need to impose the existence of additional potentially corruptible actors or data-reveal games.

Spending Conditions

Spending conditions are scripts providing the ability to apply specific rules and conditions to the transfer of funds. To understand spending conditions, we recap Pay-to-Script-Hash (P2SH) transactions for Bitcoin: P2SH is a transaction type in Bitcoin which allows transactions to be sent to a script hash instead of sending to a public key. Bitcoins locked under P2SH can only be transferred by providing a script matching the script hash. In addition, the transaction needs to carry data which makes the script evaluate to true [10].

With these transaction types Bitcoin locks funds into programmable conditions. Transactions with the right input data can fulfill such conditions, and hence unlock the funds to lock them again under new conditions. This is how funds move through time on the Bitcoin blockchain.

Programmable conditions can be created with EVM bytecode, and the use of such conditions to govern the spending of funds gives them their name - “spending conditions”. Spending conditions are smart-contract-like scripts. Yet, unlike smart contract spending conditions shall not affect arbitrary state. This would make the exit game for any spending condition unfeasible [11]. Rather, the output of spending conditions execution should not be affected by storage or other transfers. The inputs and outputs of the transaction holding the spending condition are the only permitted side effects.

This section describes how the UTXO model can be updated from simple signature checking to the use of spending conditions.

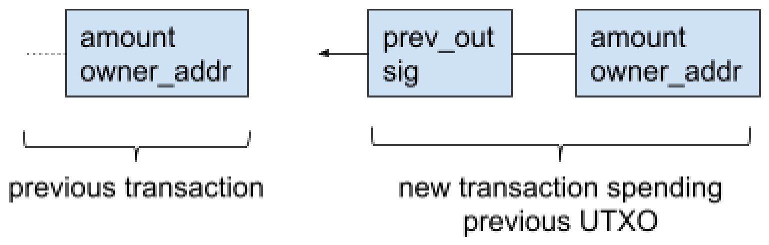

Fig. 1. MVP Plasma model

Currently funds on Plasma are protected by assigning them to the owner's public key in an UTXO. A spending transaction needs to provide a signature by the corresponding private key to release the funds [Figure 1].

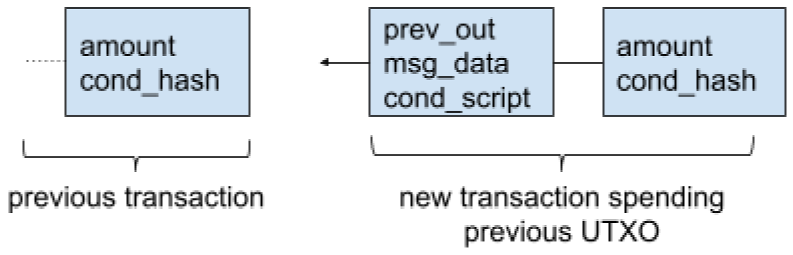

Fig. 2. Spending condition model

Spending conditions can be integrated into the UTXO model by assigning the ownership of funds to the hash of the spending condition. The spender then needs to provide an input which contains the script and call data. The script will be invoked with the call data during execution.

On Plasma spending conditions are evaluated in a verification environment with the following steps:

1. The spending transaction is parsed and the hash of the script is compared to the condition hash found in the output.
2. The spending condition is deployed as a contract.

3. An ERC20 contract is deployed alongside in the environment. The amount of tokens stated in the output is minted and assigned to the address of the spending condition contract.
4. The spending condition is invoked with the message data that is given in the input of the spending transaction.
5. The transfer events in the ERC20 contract are read. The number of events has to match the number of outputs in the spending transaction and the data of the events have to match the outputs' in amounts and destinations.
6. Any remaining funds on the account of the spending condition are considered transaction fees and are burned.

Spending conditions are intentionally designed to deliver the same results if they are executed in the virtual environment described above to evaluate the spending of UTXO's as well as on the Ethereum main network, after they have been exited from the Plasma chain.

EVM-Script

EVM-Script is a subset of the EVM instruction set excluding the following OP-codes: SSTORE, SLOAD, CREATE, SELFDESTRUCT. Hence, spending conditions can be implemented in any language that compiles down to EVM. A Solidity example is depicted next.

```
contract SpendingCondition {
    address constant spenderAddr = 0xF3beAC30C498D9E26865F34fCAa57dBB935b0D74;

    function fulfil(bytes32 _r, bytes32 _s, uint8 _v,      // signature
        address _tokenAddr,                               // inputs
        address[] _receivers, uint256[] _amounts) public {  // outputs
        require(_receivers.length == _amounts.length);

        // check signature
        address signer = ecrecover(bytes32(this), _v, _r, _s);
        require(signer == spenderAddr);

        // do transfer
        ERC20Basic token = ERC20Basic(_tokenAddr);
        for (uint i = 0; i < _receivers.length; i++) {
            token.transfer(_receivers[i], _amounts[i]);
        }
    }
}
```

Fig. 3. Spending conditions

In its simplest form a spending condition should consist of a constant and a function that takes a signature and matches its signer against the constant. By this, a spending condition allows only the owner of a private key matching the address to transfer the funds it is protecting. When the spending condition is executed the inputs of the transaction are injected into the execution environment as ERC20 and ERC721 contract instances.

Enforcing correct off-chain execution

Jason Teusch et al introduces the TrueBit [12], a protocol to verify correct execution of computation in witness of a computationally bounded judge. The computation in question can be larger than the bandwidth of the judge. A verifier initiates the judging process or “verification game” that allows to determine the validity of a solver’s solution. The judge then interacts with the two agents, solver and verifier, to verify the validity of the computation task at hand, without the need to execute the whole computation.

To understand the protocol better, it is suggested to imagine a program execution rolled out into single steps [13]. The program is executed in a virtual machine (VM). The VM consists of various elements which change their data after each single step (OP-Code) of the program. These intermediate states can be compared in an deterministic manner, by hashing state of the VM’s elements into a single root hash. The solver and the verifier agree on the state of the VM that executes the program at the step 0, but they have produced different results at the end of the execution.

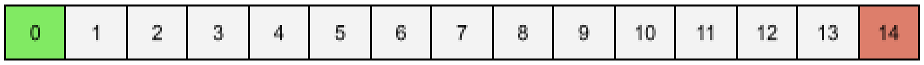

Fig. 4. Step 0 of verification game

- The judge queries the solver and the verifier for the state of the VM at the halfway mark of the execution.

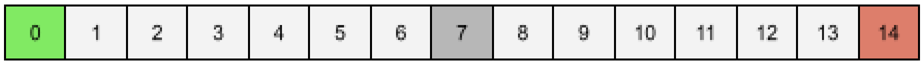

Fig. 5. Step 1 of verification game

- If the solver and the verifier agree on the state of the program execution at the halfway mark, they can deduce that the disagreement occurred in the second half of the execution.

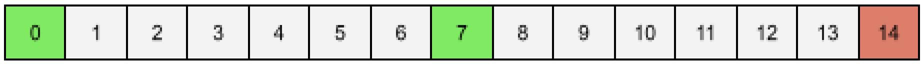

Fig. 6. Step 2 of verification game

- Continuing the binary search, the solver and the verifier are queried for the state of the VM at the halfway mark of the second half of the execution.

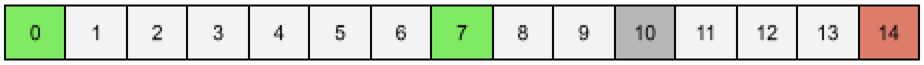

Fig. 7. Step 3 of verification game

- If the solver and verifier again find that their states match, they have to conclude that the dispute is in the last quarter of the execution.

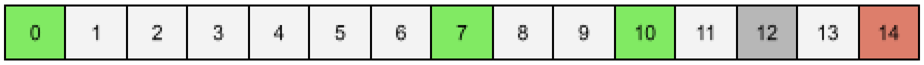

Fig. 8. Step 4 of verification game

- The search goes until the solver and verifier identified two consecutive execution steps where they agree on the state of the VM at the first step, but disagree at the second.

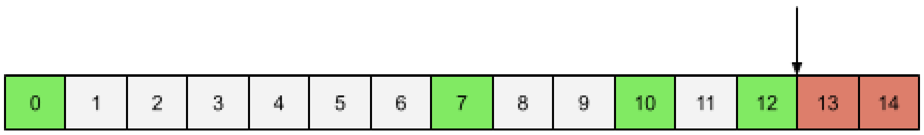

Fig. 9. Step 5 of verification game

This is when the state is handed back to the judge contract that has enough computational bandwidth to execute the single operation, to decide the dispute. The judge loads the state of the VM at step 12 and executes the next OP-code. The judge then compares the result of this computation with the state hash given by the solver. Two outcomes are possible:

- Computation result of the judge matches to the state hash of the solver, at step 13, the challenge is rejected, the computation results of the solver are accepted.
- Computation results of the judge does not match the state hash of the solver, at step 13, the challenge is successful, the solver's result is discarded.

SolEVM

In the previous section the Truebit verification game has been introduced. This game can reduce the verification effort of off-chain execution to the execution of a single operation on-chain. SolEVM is a partial implementation of the Ethereum runtime running in the EVM. It is the component that executes this last operation on-chain and decides the winner of the verification game. Thus the Truebit verification game and SolEVM together assure the correct execution of spending conditions.

To enable it, the following requirements need to be met by the implementation to make the verification of Plasma computation feasible:

1. A single computation step on SolEVM needs to run within the Ethereum gas limit.
2. The space required to describe a single state change should fit into a single transaction on Ethereum.

An implementation meeting the first requirement has been started based of the work of Andreas Olofsson on SolEVM [14] and extended with the ability to run single steps and compare state [15].

The satisfiability of the second requirement is currently in research. Difficulties arise from the fact that single computation steps (OP-codes) interpreted by the EVM often access multiple dynamic data structures. An overview over the different OP-codes and their data access size can be found in the architecture document [16].

SolEVM operates in 2 modes to enable the verification game.

- **Off-chain interpreter** - enumerates a list of states for a given computation.
- **On-chain stepper** - given a state, can compute the next state.

Solvers and verifiers use the interpreter to create a list of states for a computation and start the verification game. Once they have determined the computational step at which they disagree, the solver uses the stepper to run the disputed step and resolve the outcome of the verification game.

By introducing EVM-Script as a language to write spending conditions, and solEVM as a runtime to verify the correct execution of spending conditions we have created the ability to enforce off-chain computation.

Verification Incentivisation

The protocol described in the previous section requires an incentivisation layer for verifiers to engage in the protocol and challenge invalid computation delivered by the solver. If this incentivisation game fails, or has skewed incentives, then the state of the Plasma chain might end up invalid, which will allow an attacker to steal funds through the exit procedure.

The bare verification game where everyone can challenge any state transition included in a block would open up a DDoS attack on the operator of the chain, as there would be no cost associated with claiming a valid computation wrong. Consequently, we need to introduce the requirement for the verifier to be bonded before they can issue the challenge. Their bond will be slashed, in case the challenge is not successful. This prevents the previously described DDoS vulnerability.

However, the requirement for bonding introduces further problems to the incentive structure of the protocol, as the size of the bond might deter users from engaging as verifiers. In contrast to the TrueBit incentive layer, the users in the Plasma chain have an inherent incentive to check on the operator because their funds are held by the Plasma chain which the operator manages.

Exiting Spending Conditions

The More Viable Plasma exit game requires the owner of funds to initiate the exit and after the challenge period the funds are transferred to the owner address. When introducing Spending Conditions to the picture, challenges arise in the authorization to the exit function, as well as to the custody of the funds after the challenge period.

```
contract SpendingCondition {
    address constant spenderAddr = 0xF3beAC30C498D9E26865F34fCAa57dBB935b0D74;

    function exitProxy(
        bytes32 _r, bytes32 _s, uint8 _v,   // authorization to start exit
        address _bridgeAddr,                // address of Plasma bridge
        bytes32[] _proof, uint _oindex      // tx-data, proof and output index
    ) public {
        address signer = ecrecover(bytes32(this), _v, _r, _s);
        require(signer == spenderAddr);
        PlasmaInterface bridge = PlasmaInterface(_bridgeAddr);
        bridge.startExit(_proof, _oindex);
    }

    function fulfil(...);
}
```

Fig. 10. Exiting spending conditions

The above snippet shows a Spending Condition with an additional function ***exitProxy()***. The function first verifies that an authorized caller is invoking it. If authorized, the function forwards the call with the provided transaction data, position and proof to the Plasma bridge. The implementation of the exit authorization within the Spending Condition gives the developer the flexibility to define a custom exit authorization scheme for each Spending Condition.

```
contract PlasmaBridge is PlasmaInterface {
  using TxLib for TxLib.Outpoint;
  using TxLib for TxLib.Output;
  using Reflectable for address;

  event ExitQueueMock(bytes32 txHash);

  function startExit(bytes32[] _proof, uint _oindex) {
    bytes32 txHash;
    bytes memory txData;
    (, txHash, txData) = TxLib.validateProof(32, _proof);
    // parse tx and use data
    TxLib.Output memory out = TxLib.parseTx(txData).outs[_oindex];
    // check that caller is owner
    if (msg.sender != out.owner) {
        // or caller code hashes to owner
        require(bytes20(out.owner) == ripemd160(msg.sender.bytecode()));
    }
    emit ExitQueueMock(txHash);
  }
}
```

Fig. 11. startExit function

The above snippet shows that minimal modifications are needed to the ***startExit()*** function as it is known from More Viable Plasma. Rather than restricting the call only to the owner of funds, here the hash of the code of the calling contract might also match owner address of the exiting funds.
With these building blocks in place we can define an exit procedure for Spending Conditions as follows:

1. Contract with code of spending condition is deployed on the main chain.
2. The spending condition implements a special function called ***exitProxy()***. Developers can define the ***exitProxy()*** to limit the access to the exit of the tokens the condition protects.
3. The ***exitProxy()*** function gives the ability to register any tokens, which it holds on Plasma, for exit.
4. The ***startExit()*** function of the Plasma contract will load the code of the spending condition and compare its hash with the hash found in the output of the transaction that is registered for exit.
5. The exit is conducted as known. After the exit period, the token is transferred to the address of the spending condition.

After successful exit the tokens are held by the same spending condition on Ethereum as it was before the exit on Plasma. The spending condition can now be fulfilled to release the tokens.

State Objects

In [11] it has been discussed how contract state prevents the creation of an effective exit game for smart contracts from Plasma. With spending conditions introduced in the previous section a computing model for Plasma has been proposed that can be enforced on the root chain. This model circumvents the exit problems found in [11], by limiting the expressiveness of Spending Conditions to interact with the state of tokens, but not arbitrary contract state.

However, stateless scripting has already been possible in Bitcoin, and has not spurred a cambrian explosion of decentralized applications as seen on Ethereum. The reason for this being that turing completeness is not the crucial factor to enable smart contracts. Rather, rich statefullnes, the ability to store state across multiple invocations of a program, enables the creation of contracts and decentralized applications [17]. Kelvin Fichter [18] has layed out that state has to follow the requirement of clear ownership to be exit-able in the context of Plasma, and coined the term state object.

Non-fungible Storage Token (NST)

We propose to extend non-fungible tokens (NFTs) with the ability to store data to enable state objects. This proposal wraps a store of data with the attributes that are required to safely move it through the Plasma lifecycle of deposit, transfer and secure exit. Such a token we call non-fungible storage token (NST).

```
contract StorageToken is ERC721Token {

  mapping(uint256 => bytes32) public data;

  function read(uint256 _tokenId) public view returns (bytes32) {
    return data[_tokenId];
  }

  function verify(
    uint256 _tokenId,     // the token holding the storage root
    bytes _key,           // key used to do lookup in storage trie
    bytes _value,         // value expected to be returned
    uint _branchMask,     // position of value in trie
    bytes32[] _siblings   // proof of inclusion
  ) public view returns (bool) {
    require(exists(_tokenId));
    return tree.verifyProof(data[_tokenId], _key, _value, _branchMask, _siblings);
  }
  function write(uint256 _tokenId, bytes32 _newRoot) public {
    require(msg.sender == ownerOf(_tokenId));
    data[_tokenId] = _newRoot;
  }
}
```

Fig. 12: Non-fungible Storage Token (NST)

In figure 12 an NFT is shown, which is extended with the ability to store state. A single attribute per token is added which is the storage root of a Merkle Patricia Tree. An authenticated function to update the root is provided and a view-only function allows to verify the existence of a key-value pair according to the storage root.

Deposit and Exit

NFTs fit well into the More Viable Plasma [19] lifecycle of deposit, transfer and exit. Deposits of non-fungible tokens create an Event in the Plasma Bridge contract, which lead to an output being added to the state of the Plasma chain. The output different from a fungible token in that instead of an amount the tokenId is held and in case of NSTs, and additional field of the storage root is added.

The exit procedure does not require a priority queue, as the supply of tokens can not be inflated, and every attempted theft will have the current owner of the token as a victim. Here a challenge period with the same duration as for fungible tokens is enough before the token is dispensed to its owner.

Reading and Writing NSTs on Plasma

Given a Plasma chain implementing the presented storage token interface, spending conditions with the ability to utilize state can be constructed. We can distinguish the following two modes of usage.

- Data read from storage token owned by users on chain.
- Data read and written from storage tokens that are owned by a spending condition.

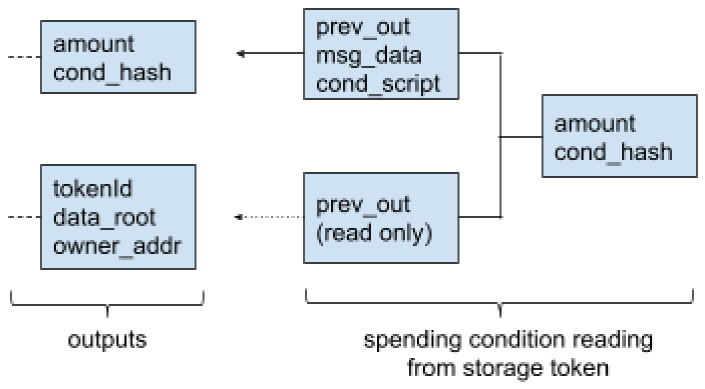

Fig. 13. Reading Storage Tokens

Figure 13 shows a transaction referencing a non-fungible storage token in it's second input. When the transaction will be processed, the execution environment will instantiate a ERC721 contract and mint a token with the according data. The spending condition contract will be able to read the storage root from the token and use this data in it's execution. The output is referenced as read-only, hence the application of the transaction to the UTXO state of the Plasma chain will not consume the NFT output.

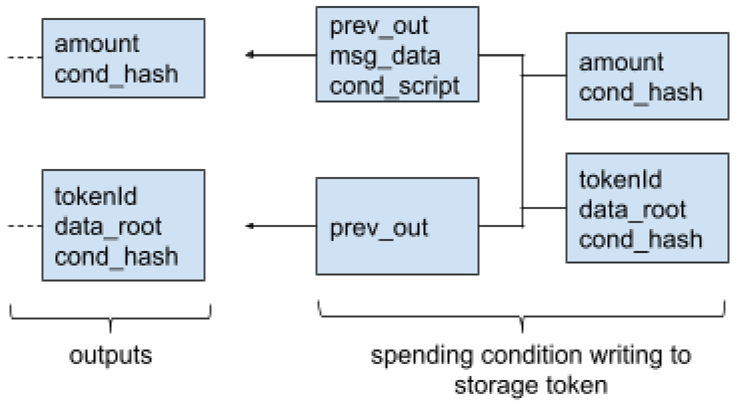

Fig. 14. Writing to Storage Tokens

Figure 14 depicts a spending condition which owns a storage token. The hash of the spending condition is set as the owner of the output. Given such access right, the spending condition can write to the storage and update the storage root of its token. An update of the token data will spend the output and create a new output with the updated root in the Plasma state.

Discussion

By changing our perspective, and considering fungible and non-fungible tokens the only first-class citizens on the Plasma chain, a stateful computation model that complies with the knows exit games for Plasma is possible. Yet, limitations are imposed through the constraints of the Plasma. The assumption of subjective data availability limits the freedom of developers in implementing the fulfilment of conditions. The need to add an exit-authorization function to every conditions also puts a limit on the number of parties collaborating in a contract.

Application Examples

Let's have a look at a few application examples that become possible using this model.

```
import "openzeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol";
import "./StorageTokenInterface.sol";

contract CounterCondition {
    uint256 constant tokenId = 1234;

    function fulfil(address[] _tokenAddr,                // inputs
        address _receiver, uint256 _amount) public {    // outputs

        // update counter
        StorageTokenInterface stor = StorageTokenInterface(_tokenAddr[1]);
        uint256 count = uint256(stor.read(tokenId));
        stor.write(tokenId, bytes32(count + 1));

        // do transfer
        ERC20Basic token = ERC20Basic(_tokenAddr[0]);
        if (count < 4) {
            require(_receiver == address(this));
        }
        token.transfer(address(_receiver), _amount);
    }
}
```

Fig. 15. Counter condition

One of the simplest example one can create in a stateful programming paradigm is a counter. The above spending condition demonstrates such an implementation. This condition does not check the signature of the spender, but rather makes use of a NFT storing the number of successful spends of the condition to itself. On the 5th spend of the condition the spender is free to choose a destination address of his liking.

```
import "openzeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol";
import "./StorageTokenInterface.sol";

contract MultisigCondition {
    uint256 constant alice = 123;   // storage token owned by alice
    uint256 constant bob = 456;     // storage token owned by bob
    uint256 constant charlie = 789; // storage token owned by charlie
    uint256 constant threshold = 2;

    function fulfil(address[] _tokenAddr,               // inputs
        address _receiver, uint256 _amount) public {    // outputs

        // check condition
        uint256 haveAgreed = 0;
        StorageTokenInterface stor = StorageTokenInterface(_tokenAddr[1]);
        haveAgreed += address(stor.read(alice)) == _receiver ? 1 : 0;
        haveAgreed += address(stor.read(bob)) == _receiver ? 1 : 0;
        haveAgreed += address(stor.read(charlie)) == _receiver ? 1 : 0;
        require(haveAgreed >= threshold);

        // do transfer
        ERC20Basic token = ERC20Basic(_tokenAddr[0]);
        token.transfer(address(_receiver), _amount);
    }
}
```

Fig. 16. Multisignature wallet spending condition

Another example for a spending condition is a stateful multisignature wallet. In this example Alice, Bob and Charlie combine state that is stored in their personal storage tokens. Once two of the three participants update their storage tokens to point to the same destination address, the funds held under the multisignature spending condition can be transferred.

Open Issues

By changing perspective, and considering fungible and non-fungible tokens the only first-class citizens on the Plasma chain, a stateful computation model that complies with the known exit game is possible. Yet, limitations are imposed by the constraints of the Plasma design. The assumption of subjective data availability limits the freedom of developers in implementing the fulfillment of conditions. The need to add an exit-authorization function to every condition also puts a limit on the number of parties collaborating in a contract.
In addition to the mentioned challenge, we see the following limitations and open issues with Plasma Leap:

- Ability to fit OP-codes with dynamic data size into blockgaslimit is unclear.
- Do computation verification incentives also capture the long tail (holders with small balances) in a single operator model?
- Deeper analysis on limbo exits under MoreVP is needed.
- How to provide compact witnesses to state updates of NSTs?
- UX challenges inflicted by liveliness assumption of L2 solutions in general.
    - inability to build watch-towers for exits.

Conclusion

Plasma allows for the creation of child chains attached to Ethereum, while handling much higher data throughput, reducing the costs and resource usage. This opens up the possibilities of building ecosystems running decentralized applications with many thousands of concurrent users.

We have created a computing model that allows to put tokens under the control of programmable conditions, and incentive games that enforce the correct execution of these conditions off-chain by the Plasma operator. Further we have extended the exit game with the ability to exit such spending conditions and associated tokens to the Ethereum network. Lastly, we have extended the computing model with the ability to store state across invocations.

Nevertheless, smart contracts as we know them on Ethereum are just one way to enable decentralized applications. With this architecture we hope to have considerably widened the scope of decentralized applications that can be implemented on layer-2 scaling solutions.

Acknowledgements

We thank the Ethereum Foundation for funding this research and parts of the implementation with a grant.

References

[1] http://plasma.io/
[2] https://plasma.io/plasma.pdf
[3] https://github.com/ethereum/research/wiki/A-note-on-data-availability-and-erasure-coding
[4] https://ethresear.ch/t/plasma-evm-state-enforceable-construction/3025
[5] https://hackmd.io/s/HyZ2ms8EX
[6] https://ethresear.ch/t/data-availability-solution-for-plasma-evm/3294
[7] https://hackmd.io/s/ByeGtM5D7

[8] https://whitepaper.matic.network/#challenges
[9] https://ethresear.ch/t/plasmabits-viable-stateful-sidechain/2344
[10] https://en.bitcoin.it/wiki/Pay_to_script_hash
[11] https://ethresear.ch/t/why-smart-contracts-are-not-feasible-on-plasma/2598
[12] https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf
[13] https://medium.com/truebit/truebit-the-marketplace-for-verifiable-computation-f51d1726798f
[14] https://github.com/Ohalo-Ltd/solevm
[15] https://github.com/leapdao/solEVM-enforcer
[16] https://github.com/leapdao/solEVM-enforcer/blob/master/docs/Architecture.md
[17]
https://np.reddit.com/r/btc/comments/6ldssd/so_no_worries_ethereums_long_term_value_is_stil
l/djt6opz/
[18] https://medium.com/@kelvinfichter/why-is-evm-on-plasma-hard-bf2d99c48df7
[19] https://media.consensys.net/the-state-of-plasma-1-6b48c1e4b295
[20] https://www.learnplasma.org/
[21] http://cryptoeconomics.study