BitStream: Decentralized File Hosting Incentivised via Bitcoin Payments

Robin Linus

robin@zerosync.org

November 11, 2023

Abstract

An atomic swap of coins for files would enable an open market for content hosting, in which anyone can monetize their excess bandwidth and data storage capacities, by serving decentralized multimedia services. Verifiable encryption provides a theoretical solution, but the computational overhead is too expensive in practice. We propose a solution to the fair exchange problem, which is highly efficient such that servers can handle large files and manage many clients simultanously. Compatible payment methods include Lightning, Ecash, and every other system that supports hash-timelock contracts. The server encrypts the file such that if there's any mismatch during decryption the client can derive a compact fraud proof. A bond contract guarantees the client receives the exact file or they can punish the server. The transfer happens fully off-chain. An on-chain transaction is required only in case a server cheated.

Implementing the bond contract is possible on mostly any platform such as Liquid, however, on Bitcoin it requires OP_CAT.

1. Introduction

Decentralized file hosting networks lack a well-aligned incentive system. Currently, paid servers for platforms like Nostr often underestimate their operating costs when charging a monthly payment for storing a user's data. Users can split their payment into daily or weekly increments if they don't trust the servers, but this strategy doesn't resolve the economic challenges servers face. Users are paying to upload their data, so servers are not paid per download. If a server fulfills too many download requests from various users, then the server can become overwhelmed from the bandwidth costs outweighing their earnings. In the context of video hosting, where the traditional revenue model may

falter, BitStream's protocol presents a sustainable alternative. For instance, a single user might upload a video once, incurring a one-time cost, but if that video becomes popular and is downloaded 100,000 times, the server's bandwidth costs could skyrocket beyond the initial upload revenue. BitStream's pay-to-download approach offers a solution: it allows the server to charge for each download, ensuring that the revenue scales with the popularity and demand for the media, creating a balanced and profitable ecosystem.

2. Purchasing Decryption Keys

The classical method to perform a fair exchange of files against coins is the following: The server encrypts the file using some sort of verifiable encryption and then sends the encrypted file to the client. The client verifies that it decrypts to the requested file. If it's correct, the clients buys the decryption key from the server in a hash time-locked contract.
Any HTLC allows the client to purchase from the server the *preimage* of a particular *paymentHash*:

$$paymentHash = \text{hash}(\, preimage \,)$$

This mechanism to purchase a decryption key is also fundamental to our scheme. It is simple and compatible with all common Bitcoin payment methods: on-chain transactions, Lightning Network, sidechains like Liquid, as well as Chaumian ecash like FediMint or Cashu.
Purchasing an encryption key is relatively easy. The harder problem is a highly efficient verifiable encryption, which is the objective of the following scheme. Similar ideas have been discussed for altcoins [1] before, but ours is the first bitcoin-focused approach. An example scenario would be a popular Nostr user who makes a post containing the $fileId$ of a video. Potentially millions of clients want to watch that video now.

3. File Identification

A file is identified simply by its Merkle root hash. Files are split into fixed-sized chunks and then hashed into a Merkle tree to derive a unique $fileId$. This is a common technique, usually used in file-sharing networks such as BitTorrent.

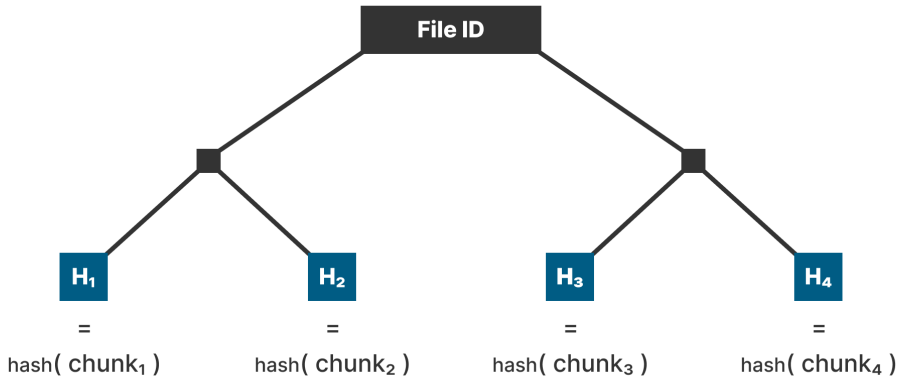

Figure 1: The *fileId* is the Merkle root of the file's chunks. It uniquely identifies the correct file.

Our key observation is that the leaves of the hash tree can be interpreted as a commitment to the individual chunks of the file, which is the basis for our fraud proofs:

$$H_i = \text{hash}(\, chunk_i \,)$$

Note that in practice, this does not provide "zero-knowledge" because a client could try to guess $chunk_i$ from knowing H_i. The file author can solve that by blinding the leaves, e.g., $H_i = \text{hash}(\, chunk_i \,||\, i \,||\, seed \,)$, where *seed* is simply some random blinding factor, attached as a leaf to the Merkle tree after the last chunk of the file. However here, for the sake of explanation, we will use the simplified equation from above.

4. File Encryption

Files are encrypted with a simple one-time pad using the bit-wise XOR operation. The formula for encrypting $chunk_i$ with *preimage* is

$$E_i = chunk_i \oplus \text{hash}(\, preimage \,||\, i \,)$$

and the formula for decryption is almost the same

$$chunk_i = E_i \oplus \text{hash}(\, preimage \,||\, i \,)$$

The server sends the client all encrypted chunks $E_1, \ldots, E_n$ and also the hashes of the unencrypted chunks $H_1, \ldots, H_n$. Additionally, the server commits to the encrypted file in a Merkle tree. The root of that tree is called *encId* and is used to hold the server accountable.

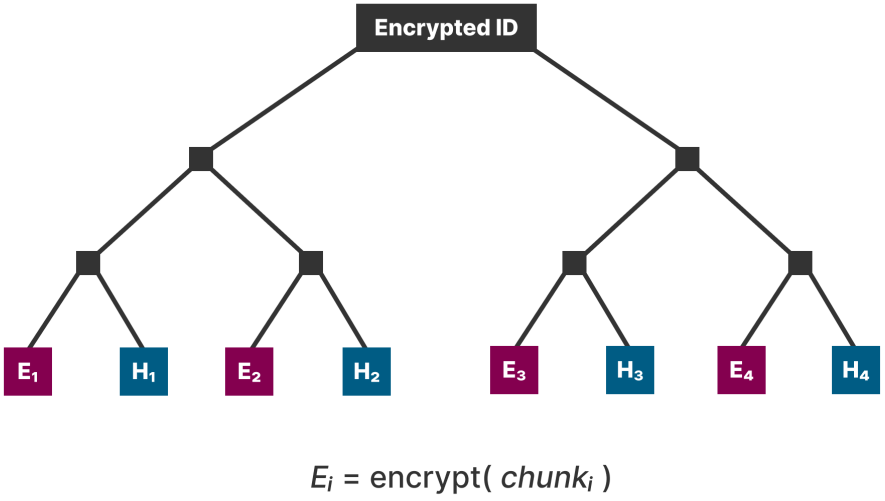

Figure 2: The leaves of the encrypted file tree contain the encrypted chunks and the leaf hashes from the unencrypted file tree.

The client can compute the file's *encId*. Clients can also verify that the commitments $H_1, ..., H_n$ hash to the requested $fileId$. Additionally, with their *signature*, the server commits to their claim *"The preimage of paymentHash decrypts encId to fileId"*

$$claim = encId \ || \ paymentHash$$
$$signature = sign_{server}(claim)$$

This is enough for the client to disprove any incorrectly encrypted file.

5. Fraud Proof

If the encrypted file does not decrypt correctly, the client can derive a succinct fraud proof, which consists of

1. The *signature* for the $claim = encId \ || \ paymentHash$
2. The *preimage* of *paymentHash*
3. A Merkle inclusion proof in *encId* for any pair (E_i, H_i) which does not decrypt correctly.

The fraud equation for an incorrect pair (E_i, H_i) expresses *"Decrypting E_i with preimage does not hash to H_i."*, in other words

$$\text{hash}\Big(\ E_i \oplus \text{hash}(\ preimage \ || \ i \ ) \ \Big) \neq H_i$$

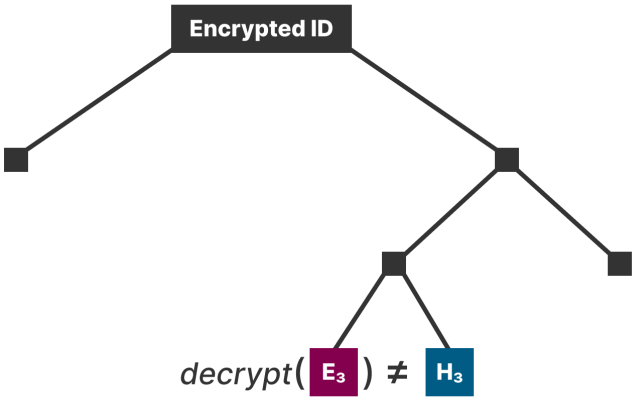

Figure 3: A fraud proof is a Merkle inclusion path for any leaf (E_i, H_i) which doesn't decrypt correctly to the corresponding chunk of the file.

6. Bond Contract and Server Discovery

Fraud proofs are processed by the server's bond contract, which essentially expresses that you can destroy the server's deposit with a fraud proof.
The server registers itself in the blockchain. Clients discover the server by scanning the blockchain. The data in the blockchain is sufficient for clients to verify the validity of the server's contract. This creates a directory of accountable servers from which clients can choose. From this on-chain directory of servers, clients can learn a server's public key, which connects that server's claims to their bond contract.
The contract burns the server's deposit if any client uploads a valid fraud proof, like a justice transaction. Verifying a fraud proof only requires four steps:

1. Verify the server's *signature* for the *claim*
2. Verify the *preimage* of *paymentHash*
3. Verify the Merkle inclusion path for the faulty pair (E_i, H_i)
4. Verify the fraud equation for that pair

Implementing this contract on the Liquid sidechain is relatively straightforward because it provides opcodes for covenants with introspection, OP_CSFS to verify the signature, OP_CAT and OP_MOD for the Merkle path, and OP_XOR for the fraud equation. A defrauded client can trigger the justice transaction without needing to have any coins on the sidechain, because the punished server's deposit pays for all the transaction fees. An example implementation is available [2] and we also demonstrated the execution of a justice transaction on the Liquid testnet [3].

On mainnet it is more complicated, however, it turns out that a single opcode, OP_CAT, [4] is sufficient to implement the bond contract directly on Bitcoin: We can emulate OP_XOR, at least for 32-bit words, by using arithmetic opcodes. The 32-bit words are sufficient as we can concatenate them. We can also emulate OP_MOD for the leaf's index, by providing in the unlocking script the index pre-parsed as a bit string. The only tricky part is to verify the server's signature for the *claim*. Our solution builds on the Schnorr + CAT hack for covenants [5]. We emulate OP_CSFS by putting the *claim* into the Taproot annex because it is hashed into the Server's signature, which means that we can get it onto the stack with the help of the sighash, which we get with Andrew Poelstra's CAT trick.

7. Optimizations

A number of protocol optimizations are possible. For example, large files may be chunked into multiple payments, such that the client has to download only the first subtree before they can start streaming a video. Additionally, clients can download in parallel different parts from multiple servers and seamlessly switch between them in case one goes offline. For the use case of hosting a website, a batch of small files could be combined into a single payment to purchase all required web resources at once. Video live streams are enabled by an author continuously signing fresh file roots, representing the video stream up to the most recent frames.
Responses from servers contain nothing uniquely specific to the client, which implies that the server can precompute the entire response for the next client's request, including the LN invoice. Furthermore, computing the hash tree for *encId* is efficiently parallelizable. Another feature is that during high demand, servers can purchase popular files from each other with BitStream to balance the load.

8. Conclusion

We have proposed an incentive system for decentralized file hosting without relying on trust or heavy-weight cryptography. Client and server perform an atomic swap of coins for files using an optimistic protocol. The server responds with a file that is allegedly encrypted correctly. The client buys the decryption key with a Lightning payment, and if the file doesn't decrypt correctly, the client can financially punish the server for cheating, which is a strong incentive for servers to act honestly. Currently, implementing the bond contract requires workarounds such as using the Liquid sidechain, however, reactivating OP_CAT would be sufficient to implement BitStream directly on Bitcoin.

Appendix: Atomic Swap of Coin for File

Client		**Server**
`knows` *fileId*		`knows` $file = chunk_1 \| \| \ldots \| \| chunk_n$
`request` *file*	$fileId \longrightarrow$	
		`generate`
		$\circ$ *preimage* $\leftarrow$ *random*
		$\circ$ $paymentHash = \text{hash}(preimage)$
		$\circ$ *LN invoice incl. paymentHash*
		`encrypt` *every chunk*
		$E_i = chunk_i \oplus \text{hash}(\, preimage \,\|\| \, i \,)$
		`compute`
		$encId = merkle(E_1, H_1, \ldots, E_n, H_n)$
		`sign` *claim*
		$claim = encId \;\|\|\; paymentHash$
	$(E_1, H_1), \ldots, (E_n, H_n)$	
	paymentHash	
	signature	
	$\longleftarrow$ *LN invoice*	
`verify`		
$fileId \stackrel{?}{=} merkle(H_1, \ldots, H_n)$		
`compute`		
$encId = merkle(E_1, H_1, \ldots, E_n, H_n)$		
`verify` *signature for claim*		
`verify` *LN invoice*		
	$\longleftarrow$ *pay invoice* / *reveal preimage* $\longrightarrow$	
`receive` *preimage*		`receive` *coins*
`decrypt` *every chunk*		
$chunk'_i = E_i \oplus \text{hash}(\, preimage \,\|\| \, i \,)$		
`verify` *every chunk*		
$\text{hash}(chunk'_i) \stackrel{?}{=} H_i$		
`if invalid`		
$\circ$ *derive fraud proof*		
$\circ$ *trigger justiceTX on-chain*		

References

[1] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. Fairswap: How to fairly exchange digital goods. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 967–984, 2018.

[2] Robin Linus. BitStream prototype implementation. `https://github.com/RobinLinus/BitStream`, 2023.

[3] Robin Linus. Demo justice transaction on the Liquid testnet. `https://blockstream.info/liquidtestnet/tx/2e50abbaabd474833d2b61863058e5d1ef93327680428ba5e8d665d983c2dddb`, 2023.

[4] Ethan Heilman, Armin Sabouri. OP_CAT BIP Draft. `https://github.com/EthanHeilman/op_cat_draft/blob/main/cat.mediawiki`, 2023.

[5] Andrew Poelstra. Cat and Schnorr Tricks I. `https://medium.com/blockstream/cat-and-schnorr-tricks-i-faf1b59bd298`, 2021.

Acknowledgments

Special thanks to Colby from the H.O.R.N.E.T. Storage Team, who inspired me to work on this and kept motivating me to complete it. Furthermore, I'd like to thank Tiero from Vulpem Ventures, who helped me implement the bond contract on the Liquid sidechain.

Sponsor BitStream developers: `39hgdBhLsMRpSdQ8HGLoaK7KK4kv7Kz4uH`