The Problem with ASICBOOST

Jeremy Rubin

April 9, 2017

1 Intro

Recently there has been a lot of interesting material coming out regarding ASICBOOST. ASICBOOST is a mining optimization that allows one to mine many times faster by taking advantage of a quirk of SHA-256. There are multiple ways of implementing ASICBOOST, and recent claims are that one which is incompatible with upgrading Bitcoin is being used.

I thought it was a bit confusing, so I hand wrote some notes for myself. They became somewhat popular, so I decide (by popular request) to LaTeX them.

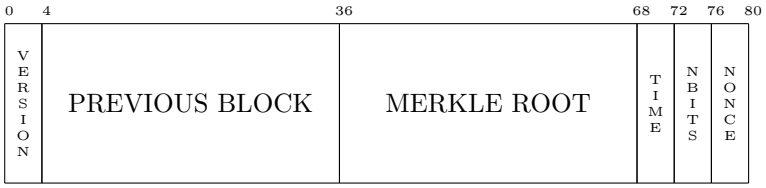
Enjoy!

Update Log:

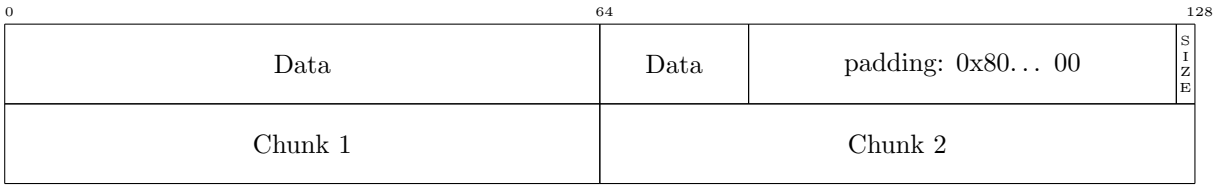
- April 9th: Fixed a few typos/formatting errors. Clarified witness commitment diagram to be more accurate.
- April 9th: Posted LaTeX Notes.
- April 6th: Posted Original Handwritten Notes.

2 Basics

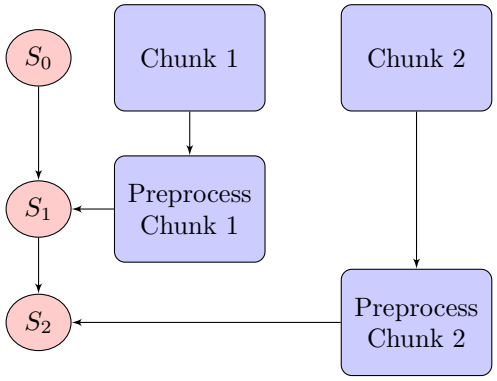
A Bitcoin Header looks like this:



A SHA-256 Hash of 80-Bytes looks like this:



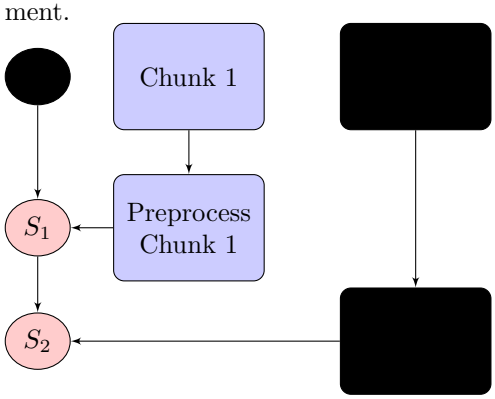
The Hash computation has the following control flow:



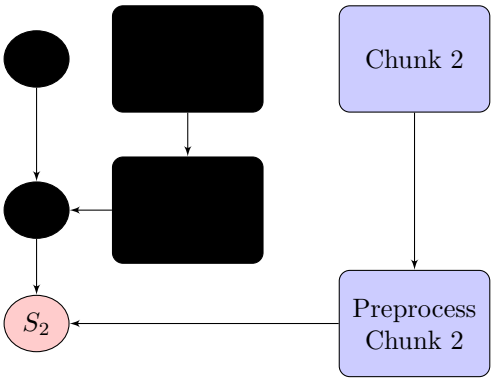
3 Reduced Work Updates

ASICBOOST takes advantage of the following reductions in work if you modify the first or second half of the header only. In the following diagrams, I've blacked out parts that aren't recomputed.

Modifying Chunk 1 Only Modifying only chunk 1 gives a small improvement.

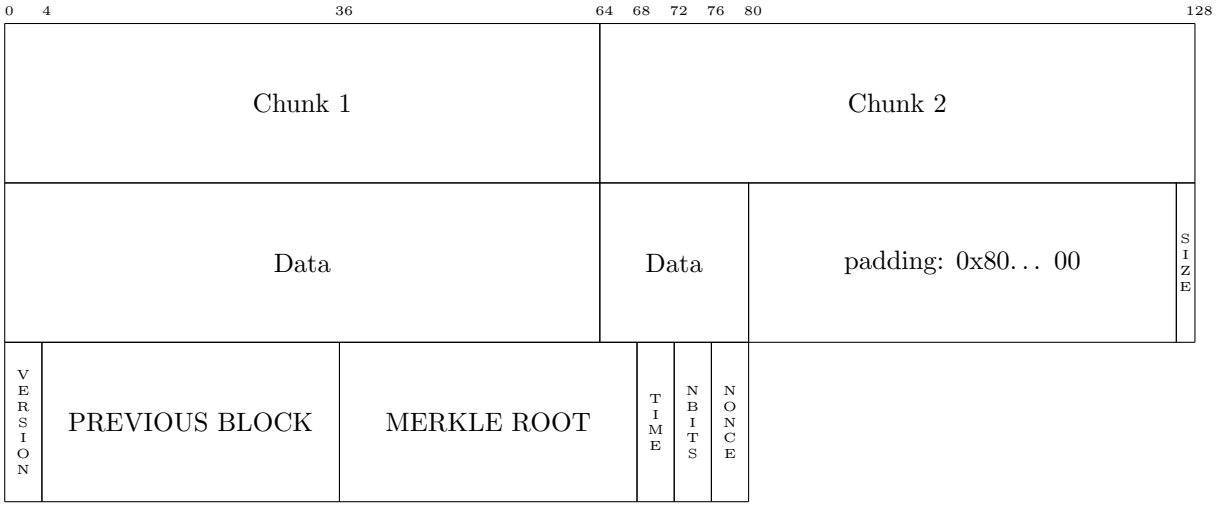


Modifying Chunk 2 Only Modifying only chunk 2 gives a huge improvement.



3.1 How do headers get hashed?

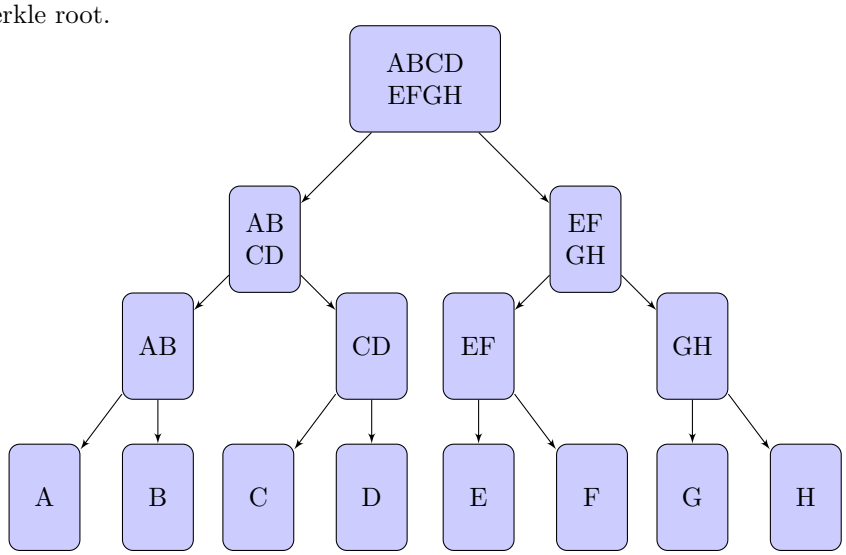
Let's juxtapose the header and hash alignment.



The Merkle root commitment is in both chunk 1 and chunk 2. The first 28 Bytes are in chunk 1, the remaining 4 bytes are in chunk 2.

3.1.1 What is in the Merkle root?

At each level, the hash of the concatenated strings is computed. Each letter $A \ldots H$ is the hash of a transaction. The node $ABCDEFGH$ is called the merkle root.

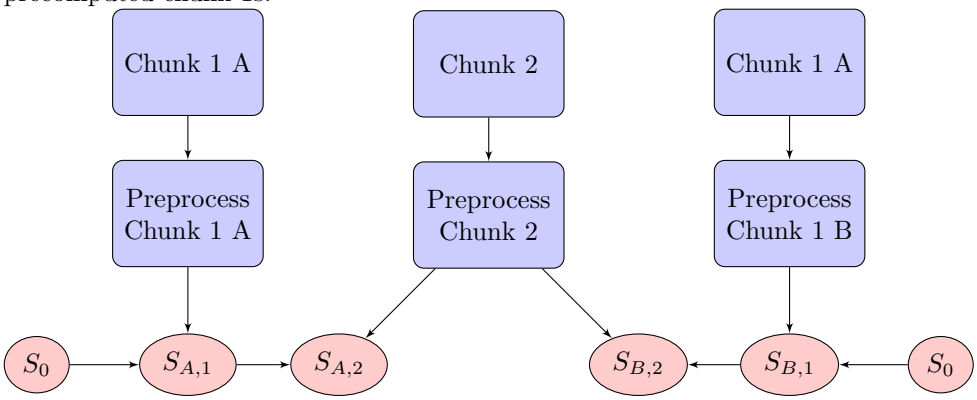


If any of the underlying data $A \ldots H$ are either modified or reordered, the

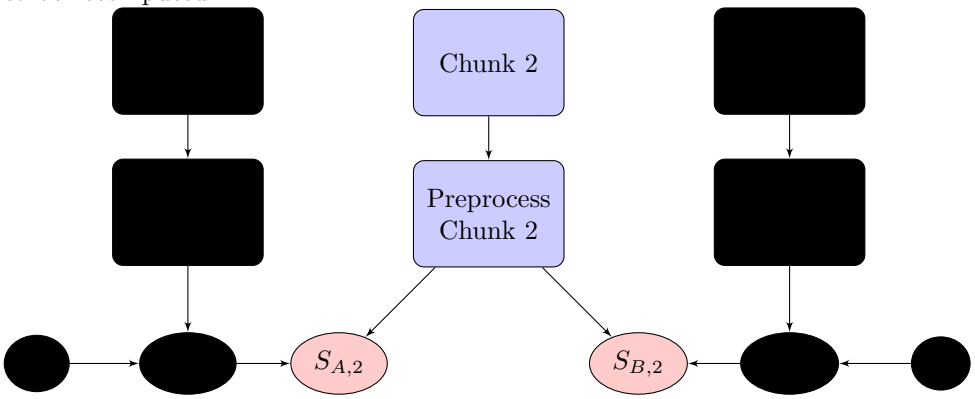
Merkle root will have a different value (uniformly random).

3.2 What if we find two merkle roots with the same last 4 bytes?

Now we can run our very efficient algorithm to only modify chunk 2 on many precomputed chunk 1s.



So now for a little extra setup work, we get a $2\times$ hash rate multiplier for every nonce we try in chunk 2. The below diagram shows the part that doesn't need to be recomputed.



If we find N chunk 1s, we can use the same trick for an $N\times$ hash rate multiplier.

4 Practical Collision Generation

How do we generate colliding (in the last 4 bytes) transaction tree Merkle commitments efficiently?

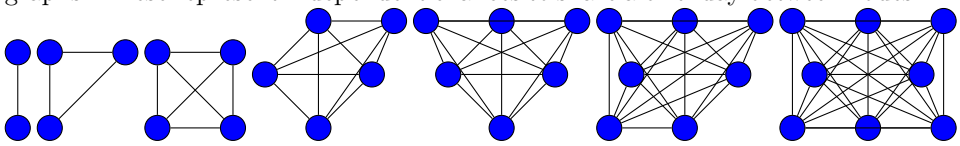
4.1 Birthday Paradox

The Birthday Paradox says that if 23 people are in a room, there is a 50% chance that at least two people share a birthday.

This is the same problem as our tree collision, but with different numbers.

It works because as the number of people increases, the number of independent chances of sharing a birthday increases more quickly.

You can visualize this as counting the number of connections in the following graphs. These represent independent chances to share a birthday between nodes.



The closed form for the number of connections for n nodes is $\binom{n}{2} = \frac{n!}{2 \cdot (n-2)!} = \frac{n \cdot (n-1)}{2} = O(n^2)$ chances

4.1.1 Generalized Birthday Paradox

A general closed formula for computing the number E of entries of T types needed to be P probable to have a C-way collision is hard to find, but the following approximation works well as an upper bound[1].

$$P \approx 1 - e^{-\binom{E}{C} T^{-C+1}}$$

For instance, for the standard birthday paradox problem:

$$P \approx 0.5 \approx 1 - e^{-\binom{23}{2} 365^{-2+1}}$$

4.2 Summary

It's clear that the key in collision hunting is to store and generate a large number of potential matches to compare against. The probability of a collision becomes very likely even with a large set of possible "birthdays" (T), and relatively small numbers of "people" (E).

Using our formula, for 4 bytes of potential "birthdays", at $110k$ "people", the probability of a shared "birthday" is more than 75%.

$$1 - e^{-\binom{110k}{2} (2^{32})^{-1}} > 0.75$$

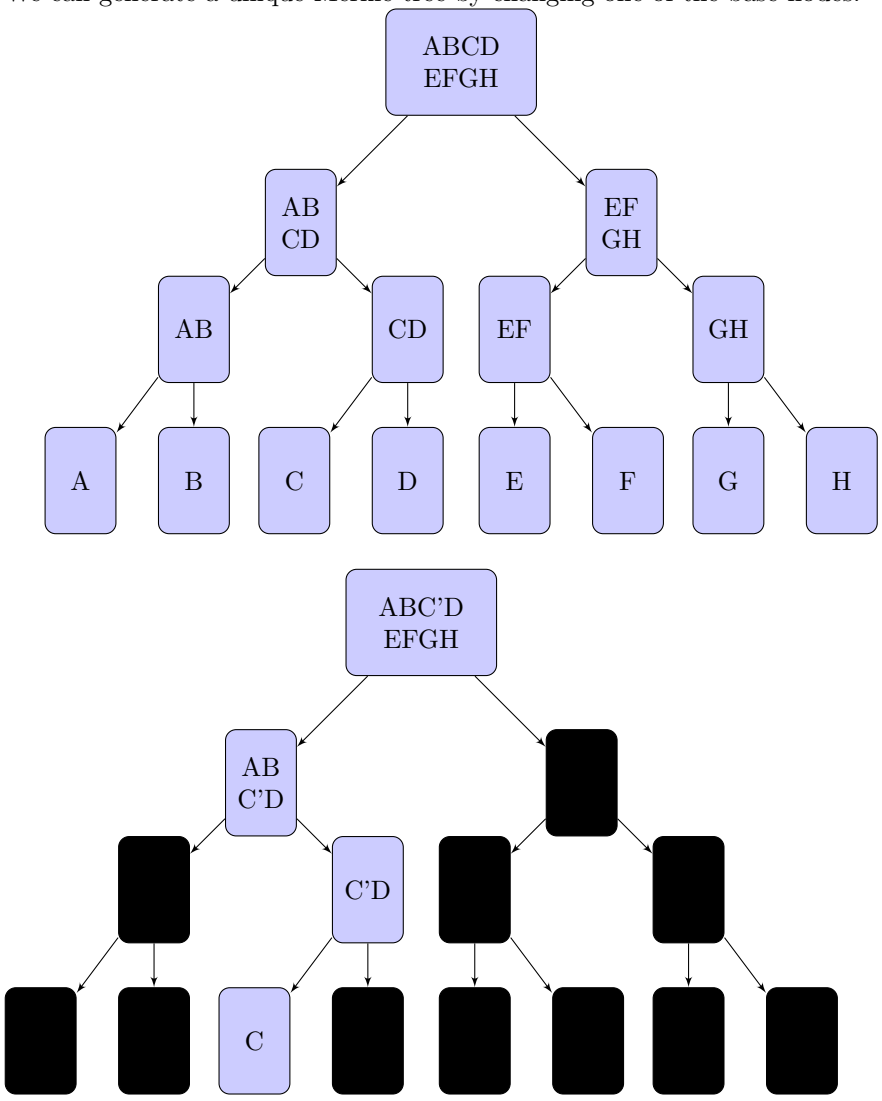
[1] It should over-count. See
https://math.stackexchange.com/questions/25876/probability-of-3-people-in-a-room-of-30-having-the-same-birthday

Creating a last 4 bytes colliding Merkle tree commitments is now reduced to a problem of generating $110k$ unique transaction commitment merkle roots.

5 Generating N Unique Merkle Trees

5.1 Naive Algorithm

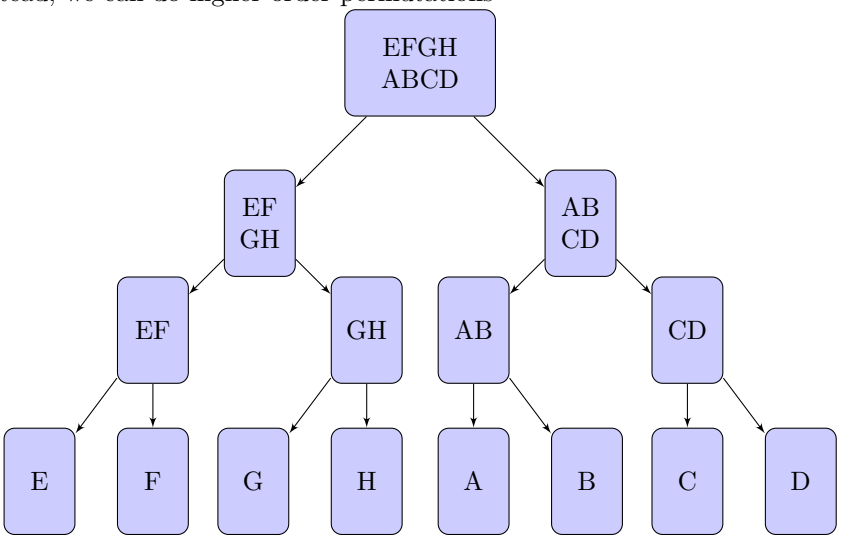
We can generate a unique Merkle tree by changing one of the base nodes.



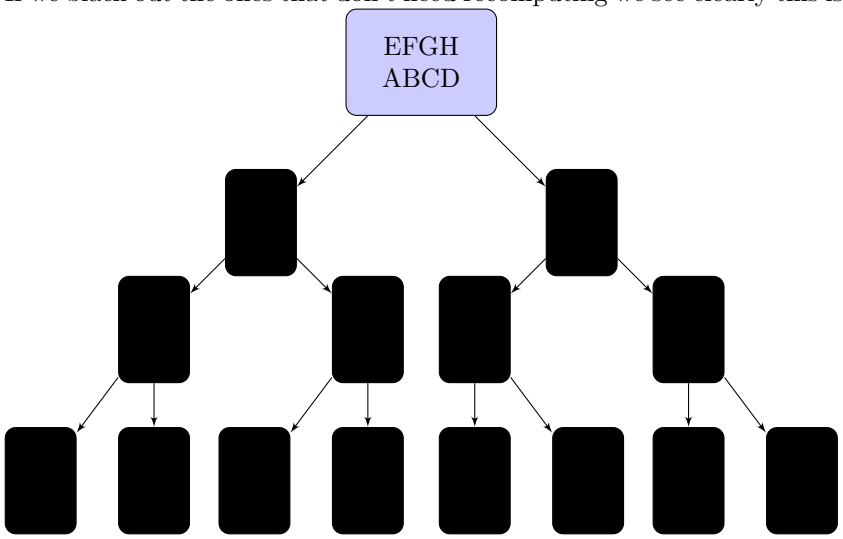
The non-blacked out boxes all need to be recomputed when changing $C \rightarrow C'$. Let m be the number of leaf nodes, the naive algorithm requires $O(N \log m)$ hashes.

5.2 Higher Order Permutations

Instead, we can do higher-order permutations



Now we only need to do one re-computation to get a second potential match. If we black out the ones that don't need recomputing we see clearly this is better



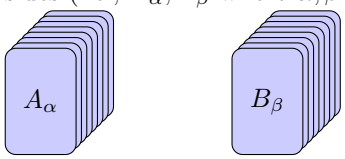
In the naive version, each new potential match is expensive to compute. In the smarter version, we can recursively apply this principle to very efficiently generate potential matches.

There are several strategies for generating candidates, not just swapping. Swapping isn't optimal because Bitcoin transactions have order dependencies, but it is demonstrative of the principles at play.

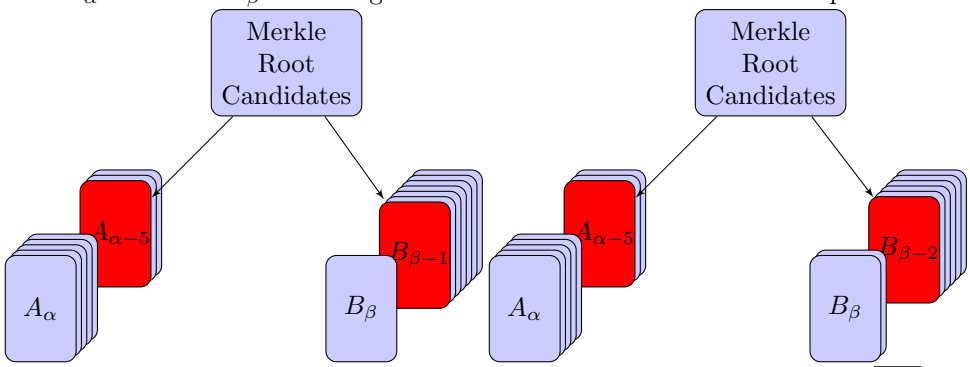
5.3 Efficient Algorithm

Let's say we want to generate R collisions, and the birthday paradox says it is likely with N hashes.

First, we generate $\sqrt{N}$ unique left hand sides and $\sqrt{N}$ unique right hand sides (i.e., A_α, B_β where $\alpha, \beta \in 1 \cdots \sqrt{N}$).



With our $\sqrt{N}$ A_αs and B_βs, we can generate N candidates by combining each A_α with each B_β. The diagram below illustrates the combination process.



We can apply this recursvely: to generate $\sqrt{N}$ A_αs, we generate $\sqrt{\sqrt{N}}$ left hand sides and right hand sides. In the base case, we can modify a single transaction or swap two trivially independent transactions to generate a unique parent.

This algorithm uses $\Theta(N)$ work. This is optimal (for this component of the algorithm), because we need to produce N hashes.

5.3.1 Complexity Proof

For the interested:

At each step we must do n work to create the outputs, and we recurse on 2 times the square root of the output size. Therefore, our recurrence is:

$$T(n) = 2 \cdot T(\sqrt{n}) + n$$

Let

$$n = 2^p$$

Substitute n with 2^p

$$T(2^p) = 2 \cdot T(\sqrt{2^p}) + 2^p$$

$$T(2^p) = 2 \cdot T(2^{p/2}) + 2^p$$

$$S(p) = T(2^p)$$

$$S(p) = 2 \cdot S(p/2) + 2^p$$

This is Case 3 of Master Theorem:

$$S(p) = \Theta(2^p)$$

$$S(p) = T(2^p)$$

$$T(2^p) = \Theta(2^p)$$

$$p = \log n$$

$$T(n) = \Theta(n)$$

This algorithm is also "Embarrassingly Parallel", so can get efficiency gains using multiple cores.

5.4 N-Way Hit

Our earlier approximation predicts that in order to be likely to produce a 4-way collision we need to generate around 2^{25} hashes, and store them for collision detection. This is about a gigabytes worth, so most computers should be able to generate this quickly.

$$0.49 \approx 1 - e^{-\binom{2^{25}}{4}(2^{32})^{-3}}$$

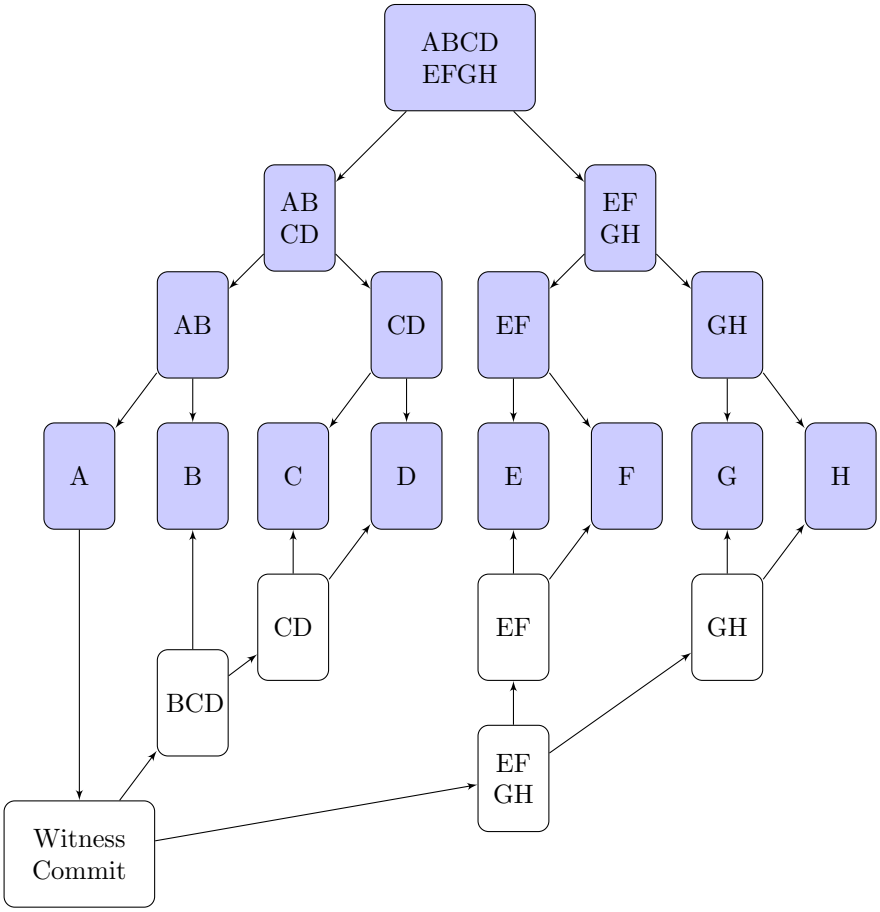
A 5-way and greater, the amount of time required to generate a collision will increase markedly, but I would imagine that miners with ASICBOOST would not want to use that, and prefer to generate many 4-way collisions with rolled coinbase extra-nonces.

6 What does Segregated Witness (SegWit) have to do with it?

In SegWit we generate an additional commitment of all the signatures and we put it into the coinbase transaction.

This commits to which transactions are present in the block and in what order they appear, which means no more easy generation of unique Merkle roots; the commitment is in the leftmost transaction, so modifying the order or contents of any transaction triggers a full a factor of $\log m$ rehash.

The figures below demonstrates this. A commits to $BCDEFGH$. Any change in the order or contents of the tree triggers the update to A's nested commitment, which also triggers an update to $ABCDEFGH$. Note the direction of the arrows for the witness commitment[2].

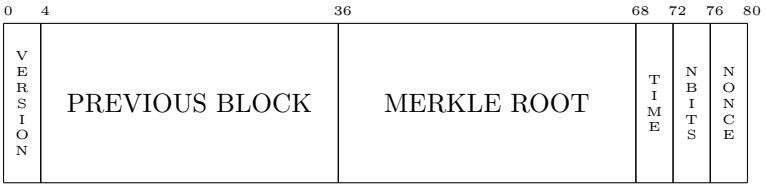


[2]Technically, A is put into the witness tree with a zero value so the structure of the tree is identical. Representing this would make this diagram less clear, because no data from A is committed to.

7 Are SegWit and ASICBOOST are fundamentally incompatible?

No.

Recall our header format...



The version field can be used to make "collisions" trivialy!
Simply set it to a different value.

7.1 Why go through the trouble of finding non-trivial collisions?

- Changing versions is easy to detect.
- Version is already used for... versions.

8 How can we fix this?

There are a few options:

1. **Don't do anything**
   - **Pro:** Any changes to Bitcoin are dangerous.
   - **Con:** Status Quo is obviously harming Bitcoin.
2. **Change SegWit to be compatible with ASICBOOST**
   - **Pro:** If we could start over, it would be easy to do.
   - **Con:** Segwit is already written, reviewed, and adopted in industry.
3. **Block just undetectable ASICBOOST**
   - **Pro:** Solves the immediate problem.
   - **Con:** We wouldn't be blocking it because undetectable optimization is wrong, but because *this specific optimization intereferes with protocol development*. The propensity to conflate the two is dangerous.
4. **Block all ASICBOOST**
   - **Pro:** ASICBOOST is not available to all miners, this levels the field.
   - **Con:** A dangerous precedent to set. Picking winners and losers.
5. **Hard fork Bitcoin to a new header format**
   - **Pro:** Could leave ASICBOOST intact, put SegWit commitment in header, and everything else on the Bitcoin Hard-Fork Wish List.
   - **Con:** Dangerous precedent, risk to split network.