This is a proposed extension to More Viable Plasma. It is inspired by conversations at the solEVM calls with @jdkanani, @esteban and @mrsmkl.

tl;dr

We propose a computing model on Plasma by breaking down smart contracts into smaller programs called Spending Conditions. A computation verification game is given to verify any Spending Condition executed on Plasma. By encapsulating state into non-fungible tokens a stateful programing model is introduced that enables “Dapps to leap onto Plasma”

. To achieve this, subjective data availability assumptions do not need to be weakened and only minimal modifications to the Plasma exit game are needed.

Problem Statement

Enforcing off-chain computation -

a computing model for rules and conditions governing funds on the Plasma chain is required, 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 [tb].

Exit authorization -

if funds are controlled by complex rules and conditions for spending with multiple owners or even no specific owners, these owners might not agree on the availability of data or there might be no-one with the authority to exit the specific funds. Thus, exits can be hard or impossible to coordinate if funds have multiple or undefined ownership [kf].

Spending authorization -

a coordination problem arises 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 [jb].

Spending Conditions

Spending conditions are scripts providing the ability to apply specific rules and conditions to the transfer of funds, similar to Bitcoin P2SH. Spending conditions are smart-contract-like scripts written in Solidity. Yet, unlike smart contract spending conditions shall not affect arbitrary state. This would make the exit game for any spending condition infeasible, as described in [why smart contracts are not feasible]. Rather, the output of spending conditions execution should not be affected by storage or other transfers. The inputs and outputs of the transaction fulfilling the spending condition are the only permitted side effects.

contract SpendingCondition { using Reflectable for address; uint256 constant nonce = 1234; // nonce, so that signatures can not be replayed 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(ripemd160(address(this).bytecode())), _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]);
    }
}
```

}

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.

Spending conditions are intentionally designed to deliver the same results if they are evaluated in the UTXO model of Plasma or as smart contracts, after they have been exited from the Plasma chain onto Ethereum.

Solidity Script -

A subset of the Solidity Language excluding the following OP-codes: SSTORE, SLOAD, CREATE, SELFDESTRUCT. Using this language spending conditions can be developed and applied on Plasma.

Computation Verification Game

The TrueBit protocol proposes a scheme to verify correct execution of computation in witness of a computationally bounded judge. The protocol can reduce the verification effort of off-chain execution to the execution of a single operation on-chain. While the original protocol works with WASM VM we adapt the protocol to be able to verify EVM computation usingSolEVM. To enable it, the following requirements need to be met by the implementation to make the verification of Plasma computation feasible:

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

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 Solidity 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.

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 { using Reflectable for address; uint256 constant nonce = 1234; // nonce, so that signatures can not be replayed 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(ripemd160(address(this).bytecode()), _v, _r, _s);
  require(signer == spenderAddr);
  PlasmaInterface bridge = PlasmaInterface(_bridgeAddr);
  bridge.startExit(_proof, _oindex);
}
function fulfil() public;
```

}

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); }

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, the hash of the code of the calling contract might also match owner field 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 function 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 are held under the hash of the spending condition 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 being 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 they have been held by before on Plasma. The spending condition can now be fulfilled to release the tokens.

State Objects

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. @kfichter 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.

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 is called non-fungible storage token (NST

).

contract StorageToken is ERC721Token, StorageTokenInterface { PatriciaTree tree; mapping(uint256 => bytes32) 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; } }

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.

Application Examples

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

contract CounterCondition { using Reflectable for address; uint256 constant tokenId = 1234; address constant spenderAddr = 0x1234;

```
function fulfil(bytes32 _r, bytes32 _s, uint8 _v,   // signature
    address[] _tokenAddr,                           // inputs
    address _receiver, uint256 _amount) public {    // outputs

    // check signature
    address signer = ecrecover(bytes32(ripemd160(address(this).bytecode())), _v, _r, _s);
    require(signer == spenderAddr);

    // 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);
}

}
```

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 makes use of a NST 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.

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);
}

}
```

Another example for a spending condition is a stateful multi-signature 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 multi-sig spending condition can be transferred.

Discussion

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.
- inability to build watch towers for exits.

Conclusion

We have created a computing model that allows to put tokens under the control of programmable conditions. We have created 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.

Smart contract as known 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 layers-2 scaling solutions.

Note:

I am looking for co-authors and reviewers of thefull version.