The Satoshi Project

This project is a cryptocurrency block hashing implementation experiment. Following Satoshi's roots, we are targetting to publish a secure decentralized cryptocurrency. The hashing implementation is a two way hash-encrypt with sliced random keys.

Implementation

Incremental Hashing

An incremental implementation has two major components: the state of the current chunk and a stack of subtree chaining values (the “CV stack”). The chunk state is structurally similar to an incremental implementation of the previous versions, and it will be familiar to implementers of other hash functions. The CV stack is less familiar. Simplifying the management of the CV stack is especially important for a clear and correct implementation.

Chunk State

The chunk state contains the 32-byte CV of the previous block and a 64-byte input buffer for the next block, and typically also the compression function parameters t and d. Input bytes from the caller are copied into the buffer until it is full. Then the buffer is compressed together with the previous CV, using the truncated compression function. The output CV overwrites the previous CV, and the buffer is cleared. An important detail here is that the last block of a chunk is compressed differently, setting the CHUNK_END flag and possibly the ROOT flag. In an incremental setting, any block could potentially be the last, until the caller has supplied enough input to place at least 1 byte in the block after. For that reason, the chunk state waits to compress the next block until both the buffer is full and the caller has supplied more input. Note that the CV stack takes the same approach immediately below: chunk CVs are added only after the caller supplies at least 1 byte for the following chunk.

Chaining Value Stack

```
fn add_chunk_chaining_value(&mut self, mut new_cv: [u32; 8], mut total_chunks: u64) {
  // This chunk might complete some subtrees. For each completed subtree,
  // its left child will be the current top entry in the CV stack, and
  // its right child will be the current value of `new_cv`. Pop each left
  // child off the stack, merge it with `new_cv`, and overwrite `new_cv`
  // with the result. After all these merges, push the final value of
  // `new_cv` onto the stack. The number of completed subtrees is given
  // by the number of trailing 0-bits in the new total number of chunks.
  while total_chunks & 1 == 0 {
  new_cv = parent_cv(self.pop_stack(), new_cv, self.key, self.flags);
  total_chunks >>= 1;
  }
  self.push_stack(new_cv);
}
```

This hash is suitable whenever a collision-resistant or preimage-resistant hash function is needed to map some arbitrary-size input to a fixed-length output.

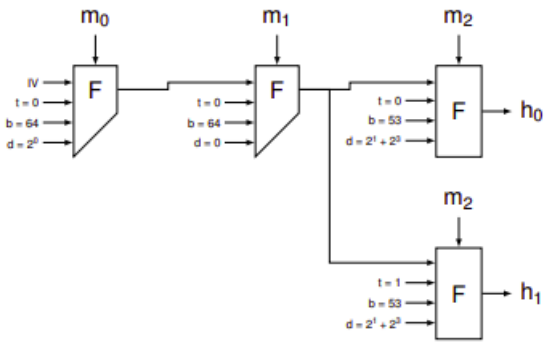
This hashing implementation works with multi-threading: Each chunk can be compressed independently, and one approach to multi-threading is to farm out individual chunks or groups of chunks to tasks on a thread pool.

Tree Structure

In other words, given a message of $n > 1024$ bytes, the left subtree consists of the first

$$2^{10+\lfloor \log_2(\lfloor \frac{n-1}{1024} \rfloor) \rfloor}$$

Parent Node Chaining Values



Block Hashing Steps

The hash implementation described above is the hash of a random generated key. This key will be sliced to `32` characters and it's `hex` is used and the rest is going to be used for a substitute ghost block. For the challenge, the first part `32` character key is only required.

This sliced hash is then passed on to the encryption (24-byte nonce) implementation.

The `u8` of bytes are the `return-value` (encrypted block value) used to decrypt for ledger proofs. (decryption is not performened except for transactions)

For the challenge, the block value is the flag.