BitVM2でGroth16 verifierを作る

BitVM

https://bitvm.org/bitvm2

https://bitvm.org/snark

最後の更新が2024年4月で、"A plan to implement a SNARK verifier in Bitcoin Script to run it in BitVM2"としか書いていないのでBitVM1のときの内容だろう

Geminiに相談

Geminiに相談2

現在のBitVMリポジトリの推測 2025/07/05

`bitvm/`

オンチェーンで使うことになる関数を持つライブラリ(クレート?)

`bitvm/src/groth16/` は zkSNARK Groth16 の verifier 実装。検証キーはハードコーティングされている。

あくまでライブラリに過ぎないのでアプリは自作が必要。

`bridge/`

BitVM bridgeのデモ実装

`final-spv/`

RISC Zero zkVM用

`prover/` が使うことになっているが、テストコードでしか使われていなさそう。

TXID と merkle proof からブロックに内包されていることを証明する `verify()` が実装されているらしい

`header-chain/`

RISC Zero zkVM用

`prover/` が使う

実行内容

Bitcoinブロックヘッダの連続したリストを受け取り、

`bits` , `target` , `target_bytes` が範囲内にある

`previous_blockhash` が正しい

ZeroSync/header_chainにインスパイアされたりしてる

RISC Zero zkVM上で実行することで、その中では普通にチェックするプログラムを動かすだけで Groth16 証明データになって出てくる

`header-chain/src/lib.rs` の `header_chain_circuit()` が最初に呼ばれるようだ。

`prover/`

`bitvm/src/groth16` の検証キーに対応する証明キーで zkSNARK Groth16 証明データを作成する実装サンプル。

Bitcoinブロックヘッダのファイルを与えて、前回までの証明データとブロック数を与えると、証明データが正しいことを検証してから追加になるブロック数分 or ブロック数を追加した証明データを作る。

証明データのファイルサイズは微妙に違うがブロック数によって増えるものではなさそうだ。220KBくらい。

ブロックヘッダしか扱っていないので `final-spv` は使わず `header-chain` だけ使用していると思われる

前提情報

BitVMといったとき、BitVM1、BitVM2などの違いを気にすること

ChatGPT などはこれらの情報が混在している可能性が高い

2025/07/02 現在、BitVM1 は deprecated、BitVM3 はPDFが出てきたばかりでリファレンス実装中。BitVM2が今のところ使用可能ということになる。

BitVM bridgeというものがあり、これは BitVM2 を応用したもの。BitVM2そのものではない。

実装に関するドキュメントはほとんど無い

調査での情報

BitVM2は Groth16 verifier が実装されている(Geminiに相談2参照)

つまり、BitVM2 で Groth16 verifier を使うなら流用できる

それに使う検証キーはハードコーディングされている

対になる証明キーを使って証明データを作る必要がある。そこら辺は各自で実装する。

鍵がハードコーディングされているので BitVM2 を使う人は同じ鍵を使うことになるが、それで問題ないそうだ

BitVM2方式

BitVM1では実行したいプログラムを論理回路で表現してBitcoinスクリプト変換し(巨大になる)、分岐ごとに署名することでチェックしていたので実用では使えなかった(認識が間違ってるかも)。

BitVM2ではその巨大なBitcoinスクリプトを1トランザクションに収まるようなサイズで分割し、1つのスクリプトの結果が次のスクリプトへの入力になって続いていくような形にする。そして最後のスクリプトの結果(か、そのハッシュ値など)をオンチェーンに載せる。
もしその結果を見て文句を言う人がいなかったら、それで終わり。
その結果が不正だと思う人がいたら、それを主張して(Challenge)正しいことを相手に説明させる。分割したスクリプトをオンチェーンで示しながら間違っていないことを断言していく(Assert)。オフチェーンで計算するのに分割するのはこのときのためである。
Challengeはこれだと攻撃のようにできてしまうので保証金を必要としている。もし Challenge が失敗した場合はその保証金を没収される。

README

現時点のREADMEだが、BitVM2のことは書き漏らしや修正不足がありそう？

- groth16 SNARK verifier が実装されているとあるが使えるのか分からない
- コンポーネント
  - `u32` 、 `u4`
  - ハッシュ関数
  - Bit Integerと演算
  - BN254: 楕円曲線。アルゴリズム9(nine)？
  - Groth16: BN254を使った proof verify。スクリプトサイズは 1GB程度。"On Proving Pairings"という論文と関係があるようだ
  - Chunk: Groth16 をチャンクに分割する
  - Signature: Winternitz signature方式？
  - Bridge: ブリッジ
- BitVM CLIとあるが、どうもブリッジのコマンドを指しているようだ

現在のBitVMリポジトリでのビルドされる実行ファイル

デフォルトでビルドされた実行ファイル

- `bitgint_transform` : `fuzz/Cargo.toml`
- `blake3` : `fuzz/Cargo.toml`
- `bridge` : `bridge/Cargo.toml`
  - BitVM2 bridgeのデモ実装
- `bridge-query` : `bridge/Cargo.toml`

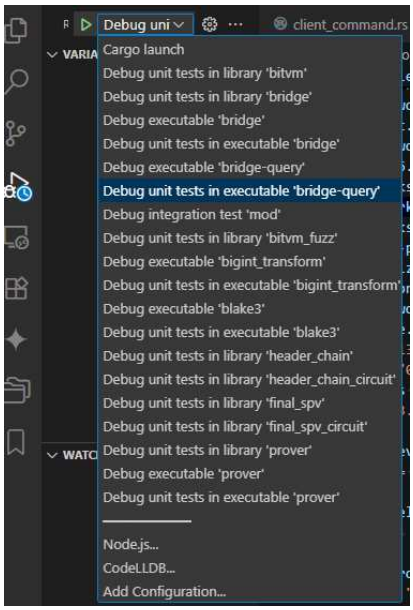
bridge

`bridge` はブリッジのデモを動かすコマンドラインツール。使った内容はBitVM bridgeのデモ

v0.1.1-b822f19を参照のこと。

`bridge-query` はブリッジの状態を問い合わせるコマンドラインツール。

fuzzテスト



`fuzz/` は Rust cargo-fuzzでファジングテストをする - DevelopersIO を参照のこと。 `cargo-fuzz` を使ったテスト。

実行すると、おそらく適当なランダム値を入れてテストをし続ける。

`fuzz/bigint_transform`

```
$ ./target/debug/bigint_transform
WARNING: Failed to find function "__sanitizer_acquire_crash_state".
WARNING: Failed to find function "__sanitizer_print_stack_trace".
WARNING: Failed to find function "__sanitizer_set_death_callback".
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2451254579
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096
bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED exec/s: 0 rss: 27Mb
WARNING: no interesting inputs were found so far. Is the code instrumented for
coverage?
This may also happen if the target rejected all inputs we tried so far
#2048   pulse  corp: 1/1b lim: 21 exec/s: 682 rss: 29Mb
#4096   pulse  corp: 1/1b lim: 43 exec/s: 585 rss: 29Mb
#8192   pulse  corp: 1/1b lim: 80 exec/s: 546 rss: 30Mb
......
```

いろいろ WARNING が出ているが今回は無視する。CPU使用率が100%近くなるので注意。

それ以外

デフォルトでビルドされないのは `final-spv/` 、 `header-chain/` 、 `prover/` である。

`cargo build --workspace` で全部ビルドさせようとしたが、 `header-chain/header-chain-guest` 、 `final-spv/final-spv-guest` は失敗した。RISC Zero のインストールが必要らしい。C++なども使うためか `rzup install rust` では足りない。

`risc0`

```
$ curl -L https://risczero.com/install | bash
($HOME/.risc0/bin にインストールされる)
$ rzup install
$ cargo risczero --version
cargo-risczero 2.1.0
```

このあと `cargo build --workspace` でビルドできたが、環境変数によってビルドで生成されたファイルをコピーするような挙動もあるので個別に README に従ってビルドする方がよいだろう。

RISC Zero

RISC Zeroは何をしているのか

Geminiによると、zk verifier のような複雑なプログラムを実行した証明(レシートと呼ぶそうだ)を RISC Zero の仮想VM で動くプログラムを書きさえすれば Circom みたいな言語を使わなくても生成してくれるそうだ。

Use Case

zkアプリを Rust や C++ で書いて動かすことができるのがポイントなので、 `final-spv/final-spv-guest/guest/src/main.rs` で呼び出している関数は `final-spv/src/` にある関数ということか。 `header-chain/` も同様。

`final-spv/` と `header-chain/` でビルドされるのはRISC Zero環境で動くプログラム(32bit RISC-V)で、 `prover/` はそれぞれの ELF ファイル(ちょっとしたヘッダがついて RISC Zero zkVM のイメージと呼ばれる)を取り込んで使っている。

zkVMのイメージには識別できるよう IDがついていて method IDと呼ばれる。

RISC Zero の method ID

`final-spv/src/lib.rs` でハードコーディングされているHEADER_CHAIN_GUEST_IDの値は RISC Zero zkVM上で使われるイメージIDで**method ID** と呼ばれる。

`chain-headerのビルド`

```
(前略)
warning: header-chain-circuit@0.1.0: Logical ELF path for ID computation: "prover/
elfs/mainnet-header-chain-guest.bin"
```

```
warning: header-chain-circuit@0.1.0: Attempting to read ELF file from: "/mnt/wsl/
home2/ueno/Bitcoin/BitVM/prover/elfs/mainnet-header-chain-guest.bin"
warning: header-chain-circuit@0.1.0: Computed method ID:
Digest(28eaea408955009568e6426210805de80be9ff49e80647483b6b351fbc09368e)
warning: header-chain-circuit@0.1.0: Computed method ID words: [1089137192,
2499827081, 1648551528, 3898441744, 1241508107, 1212614376, 523594555, 2385906108]
    Finished `dev` profile [optimized + debuginfo] target(s) in 48.46s
```

同じファイルをビルドすると method ID が毎回同じなので、ランダム要素は無いのだろう。同じ Rustファイルをビルドしているのに `BITCOIN_NETWORK` によって method ID が異なるのは、Rust にそういうしくみがあるからとのこと。option_env!()がそれで、環境変数を参照して分岐を書くが、ビルドしたときにはそのときの環境変数で通る箇所以外は捨てられるとのこと。

使っているのはここである。

```
guest.verify(HEADER_CHAIN_GUEST_ID, &input.block_header_circuit_output);
```

header-chain (RISC-V)

header-chain/README.md

> このライブラリには、ビットコインのヘッダーチェーンを証明するために必要な機能が含まれています。

`build`

```
$ REPR_GUEST_BUILD=1 BITCOIN_NETWORK=mainnet cargo build -p header-chain-circuit
```

`REPR_GUEST_BUILD=1` を付けてビルドすると `prover/elfs/mainnet-header-chain-guest.bin` にコピーされる。

```
./target/riscv-guest/header-chain-circuit/header-chain-guest/riscv32im-risc0-zkvm-
elf/docker/header-chain-guest.bin
```

この ELF ファイル(正確には method ID)は `final-spv/` のビルド時にハードコーディングで埋め込まれる。

final-spv (RISC-V)

final-spv/README.md

> このライブラリには、ビットコインのヘッダーチェーンを検証し、指定されたヘッダーチェーンにビットコイントランザクションを含めることを証明し、Blake3を使用して32バイトにパブリック値を圧縮するために必要な機能が含まれています。

ディレクトリ名は `final-spv/final-spv-guest` だがワークスペース名は `final-spv-circuit` である。

`REPR_GUEST_BUILD=1` を付けてビルドすると `prover/elfs/mainnet-final-spv-guest.bin` にコピーされる。Rustのビルドではなく RISC Zero でビルドしていると思われる。 `--release` を付けても付けなくても commit されているファイルとは差分がないようである。

build

```
$ REPR_GUEST_BUILD=1 BITCOIN_NETWORK=mainnet cargo build -p final-spv-circuit
```

RISC Zeroでビルドすると 32bit RISC-V 向けの ELF ファイルがビルドされ、それに何かヘッダを追加して binファイルになっているようだ。READMEには ELFファイルをコピーしそうに書いてあるが実際は binファイルだけである。

```
./target/riscv-guest/final-spv-circuit/final-spv-guest/riscv32im-risc0-zkvm-elf/
docker/final-spv-guest.bin
```

`final-spv/src/lib.rs` で `header-chain/` をRISC Zeroビルドしたときに出力される method ID を使っているので、ビルドする順番としては `header-chain/` が先になるだろう。

prover

prover/README.md

既に commit されている `prover/data/` や `prover/elfs/` のファイルを使ったりもする。

build

```
$ BITCOIN_NETWORK=mainnet cargo build -p prover
```

`prover/data/mainnet-heads.bin` は `wget` で取得している。

単純にブロック#0からのブロックヘッダ 80バイトを連結しただけのデータのようだ。元からあるデータはサイズが小さい。

第1引数：前回の proof file。 `None` は無し

第2引数: 出力する proof file。

第3引数: 証明するヘッダの数

第1引数を検証して、その続きを第3引数分だけ第2引数に証明データとして出力するのか。

この例は、先頭から10ブロック分のproof fileを作る。 `[]Output` は使用した header_chain の method ID のようだ。

first10

```
$ ./target/debug/prover
Usage: <program> <input_proof> <output_file_path> <batch_size>
$ time ./target/debug/prover None prover/data/mainnet_first_10.bin 10
Header chain circuit took 107645 cycles
```

```
Output: [
    1089137192,
    2499827081,
    1648551528,
    3898441744,
    1241508107,
    1212614376,
    523594555,
    2385906108,
]
Receipt saved to prover/data/mainnet_first_10.bin

real    0m57.231s
user    13m20.508s
sys     0m5.494s
```

試しに input file の `mainnet_first_10.bin` をバイナリエディタでちょっと書き換えてみたところちゃんとエラーになった。

error

```
$ ./target/debug/prover prover/data/mainnet_first_10.bin prover/data/
mainnet_10-20.bin 10
Header chain circuit took 124502 cycles

thread 'main' panicked at prover/src/lib.rs:118:10:
called `Result::unwrap()` on an `Err` value: control_id mismatch:
73fe5c0b44b1580f03e6bf6b7186732a6c79d41348a72f673ff2172de3a0f552
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

proverの役割

Bitcoin L1のブロックを証明しても仕方が無いだろうから、これは L2向けのサンプルという位置づけだろうか。それにしては mainnet/testnet/signet などの切り分けができるようにできているのは細かすぎる気がする。デフォルトではビルドされないので bridge で使っているわけではないだろう。

ブロックヘッダだけのデータを取得しているのだが、 `final-spv/` の中には Merkle tree を扱っているし、 `header_chain.rs` のコメントにZeroSyncのコードを参照していると書いてあるので、Light Client 的なことをやっているのか？

final-spvに対するissueを見ると、SPV verifier であるような書き方になっている。

final-spvのテストではsignetのブロックデータがべたっと書かれているので、そういうデータを

どこからか入手できる前提だ。

final-spv は proverではテストにしか使われていないので、証明しているのはブロックヘッダの正当性だけとなる。