BitVM bridge のデモ v0.1.0-alpha

BitVM2のコードはまだ安定版がないので~~書いている時点の最新版~~デモを動かしていたと思われる `v0.1.0-alpha` をベースに修正して試す。

https://github.com/hirokuma/BitVM/tree/demo/v0.1.0-alpha

- https://github.com/hirokuma/BitVM/tree/demo/v0.1.0-alpha-typo : EVM address を間違えていたときの記録

https://bitvm.org/demo/

このデモは、正確には BitVM2をベースにして設計した BitVM Bridge のものである。

- BitVM2自体は汎用フレームワークである

> BitVM Bridge は BitVM2 を使って、「BTCの引き出し主張を任意のブール回路で検証できるように設計された、非カストディアル・チャレンジ可能型ブリッジプロトコル」である。

このデモの手順は、BitVM2の正式な手順ではない。 `v0.1.0-alpha` の後で変更になったのか、このバージョンを意図的にそうしたのか分からないが、challenge 無しで assert フェーズの処理をした後 take_2 で Operator が取り戻すようになっている。
これより後のcommitでは正式な手順になっているようだ。
BitVM bridgeのデモ v0.1.1-b822f19

ビルド

`cargo build` するだけである。そこまで時間はかからなかった。
vscode でプロジェクトを開くと rust-analyzer が走り出してかなり長い時間ビルドらしきことをやっていた。

ヘルプ

`bridge help` で見えるコマンドと実際に使えるコマンドは実装箇所が別になっていて、ヘルプでは見えるけれども実際には使えないコマンドがあった(勘弁してほしい。。。)。

環境

Environment Setup

- 環境変数はカレントディレクトリの `.env` に書いておくと `bridge` が自動で読み込んでくれる。

Esplora API を使っているので、regtest の場合は regtest/ を参照して docker コンテナを使う方が楽。

`regtest/` の `install.sh` を実行後、 `start.sh` で常駐してブロックを定期的に生成し続ける。

docker で動いているので cli を実行するのにスクリプトを用意しておくと楽だろう。

cli.sh

```
#!/bin/bash

pid=`docker ps | grep blockstream/esplora | awk '{print $1}'`
generate_command="/srv/explorer/bitcoin/bin/bitcoin-cli -conf=/data/.bitcoin.conf -
datadir=/data/bitcoin $@"

docker exec $pid /bin/bash -c "$generate_command"
```

データストアは AWS S3、FTP/FTPS、SFTPが選択できる。ここでは SFTP を使う設定にしている。

ログにファイルパスとして `anvil_831337` を含んでいる場合はh、 Destination Network は Local になっているときである。

環境変数 `ENVIRONMENT` で見分けているようだ。

ENVIRONMENT の組み合わせ(bridge)

src-dst

Name	Source	Destination
mainnet	Bitcoin Mainnet	Ethereum Mainnet
testnet	Bitcoin Testnet	Ethereum Sepolia
regtest	Bitcoin Regtest	Local

もう1つ "bridge-query" にも似たような箇所があるが、こちらは "regtest" 相当の名前が "local" になっている。まだよくわからない。

ENVIRONMENT の組み合わせ?

src-dst

Name	Source	Destination
mainnet	Bitcoin Mainnet	Ethereum Mainnet
testnet	Bitcoin Testnet	Ethereum Sepolia
local	Bitcoin Regtest	Local

Graph ID

回路構造に対して計算されるハッシュ値。デモの手順では以下の Graph ID が登場した。

peg-in ID

peg-out ID

デモの登場人物

Alice: peg-inするユーザ

Bob: L2上で Aliceからの送金を受け取ったあと peg-out して Bitcoin に戻すユーザ

らしいが、Alice が peg-in して peg-out すると説明した方が手順としては正しいんじゃなかろうか

DEPOSITOR / OPERATOR / VERIFIER_0

3役をこなす

VERIFIER_1

責任

行為	誰が責任を負う？	結果
BTCを先に送る	オペレーター	詐欺だったら損失を被る
引き出しが正当	オペレーター	BitVM2で証明 → 担保回収できる
引き出しが不正	オペレーター	担保没収、BTC払い損
チャレンジをサボる	Vigilante	システムが詐欺を許すことになる
チャレンジ成功	Vigilante	報酬を得られる（担保の一部）

"vigilante" は「自警団員」の意味なので VERIFIER か？

どの BitVMバージョンならデモが動くのか？

タグとしては古い順に `v0.1.0-dev` 、 `v0.1.0-alpha` 、 `v0.1.1-dev` があり、デモのページは `v0.1.0-alpha` で書かれているようである。

`v0.1.0-alpha` には以下の問題があった。

Local 未対応なのでSFTPを使った

peg-out Graph ID を割り振るトランザクションの展開後、SFTPで peg-out Graph IDが載ったデータを転送できていない

peg-in confirmトランザクションが fee不足で展開できない

これは私の `bridge -n -u <TXID:n> -d <EVMADDR>` で与える `EVMADDR` に間違いがあったためだった。1文字多い文字列にしていて、peg-in confirm transaction のスクリプトがこれを文字列のまま埋め込んでいたため 1byte 多くなって fee が不足していた。

データ長はチェックしてくれよ～、と自分のミスを棚上げにする

コードに問題があるのか環境に問題があるのか分からなかったが、Windows11 WSL2上で動くようにした(SFTPサーバは Raspberry Pi4上)。

https://github.com/hirokuma/BitVM/commits/demo/v0.1.0-alpha/

Rejected Disprove Scenario (a.k.a. 'happy peg-out' execution path)

チャレンジしたが拒絶された＝問題がなかった、ということだと思われる。

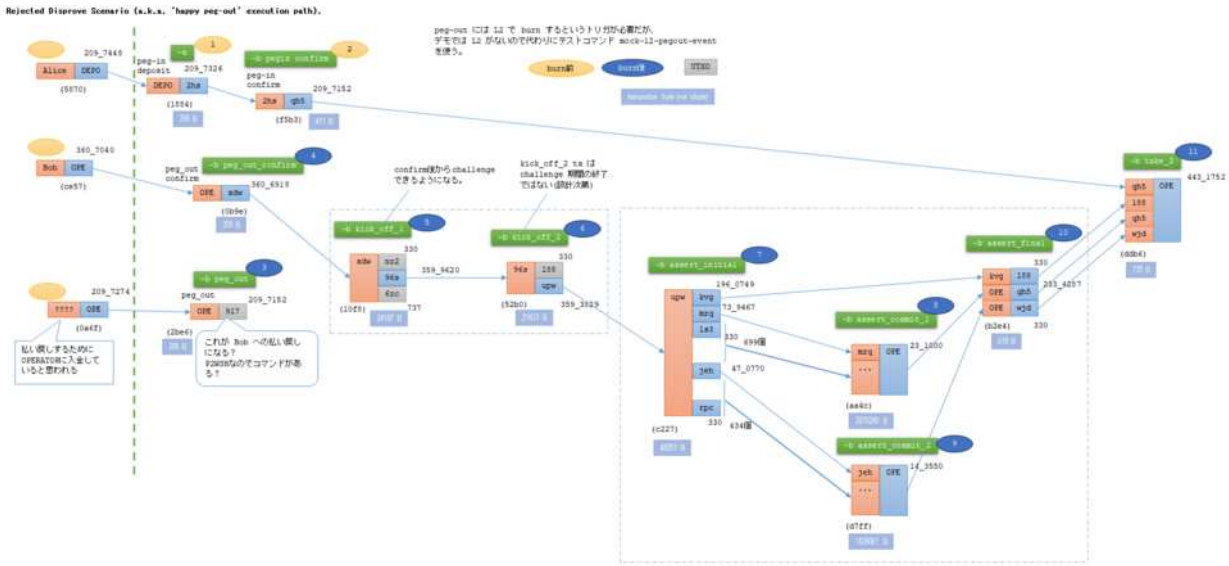
ログ

ユーザはトランザクションを計 3つ展開する。

- peg-in --> DEPOSITOR
  - peg-in Graph ID
- peg-out confirm –> OPERATOR
  - peg-out Graph ID
  - なぜ peg-out Graph ID を得るのに peg-in Graph ID を必要とするのだろう？ 同一人物？
- peg-out –> OPERATOR

運営側は 11トランザクションを展開する

- これは disprove(反証)に失敗するパターンなので、disprove したトランザクションは出てこない



PDF版

実際のトランザクションを図にしただけだが、GOAT BitVM2 pdf の Fig.2 も同じなので間違えていないだろう。

- peg-in
  - DEPOSITORに送金(`<TX-A>`)
    - Alice が送金したことになる
  - `-n(initiate-peg-in) -u <TX-A> -d <EtherAddr>` ： peg-in Graph IDを得る
    - (1) peg-in transaction が展開される

size=206、vsize=122

OPERATORに送金(`<TX-B>`)

これは誰が送金する？ Bobでよいの？

`-t(create-peg-out) -u <TX-B> -i <PEGIN_ID>` : peg-out Graph IDを得る

peg-in Graph IDを紐付けられる

TX-B が Bob だとしたら peg-in と紐付ける理由がよくわからない

まあまあ時間がかかる

peg-in Graph ID

`-c(push-nonces) -i <PEGIN_ID>` : nonce の push

同じ操作を VERIFIER_1 でも行う

`-g(push-signatures) -i <PEGIN_ID>` : signature の push

同じ操作を VERIFIER_1 でも行う

peg-in confirm

`-b(broadcast) pegin -g <PEGIN_ID> confirm` :

(2) peg-in confirm transaction が展開される→ `<TX-D>`

size=411、vsize=174

これで peg-in が確認されたら L2 に mint されて wrapped BTC が使えるようになる

おそらくこれより前までであれば peg-in transactionから取り戻すことができる

peg-out Graph ID

`-c(push-nonces) -i <PEGOUT_ID>` : nonce の push

同じ操作を VERIFIER_1 でも行う

`-g(push-signatures) -i <PEGOUT_ID>` : signature の push

同じ操作を VERIFIER_1 でも行う

Burn(仮想)

`-x(mock-l2-pegout-event) -u <TX-D>`

特定の peg-in confirm transaction を Burn させる

Bob は wrapped BTC(L2 上の BTC)を Burn して Bitcoin に戻そうとする

デモでは L2 がないので `mock-l2-pegout-event` で代用する

`bridge interactive` でインタラクティブモードにしないといけないらしい。これ以降？ このコマンドだけ？

peg-out

`-b(broadcast) tx -g <PEGOUT_ID -u <TX-E> peg_out`

事前に OPERATOR に送金して `<TX-E>` を用意しておく

Bob が L2 で Burn した分を OPERATOR がL1 で払い戻す

(3) peg-out transaction が展開される

size=206、vbyte=122

m人のOPERATORのうち1人が自分のお金で BTC を前払いする

OPERATOR 間の競争が発生するようなことが書かれている(デモでは1人なのでわからない)

この後の行程で問題が無ければ取り戻せるらしい

問題があったら前払いした方が悪いということになるそうだ

peg-out した BTC を誰に払うのかは L2 上で Burn したときに指定するらしいが、今回はよくわからず

❓ 直接 Bob に払わず P2WSH アドレスになっていた。デモの最後まで UTXO のままだったので L2で指定したアドレスがこの P2WSH だったのか、あるいはこれから払い戻しの作業が別途あるのかわからなかった。

peg-out confirm

```
-b tx -g <PEGOUT_ID> peg_out_confirm
```

(4) peg-out confirm transaction が展開される → `<TX-F>`

size=205、vbyte=122

INPUT は OPERATOR に peg-out Graph ID を作るときに指定した `<TX-B>`

以降の工程は peg-out confirm transaction から始まる

kick off

assert

take

kick off

BitVM2上で検証されるプログラム（Graph）と主張（Claim）の開始処理(kick off)

「この実行結果は正しい」という主張（Claim）をオンチェーンにコミットすること

あるGraph IDの回路に基づいて、入力と出力の正当性を主張するトランザクション

以後その主張に対して、他者がチャレンジ可能な状態になる

peg-out で支払った OPERATOR にとっては資金の回収作業になる(L2 で回収?)

kickoff-1

```
-b(broadcast) tx -g <PEGOUT_ID> kick_off_1
```

(5) kickoff_1 transaction が展開される → `<TX-G>`

size=24187、vsize=6182

INPUT は peg-out confirm transaction の `<TX-F>`

コミットメント期間とチャレンジ期間の始まりになる

kickoff-2

```
-b(broadcast) tx -g <PEGOUT_ID> kick_off_2
```

(6) kickoff_2 transaction が展開される → `<TX-H>`

size=21433、vsize=5461

INPUT は kickoff_1 transaction の `<TX-G>`

INPUT が TimeLock されているようで少し待つ必要があった

assert

assert は「断言」の意味らしい

Verifiable Claim（主張）の検証対象となる回路プログラムのステップ的な検証フェーズ

オンチェーンでチャレンジなしでも（＝non-interactive）正当性を確定できるために、段階ごとに分割されている

`v0.1.0-alpha` では 4つのコマンドに分かれているが、

assert-initial

```
-b tx -g <PEGOUT_ID> assert_initial
```

(7) assert_initial transaction が展開される → `<TX-I>`

size=49106、vsize=48953

INPUT は kickoff_2 transaction の `<TX-H>`

Graph IDの確認

提出された入力・出力コミットメントがGraphに対して整合的かを確認

assert-commit-1

```
-b tx -g <PEGOUT_ID> assert_commit_1
```

assert_commit_1 transaction が展開される → `<TX-J>`

size=2870280、vsize=739137

INPUT は assert_initial transaction の `<TX-I>`

assert-commit-2

```
-b tx -g <PEGOUT_ID> assert_commit_2
```

assert_commit_2 transaction が展開される → `<TX-K>`

size=1828061、vsize=470433

INPUT は assert_initial transaction の `<TX-I>`

assert-final

```
-b tx -g <PEGOUT_ID> assert_final
```

assert_final transaction が展開される → `<TX-L>`

size=618、vsize=351

INPUT は assert 関係の `<TX-I>` 、 `<TX-J>` 、 `<TX-K>`

disprove

反証。今回は失敗する。

```
-b tx -g <GRAPH_ID> -a <BTC_ADDRESS> disprove
```

? "attempt to subtract with overflow" で panic になったが、それがここでの失敗という意味で良いのか？

take

take_1

デモでは実行していない

take_1 は peg-in 向けの処理で、デモでは peg-out のことだけだったのでやらなくても影響ない

それならやってしまえばよいのでは？

take_2

```
-b tx -g <PEGOUT_ID> take_2
```

peg-out 向けの処理

take_2 transaction が展開される

size=735、vbyte=347

INPUT は peg-in confirm transaction の `<TX-D>` と assert_final transaction の `<TX-L>` (vout が3つあるが 3つとも)

Successful Disprove Scenario (a.k.a. 'unhappy peg-out' execution path)

Gemini にビルドされたプロジェクトのことを訊いてみる。
なお、bridge も fuzz も bitvm を参照するプロジェクトになっている。

bitvm

主に以下の目的を持つと考えられます。

Bitcoin スクリプトの操作と実行: `bitcoin-script` 、 `bitcoin` 、 `bitcoin-scriptexec` への依存は、Bitcoin のスクリプト言語を扱ったり、トランザクションを構築・検証したりする機能を示唆しています。

高度な暗号技術の利用: `ark-*` (arkworks) 関連のクレート (`ark-bn254` , `ark-ff` , `ark-ec` , `ark-groth16` など) は、ゼロ知識証明やその他の高度な暗号プロトコルの実装に使われるライブラリです。これは、複雑な計算の検証やプライバシー保護技術を Bitcoin 上で実現しようとしている可能性を示します。

計算と検証: プロジェクト名「bitvm」 (Bitcoin Virtual Machine の略と推測される) と上記の暗号ライブラリの組み合わせから、Bitcoin ネットワーク上で何らかの計算を実行し、その結果を検証する仕組みを構築していると考えられます。

総じて、 `bitvm` プロジェクトは、Bitcoin のスクリプトシステムと先進的な暗号技術 (特にゼロ知識証明) を組み合わせて、Bitcoin 上でより複雑なロジックや計算を実行・検証するための基盤技術やツールを開発しているプロジェクトであると推測されます。 `fuzzing` feature の存在

は、この複雑なシステムの堅牢性を確保するためにファズテストが行われていることを示しています。

bridge

主に以下の機能を持つと考えられます。

- BitVM と Bitcoin ネットワークの連携: `bitvm` クレートや `bitcoin` 関連のクレートに依存していることから、BitVM のコンセプトを Bitcoin ネットワーク上で実現、または関連付ける機能を持つ可能性があります。
- 高度な暗号技術の利用: `ark-*` (ゼロ知識証明関連ライブラリ) や `musig2` (マルチシグネチャ) などの依存関係から、プライバシー保護や複雑な検証ロジックの実装のために、これらの先進的な暗号技術を利用していると考えられます。
- 外部システムとの通信・連携: `esplora-client` (Bitcoin ブロックチェーンエクスプローラー)、 `aws-sdk-s3` (Amazon S3)、 `suppaftp` / `openssh-sftp-client` (FTP/SFTP) などの依存関係から、外部のブロックチェーンデータソースやストレージサービスと連携する機能を持つ可能性があります。
- コマンドラインインターフェースの提供: `clap` クレートに依存し、 `bridge` と `bridge-query` という2つのバイナリを定義していることから、ユーザーがコマンドラインを通じて操作できるツールを提供していると考えられます。

総じて、 `bridge` プロジェクトは、BitVM を基盤とし、Bitcoin ネットワーク上で高度な暗号技術を活用しながら、外部システムとも連携する機能を提供するソフトウェアであると推測されます。プロジェクト名が示す通り、異なる要素間の「橋渡し」をする役割を担っている可能性が高いです。

fuzz

ファズテストを行うためのものです。

`Cargo.toml` ファイルから以下の点が分かります。

- `[package.metadata]` と `cargo-fuzz = true` : これは `cargo-fuzz` というファズテストツールを使用することを示しています。
- 依存関係:
  - `libfuzzer-sys` : Fuzzing エンジンである libFuzzer との連携を提供します。
  - `arbitrary` : ランダムなテストデータを生成するためのライブラリです。
  - `bitvm = { path = "../bitvm", features = ["fuzzing"] }` : `bitvm` クレートの `fuzzing` feature を有効にして依存しており、 `bitvm` クレートの機能をファズテストの対象としていることを示唆しています。
- `[[bin]]` セクション:
  - `blake3` や `bigint_transform` といった名前のバイナリが定義されており、これらは個

別のファズターゲット（ファズテストの実行単位）です。

`path` が `fuzz_targets/` ディレクトリを指していることも、ファズテストの典型的な構成です。

したがって、このプロジェクトは `bitvm` クレートや、 `blake3` ハッシュ関数、 `bigint` (大きな整数) の変換処理などに対して、予期せぬ入力やエッジケースを自動的に生成して与えることで、バグや脆弱性を見つけ出すことを目的としています。

ブロックヘッダの検証

デフォルトでビルドされなかった header-chain と prover と final-spv はmainnet などのブロックヘッダに関するプロジェクトのようだ(header-chain と final-spv の中にそれぞれ guest プロジェクトもある)。

RISC Zero の `rzup install` がいるらしい。

https://dev.risczero.com/api/zkvm/install