Plasma EVM

Carl Park[1], Aiden Park[1], and Kevin Jeong[1]

[1]Onther Inc.

May, 2019

Abstract

이더리움의 EVM(Ethereum Virtual Machine)은 블록체인에서 튜링 완전한 연산을 지원함으로써 스마트 컨트랙트(Smart contract)라는 이름으로 잘 알려진 일반 프로그램(General Program)을 동작가능하게 하였다. 플라즈마 EVM은 완전히 일반화된 플라즈마로, 플라즈마 체인에서 EVM을 실행할 수 있는 새로운 형태의 플라즈마이다. 때문에 go-ethereum, py-evm, parity와 같은 현재 이더리움 클라이언트에 기반을 두고있다. 이 페이퍼에서는 자식체인의 검증된 상태만이 루트체인에 확정(Finalize)되는 것을 보장하기위해 TrueBit-like Verification Game을 이용한 상태 검증과 두 체인 사이의 Account storage에서 Enter, Exit하는 방법을 제공하여 상태가 강제되는(State-enforceable) 플라즈마 구조를 제안한다. 이는 두 체인이 완전히 동일한 구조를 갖기 때문에 가능하다. 플라즈마 EVM은 기존 이더리움 체인에 구축된 탈중앙화 어플리케이션(Decentralized Application; Dapp)을 그대로 플라즈마 체인으로 옮겨와 Dapp의 탈중앙성, 성능, 안정성, 사용성 모두를 개선시킬 수 있다.

1 Introduction

Plasma EVM은 크게 *RootChain contract*와 오퍼레이터(Operator), 사용자, 그리고 *Requestable contract*로 구성된다. 오퍼레이터는 자식 체인(Child chain)을 운영하면서 블록을 마이닝하여 Plasma EVM을 관리하는 *RootChain contract*에 제출(Submit)하고, 사용자는 오퍼레이터로부터 블록 데이터를 받아 개별적인 노드를 운영할 수 있다. *Requestable contract*는 자식 체인의 상태 전이를 강제하는 *Request*를 반영할 수 있는 컨트랙트이다. 사용자는 *Requestable contract*의 상태를 루트 체인(Root chain)과 자식 체인간에 이동시키기 위해 *Request*를 생성하고, 오퍼레이터는 블록을 마이닝하여 자식 체인의 상태를 해당 *Request*에 따라 변경한다. *Request*를 반영하는 것은 *RootChain contract*에 의해 강제되며 만약 해당 *Request*가 올바르지 않게 반영된다면 *Challenge* 당한다. 또한 Continuous Rebase를 통해 Data availability 문제를 해결한다.

2 Related Works

Plasma MVP Plasma MVP는 1 이후 가장 먼저 나온 Plasma 구현체이다. Minimal Viable Plasma 라는 이름에 맞게 아웃풋이 2개뿐인 트랜잭션과 해당 UTXO를 이용하여 자식 체인의 상

태를 표현(represent)한다. 오퍼레이터의 Invalid Block 제출과 Data Withholding Attack에 대비하기 위하여 유저는 블록에 자신의 TX가 올바르게 포함되었을 경우 다시 한번 서명(confirm)하여 자신의 트랜잭션을 확정(Finalize)할 수 있다. UTXO의 Finality가 Confirmation signature에 의해 보장 되기에, 사용자는 Exit Game을 통해 자식 체인의 UTXO를 올바르게 루트 체인으로 꺼낼 수 있다.

Plasma Cash Plasma Cash는 Plasma MVP의 단점이었던 Confirmation signature을 대체하기 위하여 자식 체인의 상태(UTXO)를 Sparse Merkle Tree를 이용하여 표현한다. Exit Game은 SMT의 inclusion proof를 이용하여 해당 트랜잭션이 1) double spending 인 경우, 2) double spending하는 UTXO에 기반한 트랜잭션인 경우, 3) 소유권이 없는 UTXO를 Exit하려는 경우를 방지할 수 있다.

More Viable Plasma Plasma MVP에서 Exit priority에 대한 기준을 UTXO의 age of output으로 산정했다. More Viable Plasma은 Exit priority를 age of the youngest input으로 산정하고 Exitable outputs를 재정의함으로써 Confirmation signature를 제거하고 기존의 Exit game의 안전성을 유지할 수 있음을 증명하였다.

Plasma Leap Plasma Leap은 스마트 컨트랙트의 동작을 작은 단위로 재정의한 Spending Conditions와 상태를 저장하는 Non-fungible storage token(NST)을 이용하여 구현된 부분적으로 일반화된(Partially generalized) 플라즈마 모델이다. Spending Conditions의 올바른 실행 여부를 검증하기 위해 Plasma EVM과 동일한 Truebit-like verification game을 사용한다. 또한 More Viable Plasma의 Exit game을 그대로 사용할 수 있기 때문에 Data availability 또한 쉽게 해결이 가능하다. 단, Plasma EVM과 달리 상태를 Account-based가 아닌 UTXO로 표현하고 실행가능한 EVM opcode의 수를 줄였기 때문에 이더리움과 동일한 수준의 기능을 지원하는 것은 불가능하다.

3 Plasma EVM

Plasma EVM 클라이언트는 기존 이더리움 클라이언트에 기반하기에 대부분의 메커니즘은 이더리움 클라이언트과 동일하다. 자식 체인의 블록, 트랜잭션 그리고 Receipt등의 자료구조는 이더리움과 같다. 다만 기존의 이더리움과 다르게 일반화된 플라즈마 프로토콜을 적용하기 위해 몇가지 변경점이 존재한다.

Rootchain Network Assumption Ethereum은 PoW(Proof of Work)로 인해 Chain reorganization 혹은 Network congestion이 발생할 수 있다. 하지만 본 글에서는 Ethereum network가 위와같은 상황에 있지 않다고 가정한다.

3.1 3 Types of Block and Epoch

Plasma EVM에는 *Non-Request Block*, *Request Block*, *Escape Block* 등 다양한 유형의 블록이 있다. *Non-Request Block(NRB)*는 사용자의 일반적인 트랜잭션을 포함하는 블록이다. Ethereum, BitCoin, 그리고 다른 블록체인의 블록과 동일하다.

*Request Block(RB)*은 *Request*를 자식 체인에 적용하기 위한 블록이다. 사용자는 *Request*를 생성하고 오퍼레이터가 *RB*를 생성하여 *Request*를 자식 체인에 적용한다. 이는 *RootChain contract*가 해당 *RB*의 *transacctionsRoot*를 지정하므로 어느 트랜잭션이 *RB*에 들어가야하는지 강제할 수 있다.

*Escape Block(EB)*은 사용자가 오퍼레이터에 의한 Block Withholding 공격에 대비하기 위한 일종의 *RB*이다. *Escape*과 *Rebase*의 자세한 내용은 3.5 항목을 참조하라.

*Epoch*은 여러 블록들을 포함하는 주기이다. 블록의 타입에 따라 *Epoch*또한 *Non-Request Epoch(NRE)*, *Request Epoch(RE)*, *Escape Epoch(EE)*으 구분된다. *Epoch*의 길이는 해당 *Epoch*에 포함되는 블록의 갯수이며, *NRE*의 길이는 사전에 정의된 상수로 변하지 않는다. 하지만 *RE*와 *EE*의 경우 유저가 *Request* 혹은 *Escape request*을 생성할 경우 길이가 가변적으로 늘어나거나 줄어들 수 있다. 단, Genesis 블록은 0번째 *NRE*에 포함되며, *NRE#0*의 길이는 항상 1 이다.

*RootChain contract*는 어느 종류의 *Epoch*이 배치되어야 하는지 지정함으로써 자식 체인의 상태 변화를 강제한다. *NRE*를 통해 자식 체인에서 발생한 트랜잭션을 허용한다. *Request*가 생성될 경우 *NRE* 이후에 *RE*가 오도록 배치하고, *Escape request*이 생성될 경우 *EE* 이후에 기존의 *NRE*와 *RE*가 *Rebase* 되도록 한다.

3.2 Block Mining

*RootChain contract*는 기본적으로 다음과 같은 순서로 Epoch을 배치시킴으로써 자식 체인의 트랜잭션과 루트 체인에서 생성된 Request을 반영할 수 있다.

1. *NRE#0* 이후에 *NRE#1*이 위치한다.
2. *NRE#N* 이후엔 항상 *RE#(N+1)*이 위치한다. 마찬가지로 *RE#N* 이후엔 *NRE#(N+1)*이 위치한다.
3. 현재 *Epoch*이 *NRE#N* 혹은 *RE#(N+1)* 일 때 생성된 *Request*은 *RE#(N+3)* 에 반영된다. 만약 *Request*이 생성되지 않는다면, 해당 *RE*의 길이는 0이다.

*RootChain contract*는 위처럼 두 종류의 블록을 제출 받기 위해 주기적으로 *Accept NRB*, *Accept RB* 상태로 변한다. *Accept NRB*는 오직 *NRB*만 제출될 수 있는 상태이며, *Accept RB*는 오직 *RB*만 제출될 수 있는 상태이다. 오퍼레이터는 *RootChain contract*의 상태에 따라 *Request block* 또는 *Non-request block*를 제출해야 한다.

Figure 1는 이러한 *RootChain contract*의 상태가 어떻게 변경되는지 보여준다. 단, 여기서 다루는 내용은 Plasma EVM의 Continuous Rebase를 포함하지 않는다. 전체적인 동작 과정에 대한 내용은 이후의 섹션 3.5에서 다룬다.

3.3 Block Submission

오퍼레이터는 각 블록마다 세 종류의 머클루트 값인 *stateRoot*, *transactionsRoot*, *receiptsRoot*를 제출해야 한다. 단, *transactionsRoot*만 제출하는 것이 가능한 상황도 있는데, 자세한 내용은 섹션 3.5에서 다룬다. 만약 상태 전이에 대한 연산이 제대로 실행되지 않아 올바르지 않은 머클루트값

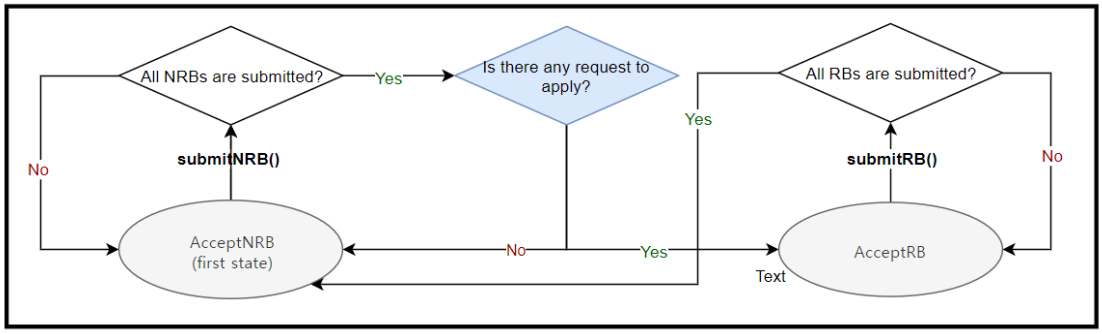

Figure 1: Simplified state transition of RootChain contract

이 제출되었다면, *Computation Challenge*를 통해 올바른 상태를 가지고 있는 블록으로 복구 될 수 있다. 이 방식은 TrueBit의 verification game과 유사한 방식이며, *preStateRoot* 및 *postStateRoot*를 사용하여 '블록 별 상태 전이 함수'를 확인하는 방식으로 해결한다.

3.4 Apply Request

*Request*은 사용자가 *RootChain contract*에 트랜잭션을 보냄으로써 생성된다. *Enter request*와 *Exit request*은 *RB*에 포함되며, *Escape request*과 *Undo request*은 *EB*에 포함된다. *Enter request*은 루트 체인에서 먼저 상태를 변경하고 *Request* 트랜잭션(*Request transaction*)의 형태로 블록에 포함된다. *Exit*, *Escape*, 그리고 *Undo request*의 경우 *Request* 트랜잭션의 형태로 블록에 포함된 이후 루트 체인에서 반영된다. 만약 해당 *Request* 트랜잭션이 실패(Reverted)되었다면 *Exit Challenge*를 통해 루트 체인에서 반영되는 것을 방지할 수 있다.

*Request*을 반영할 수 있는 컨트랙트는 *Requestable* 하며, *Requestable* 컨트랙트의 특정 함수를 호출함으로써 루트 체인과 자식 체인에서 *Request*을 반영할 수 있다. 각 컨트랙트 별로 *Requestable* 인터페이스를 개별적으로 구현하여 이더리움이 지향하는 General-purpose computing platform 을 플라즈마화(Plasmafy) 할 수 있다.

오퍼레이터는 *RootChain contract*가 *Request*을 생성할 수 있도록 각 체인의 *Requestable* 컨트랙트의 주소를 사전에 연결해야한다. 단, 각 체인의 *Requestable* 컨트랙트는 반드시 동일한 *codeHash* 가져야 하며, 이는 두 컨트랙트가 같은 Storage 레이아웃을 갖는다는 것을 의미한다.

3.4.1 Request and Request Transaction

*Request*는 다음 4개의 파라미터로 구성된다.

- *requestor*: *Request*를 생성한 어카운트
- *to*: 루트체인에 배포된 *Requestable contract*의 주소
- *trieKey*: *Request*의 식별자
- *trieValue*: *Request*의 값

*Request transaction*은 *Request block*에 포함되고 누구나 해당 블록을 마이닝할 수 있어야 하기에 *Null Address*를 *Transactor*로 하여 생성된다. *Null Address(NA)*는 비밀키가 없으며 주소가 $0x00$다. *Request transaction* 은 다음 5개의 파라미터로 구성된다.

- *sender*: *NA*
- *to*: 자식 체인에 배포된 *Requestable* 컨트랙트의 주소
- *value*: 0
- function signature: *Requestable*의 함수를 호출하기 위한 function signature
- parameters: emphRequestable의 함수를 호출하기 위한 파라미터

*RootChain contract*는 *Request transaction*의 해시와 *RB*의 *transactionsRoot*를 계산하여 *RB*에 *Request*이 반드시 포함될 수 있도록 강제할 수 있다. 또한 유저들은 *RootChain contract*에서 *Request* 데이터를 가져올 수 있기 때문에 *RB*에 대해서는 오퍼레이터가 Block Withholding Attack 할 수 없다. 이는 *EB* 또한 마찬가지이다.

3.4.2 Requestable Contract

Requestable 인터페이스는 아래의 함수를 가지고 있으며, *Requestable contract*는 아래 *Requestable* 인터페이스를 구현한다.

```
interface Requestable {
  function applyEnter(bool isRootChain, uint256 requestId, address
     requestor, bytes32 trieKey, bytes trieValue)
    external returns (bool success);

  function applyExit(bool isRootChain, uint256 requestId, address
     requestor, bytes32 trieKey, bytes trieValue)
    external returns (bool success);

  function applyEscape(bool isRootChain, uint256 requestId, address
     requestor, bytes32 trieKey, bytes trieValue)
    external returns (bool success);

  function applyUndo(bool isRootChain, uint256 requestId, address
     requestor, bytes32 trieKey, bytes trieValue)
    external returns (bool success);
}
```

*Requestable contract*에서 *Request*를 통해 변경이 가능한 변수는 요청가능(*Requestable*)한 변수이다. 단, *Requestable contract*의 모든 변수가 요청가능(*Requestable*)일 필요는 없다. *Request*와 관

련된 기능이 필요 없는 변수도 있고, 특정 변수에 대한 *Request* 권한을 사용자별로 구분해야 하는 경우도 고려해야 하기 때문이다. 예를 들어, 누구나 타인의 토큰 밸런스에 대해 *Request* 가 가능하다면 이는 바람직하지 않을 것이다. 따라서 우리는 미리 *Requestable* 컨트랙트에 이러한 문제를 해결할 수 있는 로직을 배치하고자 한다. 이는 $trieKey$를 이용하여 특정 변수에 대한 *Request* 권한을 확인 할 수 있기에 가능하다.

*Request*의 종류에 따라 *Request transaction*은 다른 Function signature를 가진다. 또한 *Requestable* 컨트랙트의 구현 예시는 B를 참고하라.

3.4.3 Apply Enter Request

*RootChain contract*는 *Enter request*를 각 체인에 배포된 *Requestable contract*에 다음과 같이 반영한다.

1. 사용자는 *RootChain contract*으로 $RootChain.startEnter()$를 호출하는 트랜잭션을 전송한다.
2. *RootChain contract*는 *Enter request*를 *Requstable contract*에 적용한다. 이는 루트 체인에서 일어난다. 만약 이 과정에서 트랜잭션이 실패(reverted)된다면 *Enter request*는 생성되지 않는다.
3. 2단계가 올바르게 진행되었다면, *RootChain contract*는 *Enter request*를 기록한다.
4. *Request Epoch*에서 오퍼레이터는 *Request transaction*을 포함하여 *Request Block*을 마이닝한다.
5. *Request transaction*이 자식 체인의 상태를 *Enter request*에 따라 변경한다.

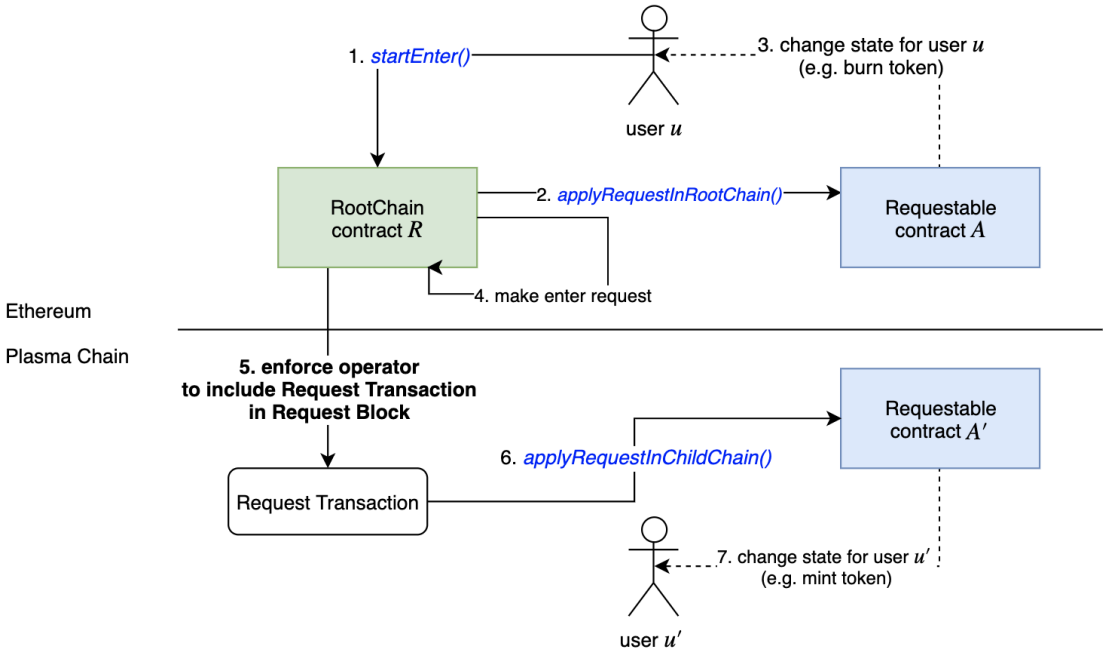

Figure 2: Enter Diagram

3.4.4 Apply Exit Request

*RootChain contract*는 *Exit request*를 각 체인에 배포된 *Requestable contract*에 다음과 같이 반영한다.

1. 사용자는 *RootChain contract*으로 *RootChain.startExit()*를 호출하는 트랜잭션을 전송한다.
2. *Enter request*와는 다르게 *Exit request*는 즉각적으로 기록되고 *Request transaction*의 형태로 *RB*에 포함된다.
3. *RB*에 대한 *Challenge* Period가 종료된 이후 *Exit request*에 대한 *Challenge* Period가 시작된다. 만약 2단계의 *Request transaction*이 실패되었다면 *RootChain.challengeExit()* 함수를 통해 *Exit Challenge* 할 수 있다.
4. 만약 3단계의 *Challenge*가 성공적으로 진행되지 않았다면, 유저가 $RootChain.finalizeRequest()$를 호출함으로써 *Exit request*는 *Finalize*된다. 해당 함수에서 *Exit request*는 루트 체인에 배포된 *Requestable contract*에 *Request*을 반영한다.

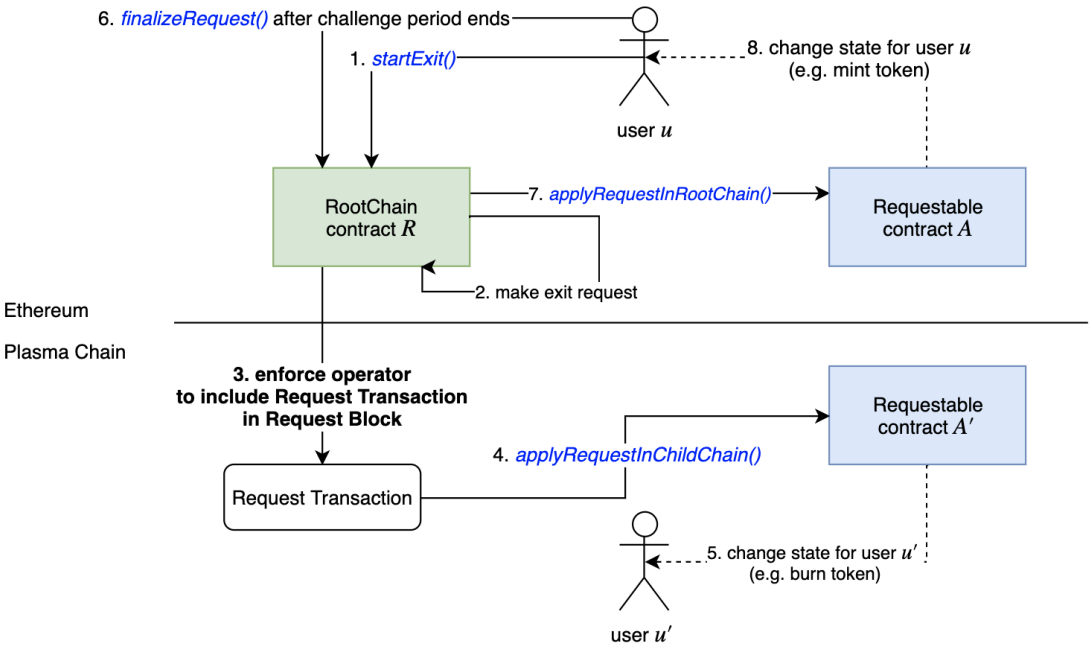

Figure 3: Exit Diagram

3.4.5 Apply Escape and Undo request

*Escape request*와 *Undo request*는 *Exit request*와 마찬가지로 처리되지만, *Request transaction*은 *Request Block*이 아닌 *Escape Block*에 포함된다.

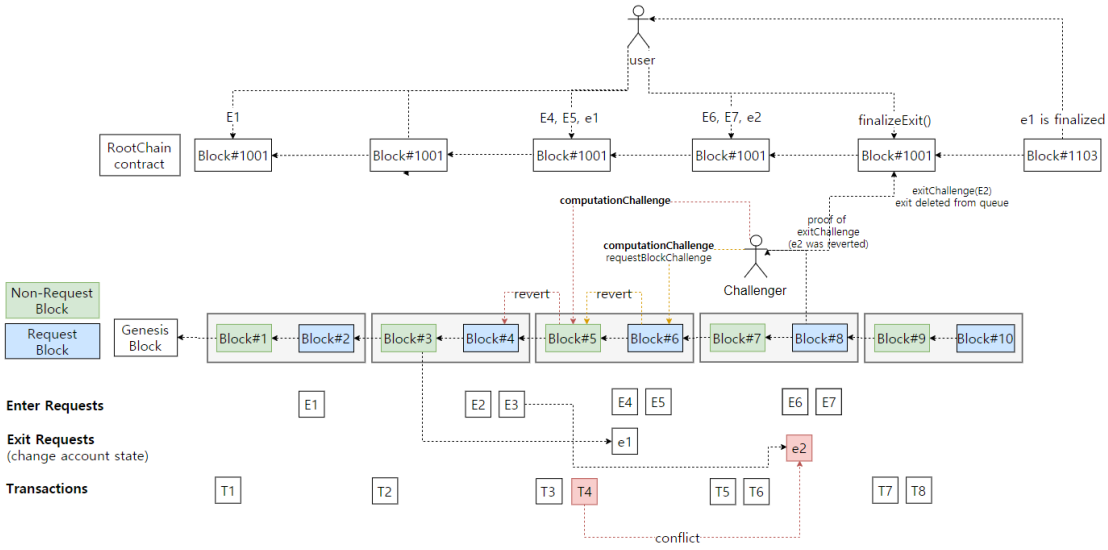

Figure 4: Request and challenge diagram

3.5 Continuous Rebase

플라즈마 EVM에서 *Rebase*[1]는 정상적인 작동과정의 일부이다. 따라서 지속적이고 주기적인 *Rebase*를 통해 사용자들의 *Escape request*을 반영할 수 있게 한다. 이를 통해 사용자들은 Data availability 문제(이하 DA문제)가 있을 경우 *Escape request*를 제출하여 *Finalize*된 마지막 블록을 기준으로 안전하게 탈출할 수 있다.

3.5.1 Rebase

*Rebase*는 기존에 루트체인에 제출한 블록을 다른 블록을 기준으로 다시 마이닝하는 것이다. 즉, 트랜잭션들을 다른 부모블록을 기준으로 재연산하는 것이다. 때문에 *Rebase*전과 후의 블록의 *transactionRoot*는 서로 일치하게 된다. 하지만 각 트랜잭션의 구체적인 실행결과는 달라질 수 있기 때문에, *stateRoot*와 *receiptsRoot*는 일치하지 않을 수 있다.

기존의 *NRE*, *ORE*들을 *Rebase*하는 *Epoch*은 *Rebased Non-Request Epoch(NRE')*, *Rebased Request Epoch(RE')*이다. 마찬가지로 *Rebased Non-Request Block(NRB')*, *Rebased Request Block(RB')*는 *Rebase*된 블록들이다.

3.5.2 Cycles and Stages

Plasma EVM의 작동은 *Cycle*을 하나의 주기로 하여 이루어진다. 하나의 *Cycle*은 총 4개의 *Stage*로 구성되어 있다. 각 *Stage*는 순서대로 *Pre-commit*, *DA-check*, *Commit*, *Challenge*에 해당한다. 모든 *Stage*를 순서대로 완료하면 해당 *Cycle*은 *Finalize* 되는 구조이다. 전체적인 동작 과정은 Figure 5를 참고하라.

[1]It is named after git's rebase.

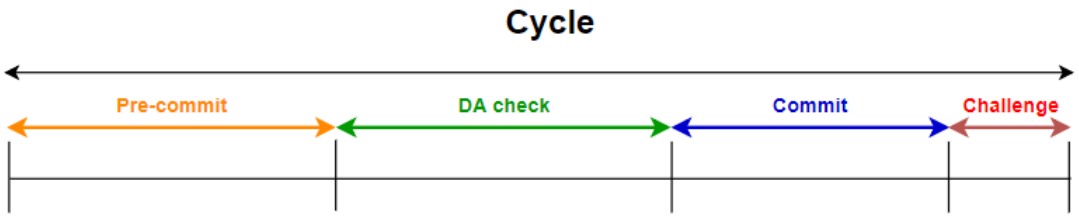

Figure 5: Cycles and stages

3.5.3 Pre-commit

*Pre-commit*은 일종의 예비 제출 단계로, 오퍼레이터는 3.2와 같이 블록을 마이닝한 후 해당 블록들의 *transactionRoot*를 루트체인에 제출한다. 또한 동시에 사용자들에게 해당 블록을 전파한다. 만약 *Pre-commit* 단계에서 오퍼레이터가 잘못된 블록 데이터를 전송하거나 혹은 전체 데이터를 전송하지 않을 경우 사용자는 *Escape request*를 제출할 수 있다. 단, *RB*의 경우 *RootChain contract*에 의해 *transactionRoot*가 결정되기 때문에, 이를 루트체인에 제출하지 않아도 된다.

*transactionRoot*만을 루트체인에 제출하는 이유는 *stateRoot*와 *receiptsRoot*가 *Rebase*과정에서 최종적으로 결정되기 때문이다. 또한 반드시 *transactionRoot*를 제출해야 하는 이유는 해당 블록에 포함될 트랜잭션이 무엇인지 알 수 있다면 해당 블록의 올바른 *stateRoot*또한 알 수 있기 때문이다. 따라서 오퍼레이터에게서 *RootChain contract*에 제출된 *transactionRoot*와 일치하는 블록 데이터를 전송받은 사용자는 이후 *Commit*단계에서 오퍼레이터가 데이터를 숨기고 잘못된 *stateRoot*를 제출하더라도 *Challenge* 할 수 있다.

Figure 6: Pre-commit

3.5.4 DA-check

*DA-check*는 사용자가 *Pre-commit* 단계에서 오퍼레이터가 전송한 블록 데이터의 가용성과 트랜잭션의 데이터 크기를 확인하고 문제가 없는지 점검하는 단계이다. 사용자가 DA문제 뿐만 아니라 각 트랜잭션의 크기도 점검해야 하는 이유는 루트체인 Block gas limit을 초과하는 트랜잭션은 루트 체인에서 검증이 불가능하기 때문이다. 만약 점검 과정에서 문제가 있는 비잔틴 상황이라면, 사용자는 스스로를 보호하기 위하여 해당 체인에서 비상탈출(Escape)해야 한다.

1) *Escape request* 제출 *Escape request*는 오퍼레이터에 의해 DA 문제가 생겼을 경우 사용자가 해당 플라즈마 체인에서 안전하게 탈출할 수 있는 장치이다. 구조와 기능은 *Exit request*와 동일하지만, 처리기준이 되는 블록이 항상 이전 *Cycle*의 *Commit*단계의 마지막 블록이라는 점에서 차이가 있다. 이는 사용자가 이전 *Cycle*의 마지막 상태를 기준으로 해당 체인에서 탈출할 수 있도

록 하기 위함이다. 만약 이전 *Cycle*에서 Escape하지 않았다면 사용자는 이전 *Cycle*에 대한 데이터를 모두 올바르게 전송받았음을 의미하고 이는 곧 해당 *Cycle*의 올바른 상태가 무엇인지 알고 있음을 의미한다.

2) 해당 *Cycle*에 Enter한 경우 *Undo request* 제출 *Escape request*는 이전 *Cycle*의 마지막 블록을 기준으로 처리되기 때문에 현재 *Cycle*의 *Enter request*를 되돌리는 역할을 할 수 없다. 때문에 사용자는 반드시 현재 *Cycle*의 Enter를 취소하는 *Undo request*를 *Escape request*와 함께 제출하여야 한다. *Undo request*는 자식 체인에 *Enter request*가 반영되는 것을 방지하며, 이후 *Undo request*가 *Finalize* 되면 사용자는 루트체인에서의 *Enter request*로 인한 상태 전이를 되돌릴 수 있다.

3) 해당 *Cycle*에 Exit한 경우 Exit 취소 만약 *Escape request*로 모든 상태를 Exit한 후에 또다시 *Exit request*가 실행된다면 해당 *Request*는 revert 되어 *Exit challenge*의 대상이 될 수 있다. 따라서 *Escape request*를 제출한 사용자는 *RootChain contract*에 해당 *Cycle*의 모든 *Exit request*를 취소하는 *Request*을 제출해야 한다. 취소된 *Exit request*는 그 실행결과와 관계 없이 *Exit challenge*의 근거가 될 수 없다. 또한 취소된 *Exit request*는 *Finalize* 되더라도 루트체인에서 실질적인 상태변환을 발생시킬 수 없다. 따라서 취소된 *Exit request*로 사용자는 어떠한 부적절한 이득도 얻지도 않으며 손실도 입지 않을 수 있다.

DA-check 단계가 명시적으로 필요한 이유는 *Pre-commit* 단계에서 오퍼레이터가 데이터를 올바르게 전송하지 않는 경우 사용자가 안전하게 *Escape request*를 제출할 수 있도록 여유시간을 주는데 그 목적이 있다. 예를 들어, 오퍼레이터가 악의적으로 *Pre-commit* 단계의 마지막에 Block Withholding Attack을 하더라도 사용자가 이를 인지하고 적절한 행동을 취할할 수 있도록 여유 시간을 제공하는 것이다. 따라서 *Pre-commit* 단계의 초기부터 DA문제가 발생했을 경우 사용자는 *DA-check* 단계를 기다리지 않고 즉시 *Escape request*를 제출할 수 있다.

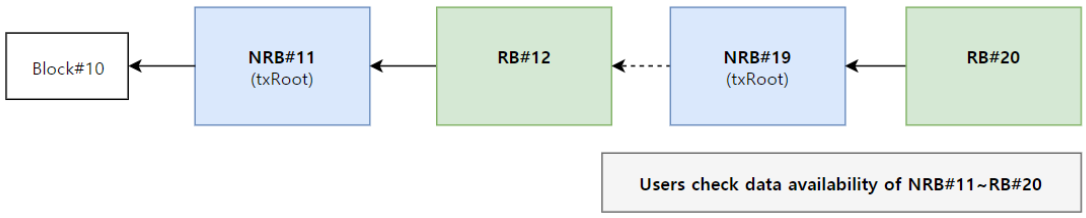

Figure 7: DA-check

3.5.5 Commit

*Commit*은 오퍼레이터가 *Pre-commit*과 *DA-check*에서 제출된 *Escape request*와 *Undo request*를 반영하는 *Escape block(EB)*를 제출하고, *EB*를 기준으로 *Pre-commit*단계에서 제출된 모든 블록들을 *Rebase*하고 *Pre-commit*된 블록의 *stateRoot*와 *receiptsRoot*를 제출하는 과정이다.

이 때 *EB*의 부모블록은 이전 *Cycle*의 Commit된 마지막 블록이다. 단, *Escape request*가 단 하나도 제출되지 않았을 경우, *EB* 제출 과 *Rebase*는 생략하고, *Pre-commit* 단계에서 마이닝한 블록의 *stateRoot*와 *receiptsRoot*를 그대로 제출한다.

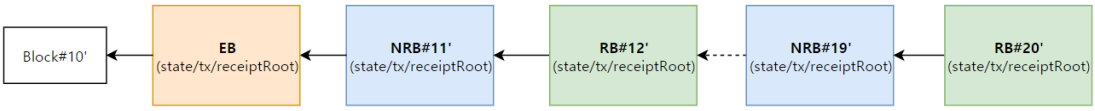

Figure 8: Commit

3.5.6 Challenge

*Challenge*단계에서 사용자들은 *Commit* 단계에서 제출된 블록이 유효하지 않을 경우, 이에 대한 *Challenge*를 제출할 수 있다. 단, 이 때 제출가능한 *Challenge*는 *Exit challenge*를 제외한 *Null Address Challenge*, *Computation Challenge*이다.

제출된 *Challenge*가 다수일 경우 챌린저가 승리하는 순간 다른 *Challenge*는 취소된다. 또한 챌린저가 승리한 경우 해당 *Cycle*의 *Commit*된 블록을 포함한 이후의 모든 블록들이 취소되며, 해당 *Cycle*은 *DA Check*단계로 되돌아간다.

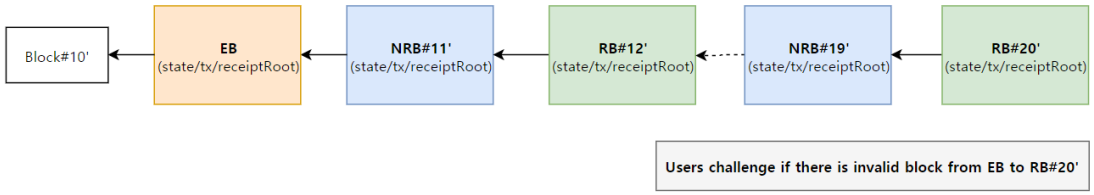

Figure 9: Challenge

3.6 Finalize

Challenge 단계 동안 성공적인 챌린지가 없다면, 해당 *Cycle*은 *Finalize* 되며, *Commit* 단계에서 제출된 모든 블록은 동시에 *Finalize* 된다. 단, 제출된 모든 챌린지가 종료되기 전까지 *Challenge* 단계는 종료되지 않으며, 해당 *Cycle*을 포함한 이후의 모든 *Cycle*은 *Finalize*되지 않는다.

3.7 Stage Length

*Stage length*는 *Pre-commit*과 *Commit*에서 처리되어야 하는 *Epoch*의 수이다. *Pre-commit length*는 $N_{NRE} * 2$ 로 이는 *RootChain contract* 배포시 오퍼레이터에 의해 결정된다. *Commit length*는 $N_{NRE} * 2 + 1$ 로 역시 *RootChain contract* 배포시 결정되며, 추가된 하나의 *Epoch*은 *Escape Epoch*을 의미한다.

3.8 Stage Period

*Stage period*는 각 *Stage*에 할당된 시간이다. *Pre-commit period*과 *Commit period*의 경우 이후 다룰 *Halting condition*의 충족 기준이 되며, *DA-check period* 와 *Challenge period*는 단순히 해당 *Stage* 기간이다.

모든 *Stage period*는 *RootChain contract* 배포시 오퍼레이터에 의해 결정된다. *Pre-commit*과 *Commit*은 *Stage period*와 관계없이 각 *Stage length*에 해당하는 *Epoch*을 제출한 경우 완료될 수 있다. *DA-check*와 *Challenge*는 오직 *Stage period* 이후에만 완료된다. 단, *Challenge period* 의 경우 *Computation Challenge*의 진행여부에 따라 연장될 수 있다.

3.9 Halting condition

어떠한 이유로든 오퍼레이터가 *Cycle*의 각 *Stage*의 정해진 절차를 제대로 수행하지 않을 경우 *Halting condition*이 충족될 수 있다. *Halting condition*이 충족될 수 있는 *Stage*는 *Pre-commit*, *Commit*이며, 구체적인 충족 조건과 그 결과는 다음과 같다.

Pre-commit 오퍼레이터가 *Pre-commit period* 이내에 *Pre-commit length*개의 *Epoch*을 모두 제출하지 못할 경우 Halting condition이 충족된다. 이 경우 해당 체인은 *Shutdown*된다.

Commit 오퍼레이터가 *Commit period* 동안 *Commit length*개의 *Epoch*을 모두 제출하지 못할 경우 Halting condition이 충족된다. 이 경우 해당 *Cycle*을 포함한 이후의 모든 *Cycle*의 진행이 일시 중단된다. 만약 멈춘 시점에 *EB*가 제출되지 않은 상태라면 누구나 *EB*를 제출할 수 있게 되고, 해당 블록이 제출되면 일시 중단된 *Cycle*이 재개된다. 만약 *Rebase*의 진행중에 멈췄다면 해당 체인은 *Shutdown*된다.

Shutdown *Shutdown*은 일종의 플라즈마 체인 폐쇄 절차이다. *Shutdown*이 되면 더 이상 해당 체인의 추가적인 *Cycle*은 진행될 수 없다. 오직 Last finalized *Cycle*을 기준으로 *Escape request*제출 - *EB* 제출 - *Challenge*를 반복하여 해당 체인의 모든 사용자들이 안전하게 Exit할 수 있도록 한다. *Shutdown*상태에서는 누구나 *EB*제출이 가능하기 때문에 사용자들 스스로 Exit이 가능하다.

지금까지 살펴본 하나의 *Cycle*의 동작과정은 Figure 10로 나타낼 수 있다.

3.10 Overlap of Cycle

지금까지 논의한 것은 하나의 *Cycle*의 동작 과정이었다. 여러개의 *Cycle*이 맞물렸을 때는 이전 *Cycle*의 *Challenge*단계의 완료 여부와 관계 없이 다음 *Cycle*의 *Pre-commit* 단계가 시작될 수 있다.

Figure 11에서 *Cycle* 2의 *Pre-commit* 단계가 완료되고 *DA-check* 단계가 시작됨과 동시에 다음 *Cycle*인 *Cycle* 3의 *Pre-commit* 과정이 시작됨을 알 수 있다. 이 때, *Cycle* 3의 *Pre-commit* 단계의 첫 블록은 *Cycle* 2의 가장 마지막 *Pre-commit* 블록을 기준으로 상태가 연산된다. 이처럼 *Cycle*이 서로 맞물려 진행되기 위해서는 다음과 같은 조건들이 충족되어야 한다.

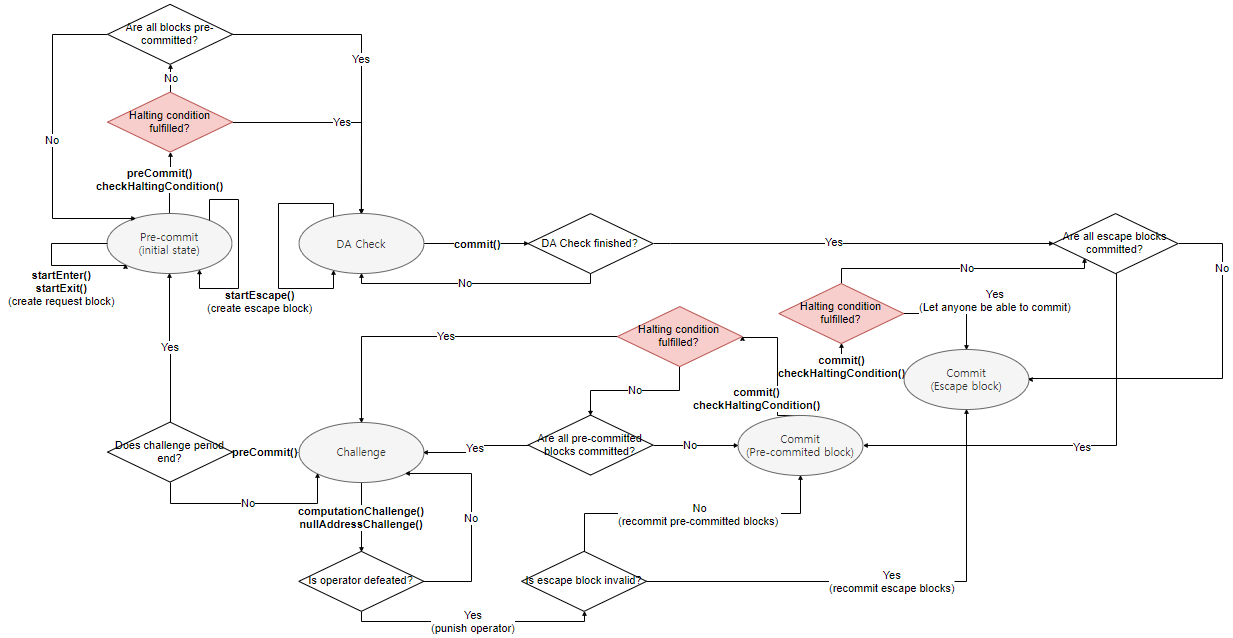

Figure 10: Continuous rebase (Sequential cycle)

1. 현재 *Cycle*의 *Pre-commit*이 완료되는 즉시 다음 *Cycle*의 *Pre-commit*이 시작된다.
2. 현재 *Cycle*의 *Pre-commit*이 완료되는 즉시 현재 *Cycle*의 *DA-check*가 시작된다.
3. 현재 *Cycle*의 *Commit*은 이전 *Cycle*의 *Commit* 완료 이후에 시작될 수 있다.

이를 통해 오퍼레이터가 정상적으로 자식체인을 운영할 경우 사용자들은 계속 해당 체인을 이용할 수 있게 된다. 중요한 것은 이전 *Cycle*의 *Challenge stage*의 완료와 관계 없이 다음 *Cycle*의 진행이 가능하다는 것이다. 단, 이 경우 이후의 *Cycle*에 대한 *Challenge*가 종료되더라도 이전 *Cycle*이 *Finalize*되기 전까지는 *Finalize*될 수 없다.

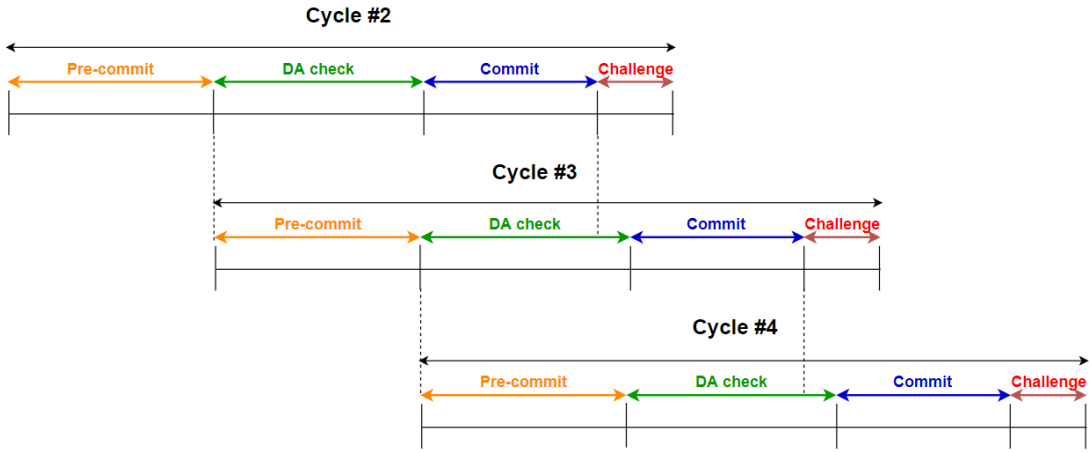

Figure 11: Overlap of cycle

3.11 Finality

Block 플라즈마 EVM은 챌린지 기간을 어떠한 유효한 챌린지 없이 넘긴 블록에 대해 플라즈마 XT처럼 블록의 체크포인트를 지정할 수 있다.

Request Block의 Finality와 별개로 Enter를 제외한 모든 *Request*에 대한 Finality도 존재한다. *Request*가 포함된 블록이 *Finalize* 되면 해당 *Request*에 대한 챌린지 기간인 *Request challenge period*가 시작되는데, 이 기간 동안 어떠한 유효한 챌린지도 제출되지 않는다면 해당 *Request*는 *Finalize* 된다.

4 Challenge

다음과 같은 *Challenge*를 통해 블록 혹은 *Request*의 유효성을 검증한다.

4.1 Null Address Challenge

NRB에 *Transactor*가 NA인 트랜잭션이 있는지 확인한다. 그러한 트랜잭션은 *Request transaction*을 의미하기 때문에 RB, EB가 아닌 NRB에 포함될 경우는 오퍼레이터가 비잔틴한 경우이다.

4.2 Exit challenge

RB에 포함된 *Exit request*와 *Escape request*가 유효하지 않다면 이에 대응 하는 *Request transaction*은 Revert 되며, 해당 *Request*는 반드시 *RootChain contract*에서 삭제되어야 한다. *Exit challenge*는 Revert 된 *Request transaction*을 증거로 삼아 올바르지 않은 *Exit request*를 검증한다. 그러나 *Exit request*에 대해 챌린지하기 전에, 해당 *Request*를 포함하고 있는 블록이 *Finalize* 되어야 한다. 그 이유는 유효한 블록에서 Revert되지 않는 *Request transaction*이 유효하지 않은 블록에서는 Revert될 수 있기 때문이다. 따라서 *Exit challenge*는 챌린지 대상을 포함하는 블록이 *Finalize*된 후에 시작 될 수 있다.

4.3 Computation Challenge

오퍼레이터가 제출한 NRB, RB, EB에 대해 *Computation Challenge*는 오퍼레이터가 트랜잭션을 올바르게 실행했는지를 검증한다. 오퍼레이터가 잘못된 *stateRoot*를 제출하면, *blockData*, *preStateRoot* 및 *postStateRoot*를 바탕으로 TrueBit-like Verification game을 통해 챌린지 된다.

$$preState = commitedStateRoots[i-1]$$

$$postState = commitedStateRoots[i] = STF_{block}(preState, Block_i)$$

*RootChain contract*에서 STF_{block}를 실행한 output과 이미 제출된 output을 비교하여 블록의 상태 전이가 올바르게 이루어졌는지 검증할 수 있다.

4.4 Verification Game

TrueBit은 outsource된 연산을 검증하기 위한 방법으로 Verification game을 제안했다. 그러나 TrueBit이 제안한 게임의 마지막 단계는 이더리움에서 연산을 한 번 수행하고 실제 output과 예상 output을 비교하는 방법을 사용한다. 우리는 Ohalo Limited와 Parsec Labs에서 구현해 왔던 EVM 내부에서 EVM을 실행하는 스마트 컨트랙트인 solEVM 12을 사용하여 연산 결과를 검증하고자 한다. 단, 수수료 위임 체인을 사용하려면 수수료 위임 트랜잭션 실행 모델이 solEVM에 반영되어야 한다.

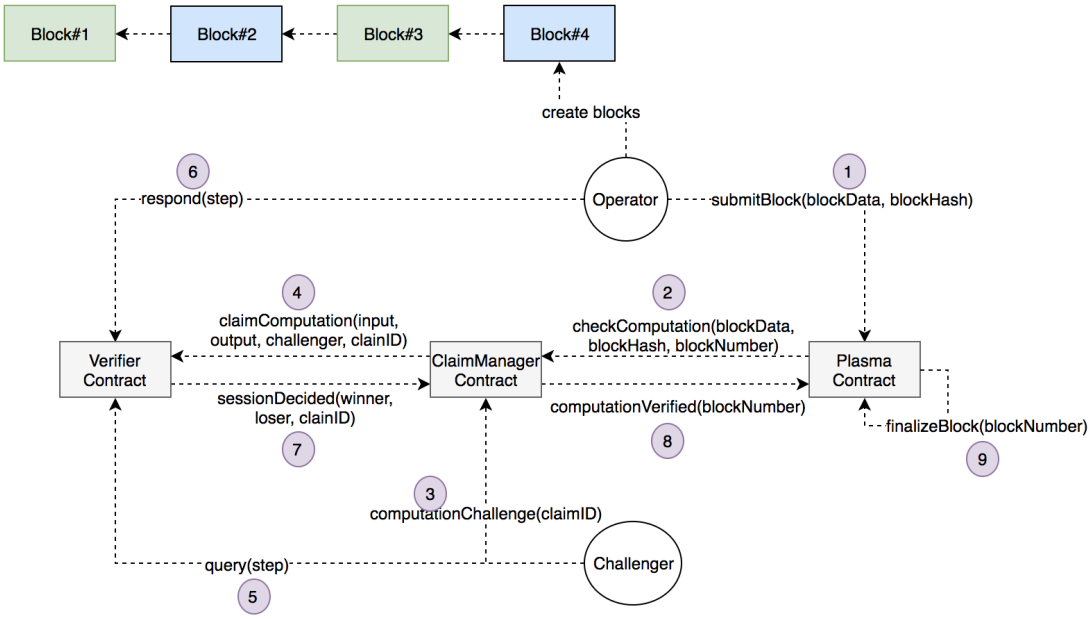

Figure 12: Verification game

5 Attack scenarios

5.1 Block Withholding Attack

오퍼레이터가 *Pre-commit*단계에서 Block withholding attack을 할 경우 사용자들은 *Pre-commit*과 *DA-check* 단계에서 *Escape request*를 통해 해당 체인에서 안전하게 Exit 할 수 있다.

5.2 Invalid block

오퍼레이터는 *Commit*단계에서 올바르지 않은 블록을 루트체인에 제출할 수 있다. 하지만 이 경우 해당 체인에 남아있는 사용자들은 모두 해당 블록의 트랜잭션이 무엇인지 확인한 상태이므로 올바른 블록이 어떻게 구성되어야 하는지 알 수 있다. 따라서 사용자는 *Computation challenge*를 통해 올바르지 않은 블록을 취소할 수 있다.

6 Further Research

6.1 Transaction Finality

*Rebase*과정을 플라즈마 체인의 정상적인 작동과정으로 간주하면서 DA 문제는 해결되었다. 하지만 사용자들은 이러한 점을 악용해 DA문제의 여부와 관계 없이 *Escape request*를 제출하여 의도적으로 본인의 트랜잭션을 취소할 수 있게 되었다. 따라서 사용자는 해당 트랜잭션을 취소하는 *Escape request*가 제출되었는지 반드시 확인해야 한다.

예를 들어, 탈중앙 거래소(Decentralized exchange; DEX)가 Plasma EVM위에서 운영되고 있다고 하자. 또한, 하나의 *Cycle*이 *Finalize* 되기 까지 약 24시간이 소요된다고 하자. 사용자 A는 약 1시간 전에 사용자 B에게서 토큰 T를 100ETH에 구매했다. 하지만 현재 토큰 T의 가격은 1ETH로 폭락하였다. 이 경우 사용자 A는 해당 트랜잭션이 *Finalize* 되기 전에 *Escape request*를 제출하여 본인의 모든 잔액을 미리 출금하여 해당 트랜잭션을 취소시킬 것이다. 이 경우 해당 토큰을 판매한 사용자 B는 중대한 손실을 입을 수 있다.

6.2 Light Client Support

사용자들은 DA문제의 발생 여부를 인지하기 위해 항상 모든 블록 데이터를 다운로드 받아야 한다. 다시 말해 모든 사용자들이 해당 플라즈마 체인에서 풀노드(Full node)를 구동시키고 있어야 함을 의미한다. 이는 UX(User Experience)를 심각하게 저해하는 요소가 될 수 있다.

물론 이러한 문제는 비단 Plasma EVM에만 해당되는 것은 아니다. 대부분의 다른 플라즈마 모델들도 사용자들이 모든 데이터를 확인하기 위해 풀노드를 구동시켜야 하는 제약을 갖고 있다. 때문에 이러한 문제점을 해결하기 위해 Erasure coding을 이용하여 일종의 Light client만으로도 Data availability를 확인할 수 있게하는 연구가 제시되고 있다.[2] Plasma EVM에서도 이와 같이 사용자들이 풀노드가 아닌 Light client만으로 DA문제의 발생 여부를 인지할 수 있도록 하기 위해 추가적인 연구가 필요하다.

6.3 Addressing Requestable Contracts in Both Chain

*Requestable contract*는 양 체인에 동일한 codeHash를 가져야 한다. 기존의 Contract Creation Transaction은 Transactor의 주소와 Nonce를 이용하여 컨트랙트의 주소를 결정해야 했기에, 오퍼레이터가 두 컨트랙트의 codeHash와 주소를 확인하고 *RootChain contract*에 mapping하는 방법밖에 없었다. 하지만 Constantinople 하드 포크 11에서 새로 추가될 CREATE2를 이용하여 단순히 자식 체인에서 생성한 트랜잭션을 루트 체인에서 동일하게 실행한다면 누구나 *Requestable contract*를 양 체인에 동일한 주소로 배포할 수 있다.

CREATE2를 이용하여 컨트랙트를 생성하는 Factory 컨트랙트를 루트 체인에 배포된 *RootChain contract*와 동일한 주소를 가지게 하여 자식 체인의 Genesis 블록에 넣어둔다면 위와같은 기능을 수행할 수 있다.

[2] Vitalik Buterin. A note on data availability and erasure coding

7 Conclusion

Plasma EVM은 *Request*, *Request block*, *Requestable contract*를 통해 일반화된 상태를 플라즈마 체인에서 다룰 수 있게 하였다. 챌린지는 항상 올바른 상태만이 루트체인에서 fianlzie될 수 있도록 보장한다. 무엇보다 일반화된 플라즈마의 치명적인 취약점이었던 DA문제를 Continuous Rebase를 통해 해결함으로써 어떠한 공격상황에서도 정직한 사용자들의 피해를 막을 수 있게 되었다. 하지만 *Rebase*로 인한 Instant finality 보장 불가능과 같은 단점들은 여전히 해결해야 할 주요 과제이다.

A Glossary

A.1 General

- Root chain: Ethereum blockchain
- Child chain: Plasma blockchain. It can also be called plasma chain.
- Operator: Agent that operate child chain
- NULL ADDRESS: $0x00$ with nonce and signature $v = r = s = 0$, denoted NA
- Transactor: Account which generate transaction, *tx.origin*
- RootChain manager contract: A plasma contract on root chain accepting enter / exit (ETH / ERC20)
- Request: A request which enforces to apply state transition by root chain
- Requestable contract: Contracts able to accept exit / enter request in both of root and child chain. 2 identical contracts should be deployed in root and chind chain, and R maps two addresses.
- Enter request: A request to enter something from root chain to child chain. eg) deposit asset, move account storage variable.
- Exit request: A request to exit asset or account storage from child chain to root chain. Any exit request on root chain immediately updates account storage in child chain. If the update in child chain is rejected(TX reverted), the exit can be challenged with the computation output of the update as proof.
- Escape request: A request to escape from child chain. It has identical structure to exit request, but there is restriction on the submission time.
- Undo request: A request to prevents future enter request in child chain from being applied.
- Request block: A block applying state transition that is enforced by the root chain, denoted RB
- Non-Request block: A block where transactions are only related between accounts in child chain, denoted NRB
- Request block: A request block applying enter / exit request by the operator, denoted RB
- Escape request block: A request block only including escape requests and undo requests, denoted EB
- Epoch: A period in which the same block must be submitted. Epoch for each type of block is denoted NRE, ORE, and ERE respectively.

A.2 Challenge

- Null Address Challenge: challenge if NRB contains a transacton from NA.
- Computation Challenge: challenge if block have the state computed in a wrong way.
- Exit Challenge: challenge if invalid *Exit request* cannot be accepted in child chain (but the request should be included in RB).
- Finalized: Every block and *Exit request* can be deterministically finalized only if any no successful challenge exists.

A.3 Continuous Rebase

- Rebase: Re-mining blocks based on another block. As a result, stateRoot and receiptsRoot of blocks can be changed but transactionRoot should be identical to original one.
- Cycle: The entire operational period of the Plasma EVM, one cycle consists of a total of four stages. Each stage is Pre-commit, DA-check, Commit and Challenge.
- Pre-commit: A stage that operator mines blocks in child chain and submits *transactionRoot* of them. At the same time, the operator must broadcast block data to users so that they can check data availability.
- DA-check: A stage that users check data availability of blocks submitted in Pre-commit. In case of unavailability, they can submit *Escape request* to escape from that chain. However, *Escape request* can be submitted at both Pre-commit, DA-check stages.
- Commit: A stage that operator must submit *Escape request block* applying *Escape request* and do *Rebase* blocks submitted in Pre-commit.
- Challenge: A stage that users can challenge if blocks submitted in Commit are invalid.
- Stage length: The number of *Epoch* to be processed in *Pre-commit* and *Commit*.
- Stage period: The time allocated for each *Stage*.
- Halting condition: A specific condition to halt child chain.
- Finalize: After Challenge stage, whole blocks in a cycle can be finalized.
- Shutdown: A closing procedure of plasma chain. After shutdown, no more cycle cannot be initiated. It only allows users to escape from that chain.

B Requestable Contract Examples

B.1 Requestable token contract

```
// @NOTE: This implements only enter and exit requests.
contract RequestableSimpleToken is Ownable, RequestableI {
  using SafeMath for *;

  // `owner` is stored at bytes32(0).
  // address owner; from Ownable

  // `totalSupply` is stored at bytes32(1).
  uint public totalSupply;

  // `balances[addr]` is stored at keccak256(bytes32(addr), bytes32(2))
     .
  mapping(address => uint) public balances;

  // requests
  mapping(uint => bool) appliedRequests;

  /* Events */
  event Transfer(address _from, address _to, uint _value);
  event Mint(address _to, uint _value);
  event Request(bool _isExit, address _requestor, bytes32 _trieKey,
     bytes _trieValue);

  function transfer(address _to, uint _value) public {
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);

    emit Transfer(msg.sender, _to, _value);
  }

  function mint(address _to, uint _value) public onlyOwner {
    totalSupply = totalSupply.add(_value);
    balances[_to] = balances[_to].add(_value);

    emit Mint(_to, _value);
    emit Transfer(0x00, _to, _value);
```

```
}

// User can get the trie key of one's balance and make an enter
   request directly.
function getBalanceTrieKey(address who) public pure returns (bytes32)
    {
  return keccak256(bytes32(who), bytes32(2));
}

function applyRequestInRootChain(
  bool isExit,
  uint256 requestId,
  address requestor,
  bytes32 trieKey,
  bytes trieValue
) external returns (bool success) {
  // TODO: adpot RootChain
  // require(msg.sender == address(rootchain));
  // require(!getRequestApplied(requestId)); // check double applying

  require(!appliedRequests[requestId]);

  if (isExit) {
    // exit must be finalized.
    // TODO: adpot RootChain
    // require(rootchain.getExitFinalized(requestId));

    if (bytes32(0) == trieKey) {
      // only owner (in child chain) can exit `owner` variable.
      // but it is checked in applyRequestInChildChain and
         exitChallenge.

      // set requestor as owner in root chain.
      owner = requestor;
    } else if (bytes32(1) == trieKey) {
      // no one can exit `totalSupply` variable.
      // but do nothing to return true.
    } else if (keccak256(bytes32(requestor), bytes32(2)) == trieKey)
       {
```

```
      // this checks trie key equals to `balances[requestor]`.
      // only token holder can exit one's token.
      // exiting means moving tokens from child chain to root chain.
      balances[requestor] += decodeTrieValue(trieValue);
    } else {
      // cannot exit other variables.
      // but do nothing to return true.
    }
  } else {
    // apply enter
    if (bytes32(0) == trieKey) {
      // only owner (in root chain) can enter `owner` variable.
      require(owner == requestor);
      // do nothing in root chain
    } else if (bytes32(1) == trieKey) {
      // no one can enter `totalSupply` variable.
      revert();
    } else if (keccak256(bytes32(requestor), bytes32(2)) == trieKey)
     {
      // this checks trie key equals to `balances[requestor]`.
      // only token holder can enter one's token.
      // entering means moving tokens from root chain to child chain.
      require(balances[requestor] >= decodeTrieValue(trieValue));
      balances[requestor] -= decodeTrieValue(trieValue);
    } else {
      // cannot apply request on other variables.
      revert();
    }
  }

  appliedRequests[requestId] = true;

  emit Request(isExit, requestor, trieKey, trieValue);

  // TODO: adpot RootChain
  // setRequestApplied(requestId);
  return true;
}
```

```
function decodeTrieValue(bytes memory trieValue) public pure returns
    (uint v) {
  require(trieValue.length == 0x20);

  assembly {
     v := mload(add(trieValue, 0x20))
  }
}

// this is only called by NULL_ADDRESS in child chain
// when i) exitRequest is initialized by startExit() or
//      ii) enterRequest is initialized
function applyRequestInChildChain(
  bool isExit,
  uint256 requestId,
  address requestor,
  bytes32 trieKey,
  bytes trieValue
) external returns (bool success) {
  // TODO: adpot child chain
  // require(msg.sender == NULL_ADDRESS);
  require(!appliedRequests[requestId]);

  if (isExit) {
    if (bytes32(0) == trieKey) {
      // only owner (in child chain) can exit `owner` variable.
      require(requestor == owner);

      // do nothing when exit `owner` in child chain
    } else if (bytes32(1) == trieKey) {
      // no one can exit `totalSupply` variable.
      revert();
    } else if (keccak256(bytes32(requestor), bytes32(2)) == trieKey)
      {
      // this checks trie key equals to `balances[tokenHolder]`.
      // only token holder can exit one's token.
      // exiting means moving tokens from child chain to root chain.
```

```
        // revert provides a proof for `exitChallenge`.
        require(balances[requestor] >= decodeTrieValue(trieValue));

        balances[requestor] -= decodeTrieValue(trieValue);
      } else { // cannot exit other variables.
        revert();
      }
    } else { // apply enter
      if (bytes32(0) == trieKey) {
        // only owner (in root chain) can make enterRequest of `owner`
           variable.
        // but it is checked in applyRequestInRootChain.

        owner = requestor;
      } else if (bytes32(1) == trieKey) {
        // no one can enter `totalSupply` variable.
      } else if (keccak256(bytes32(requestor), bytes32(2)) == trieKey)
         {
        // this checks trie key equals to `balances[tokenHolder]`.
        // only token holder can enter one's token.
        // entering means moving tokens from root chain to child chain.
        balances[requestor] += decodeTrieValue(trieValue);
      } else {
        // cannot apply request on other variables.
        revert();
      }
    }

    appliedRequests[requestId] = true;

    emit Request(isExit, requestor, trieKey, trieValue);
    return true;
  }
}
```

References

[1] Joseph Poon and Vitalik Buterin. Plasma: Scalable Autonomous Smart Contracts, https://plasma.io/

[2] Vitalik Buterin. Minimal Viable Plasma, https://ethresear.ch/t/minimal-viable-plasma/426

[3] Vitalik Buterin. Plasma Cash: Plasma with much less per-user data checking, https://ethresear.ch/t/plasma-cash-plasma-with-much-less-per-user-data-checking/1298

[4] Kelvin Fichter. Plasma XT: Plasma Cash with much less per-user data checking, https://ethresear.ch/t/plasma-xt-plasma-cash-with-much-less-per-user-data-checking/1926

[5] PARSEC Labs. PLASMA - FROM MVP TO GENERAL COMPUTATION, https://parseclabs.org/files/plasma-computation.pdf

[6] Johann Barbie. Plasma Leap - a State-Enabled Computing Model for Plasma, https://ethresear.ch/t/plasma-leap-a-state-enabled-computing-model-for-plasma/3539

[7] Jason Teutsch, Christian Reitwießner. A scalable verification solution for blockchains, https://people.cs.uchicago.edu/ teutsch/papers/truebit.pdf

[8] Vitalik Buterin. A note on data availability and erasure coding, https://github.com/ethereum/research/wiki/A-note-on-data-availability-and-erasure-coding

[9] Kelvin Fichter. Why is EVM-on-Plasma hard?, https://medium.com/@kelvinfichter/why-is-evm-on-plasma-hard-bf2d99c48df7

[10] Ben Jones, Kelvin Fichter. More Viable Plasma, https://ethresear.ch/t/more-viable-plasma/2160

[11] Ethereum Blog. Ethereum Constantinople/St. Petersburg Upgrade Announcement, https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announceme

[12] Ohalo Limited. Solidity EVM and Runtime, https://github.com/Ohalo-Ltd/solevm