Lecture 11: Extracting trust from the blockchain and scaling externally

Principles of Blockchains, University of Illinois,
Professor: Pramod Viswanath
Scribe: Peiyao Sheng

March 4, 2021

Abstract

So far, we have considered scaling methods that alter the consensus protocol itself (albeit in modest ways). In this lecture, we study how to scale existing blockchain performance *without* changing the consensus layer. This is done by extracting trust from the consensus embedded in the core blockchain, but offloading computation and storage. Given that the consensus protocol (layer 1) is untouched, these methods are known to afford *Layer 2* scaling. Two prominent instances are *payment channels* (focused on the UTXO state management system) and *side blockchains* (can handle account based systems and smart contracts). This lecture studies both these mechanisms in detail.

Introduction

In the last three lectures, we have considered proposals to scale throughput, latency, storage, communication and computation. In each of these proposals, the longest chain consensus protocol was modified. In a practical blockchain already operating in the real world, it is quite onerous to change the consensus layer: such a change would require a "meta consensus" among the participating nodes, i.e., consensus on how to change the consensus mechanism! Even a successful switch in the consensus mechanism would still lead to a "hard fork" in the ledger where the two paths would be following different consensus protocols. In this context, proposals to scale performance without changing the consensus layer are very appealing. Such are the goals of this lecture, where we discuss the two most promising "layer 2" proposals that scale performance by offloading computation and storage without impacting the core technology of blockchains ("layer 1") and overall security. The first mechanism is via *side chains* and allows a general account based model and smart contract. The second mechanism is via *payment channels* and is more restricted, e.g., to the UTXO model of handling payments.

Side Blockchains

A side blockchain is a smaller blockchain (in terms of trust represented for example by the number of nodes or hash power). The side blockchain is connected to a trusted blockchain by having its nodes propose blocks by committing the hashes of the blocks periodically to the trusted blockchain (Fig.1a). The ordering of blocks in the side blockchain is determined by the order of the hashes in the trusted blockchain; this way the security of the side blockchain is directly derived from that of the trusted blockchain. This mechanism is simple, practical, and efficient – a single trusted blockchain can cheaply support a large number of side blockchains, because it does not need to store, process, or validate the semantics of the blocks of the side blockchains. Rather, it only orders and records

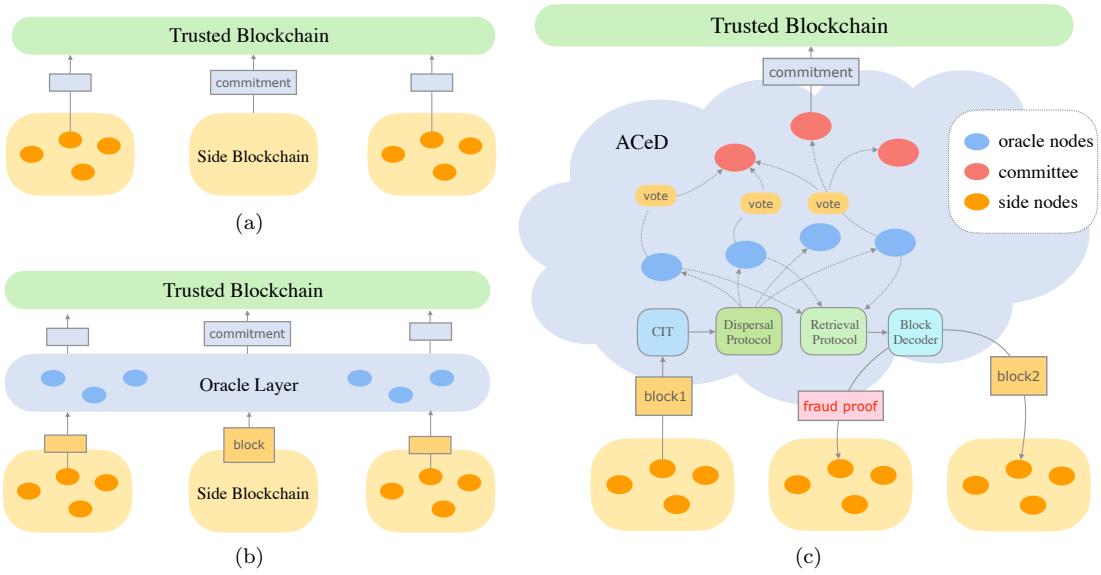

Figure 1: (a) Side blockchains commit the hashes of blocks to a larger trusted blockchain. (b) An oracle layer is introduced to ensure data availability. (c) ACeD is a scalable data availability oracle.

the hashes of these blocks. These side blockchains can remain safe and live even if they do not have honest majority. In many ways, the overall architecture of such a system is similar to that of a uni-consensus based sharded blockchain, which we saw in the previous lecture.

Data Availability Attack Both the side-blockchain scheme and the sharded blockchain scheme are vulnerable to the *data availability attack*. In this attack, a malicious node in a side blockchain network (or a shard) commits the hash of a block to the trusted blockchain without transmitting the block data to the other nodes. Let us understand why this is a serious problem. When preparing a single ledger out of the blockchain, should users wait until they receive this missing block, or should they skip this block in the ledger they prepare? If they decide to wait, they may be kept waiting indefinitely, leading to a loss of liveness. If instead they move on and prepare the ledger by ignoring the block, the adversary may reveal the block at a later time, at which point it cannot be ignored. The block may contain transactions that are incompatible with the rest of the ledger, causing a safety violation.

The data-availability attack is a well-known attack in the blockchain community, first introduced in the context of *light clients* in blockchains. It was popularized in this note by Vitalik Buterin of Ethereum and an accompanying paper. Light nodes merely store the headers of blocks and verify the proof-of-work criterion. They rely on full nodes to validate the block, and to provide a *fraud proof* if the block is invalid. Now consider the following data-availability attack: an adversary publishes a block header but does not publish the block data. A full node would simply ignore such a block until it receives the complete block data (which may never come). Thus, for all purposes, such a block would be an invalid block for a full node. How can a full node convince a light node to also ignore the block? A fraud proof would not be useful here. Rather, the light nodes and full nodes can construct a *data-availability proof* for themselves, a technique described in this paper. Such a proof can convince a light node whether or not a block of data is available.

In the context of light nodes, the data-availability attack is not fatal. Typically, miners also run full nodes. If a certain block is unavailable, they simply ignore it and mine in parallel to it (i.e., mine on its parent block). As long as the block remains unavailable, no honest miner will build on it. Eventually, it will fall out of the longest chain. At this stage, light nodes will automatically ignore

Table 1: Data availability attack in two scenarios.

Bitcoin light nodes	Sidechain clients
Light nodes random sample chunks	Oracle nodes store dispersed data
Rely on one honest full node to reconstruct the block	Any client can reconstruct the block
Probabilistic secure: need enough light nodes to ensure reconstruction	Deterministic secure: specific protocol to guarantee reconstruction

such a block. In contrast, the attack is more serious in the context of side blockchains and sharding. The crucial difference is that a side blockchain (or a shard) may not have an honest majority of miners/full nodes. Thus, there is no consistent, systematic way for a side blockchain's (or a shard's) participants to decide whether or not to include a missing block in their ledger. A *data availability oracle* is a tool by which nodes in such a system can be ascertain whether or not a block is available. Such an oracle must defend against various kinds of data-availability attacks. The properties that such an oracle must satisfy are given below. Many early techniques designed to protect against the data-availability attack, such as [4, 1] do not provide as strong a security guarantee as an oracle. This is primarily because they were designed to protect light nodes, where strong security properties were not essential; as described, even if the data-availability attack was successful, it would not be fatal. A comparison of the data-availability attack and its solutions for light nodes versus side blockchains is given in Table 1.

Data Availability Oracle A *data availability oracle* is an intermediate layer that accepts blocks from side blockchains, pushes verifiable commitments to the trusted blockchain and ensures data availability to the side blockchains. The oracle layer nodes work together to reach a consensus about whether the proposed block is retrievable (i.e., data is available) and only then commit it to the trusted blockchain. (Fig.1b). There are some straightforward constructions of such an oracle.

- *Repetition*: each oracle node keeps a full copy of the block.
- *Dispersal*: divide the block into several chunks and evenly disperse the chunks to oracle nodes. each node keeps a part of the block.

Repetition can simply conduct a voting to decide on the majority result, however the communication and storage overhead is proportional to the number of oracle nodes. If the number of oracle nodes is small (and yet deemed trustworthy), this is a very feasible approach. In a true decentralized sense this approach is not scalable. *Dispersal* minimizes the redundancy of data, but is not secure since even one malicious oracle node can violate the retrievability. So a trade-off between security and scalability is spotted here and the key challenge is how to securely and efficiently share the data amongst the oracle nodes to verify data availability.

Erasure Coding To design a scalable oracle, we introduce the use of erasure coding. The dispersal protocol can not tolerate even one malicious node hiding one data chunk of the block, but by adding redundancy to the block through appropriate erasure codes[2], any $1-\alpha$ fraction of the data chunks are enough to reconstruct the entire block, where α is undecodable ratio determined by a pair of erasure code and decoding algorithm, which is the least fraction of data adversary needs to hide to make the block undecodable.

For example, an (n, k) Reed-Solomon (1D-RS) code[3] evenly partitions a block B of b bytes into k data symbols of b/k bytes each as $B = [m_1, \cdots, m_k]$, and linearly combines them to generate a coded block with n coded symbols, $C = [c_1, \ldots, c_n]$. If the undecodable ratio $\alpha = 1/3$, and there are n oracle nodes and each of them is assigned with one coded symbol to store, then any subset of $2/3$ coded symbols in C is enough to reconstruct the entire coded block.

Table 2: Performance metrics for different data availability oracles (N: number of oracle nodes, b: block size).

	maximal adversary fraction	normal case		worst case		communication complexity
		storage overhead	download overhead	storage overhead	download overhead	
uncoded (repetition)	1/2	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(Nb)$
uncoded (dispersal)	1/N	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(b)$
1D-RS	1/2	$O(1)$	$O(1)$	$O(b)$	$O(b)$	$O(b)$
2D-RS [1]	1/2	$O(1)$	$O(1)$	$O(\sqrt{b}\log b)$	$O(\sqrt{b}\log b)$	$O(b)$
ACeD	1/2	$O(1)$	$O(1)$	$O(\log b)$	$O(\log b)$	$O(b)$

Coding integrity and correctness Intuitively, one can use a Merkle tree to provide proof of inclusion for any coded symbol. But a malicious block producer can construct a Merkle tree of a bunch of nonsense symbols so that no one can successfully reconstruct the block; this is the so-called *incorrect-coding* attack. An approach to detect such attacks is via an *incorrect-coding proof* (also called "fraud proof"), which contains symbols that fail the parity check and can be provided by any node who tries to reconstruct the data. The size of the proof determines the efficiency of the fraud proof method. We find that the fraud proof of 1D-RS contains k coded symbols, essentially not much better than downloading the original block (n symbols). To reduce the size of the fraud proof, 2D-RS [1] places a block into a $(\sqrt{k}, \sqrt{k})$ matrix and apply $(\sqrt{n}, \sqrt{k})$ Reed-Solomon code on all columns and rows to generate n^2 coded symbols; 2D-RS reduces the fraud proof size to $O(\sqrt{b}\log b)$ if we assume symbol size is constant. Further, a cryptographic hash accumulator called Coded Merkle Tree (CMT) [4] is proposed and reduces the proof size to $O(\log b)$. However, note that CMT is designed for light nodes to randomly sample coded symbols for data availability check, it provides probabilistic security and thus can not solve the oracle problem for side blockchains.

ACeD

Authenticated Coded Dispersal (ACeD) is a recent work that proposes a scalable solution to the data availability oracle problem. The performance comparison between ACeD and other solutions is in Table 2. There are four core components in ACeD, as depicted in Figure 1c.

- A coded commitment generator called *Coded Interleaving Tree* (CIT), which is constructed layer by layer in an interleaved manner embedded with erasure codes. The interleaving property avoids downloading extra proof and thus minimizes the number of symbols needed to store.
- A pair of dispersal and retrieval protocol are designed to disperse tree chunks among the network with the least redundancy and ensure the retrievability of all data.
- A hash-aware peeling decoder is used to achieve linear decoding complexity. The fraud proof is minimized to a single parity equation.

Coded Interleaving Tree CIT is as efficient as CMT on the fraud proof size due to the similar tree structure, but with an entirely different set of construction rules. Specifically, CMT uses a *pull model* to randomly sample symbols via an anonymous network, thus is applicable only to light nodes. In contrast, CIT is designed to support a secure deterministic dispersal with a *push model*, there is no anonymity assumption, and is designed to be incentive compatible.

The construction rules of CIT are best seen in stages. CIT takes a block proposed by a client as an input and creates three outputs: a commitment, a sequence of coded symbols, and their proof of membership POM. The commitment is the root of CIT, the coded symbols are the leaves of CIT

Table 3: System Performance Metrics

Metric	Formula	Explanation
Maximal adversary fraction	β	The maximum number of adversaries is βN.
Storage overhead	$D_{\text{store}}/D_{\text{info}}$	The ratio of total storage used and total information stored.
Download overhead	$D_{\text{download}}/D_{\text{data}}$	The ratio of the size of downloaded data and the size of reconstructed data.
Communication complexity	D_{msg}	Total number of bits communicated.

in the base layer, and POM for a symbol includes the Merkle proof (all siblings' hashes from each layer) and a set of parity symbols from all intermediate layers.

The construction process of an example CIT is illustrated in figure 2. Suppose a block has size b bytes, and its CIT has ℓ layers. The first step to construct the CIT is to divide the block evenly into small chunks, each is called a data symbol. The size of a data symbol is denoted as c, so there are $s_\ell = b/c$ data symbols. And we apply erasure codes with coding ratio $r \leq 1$ to generate $m_\ell = s_\ell/r$ coded symbols in the base layer. Then by aggregating the hashes of every q coded symbols we get m_ℓ/q data symbols for its parent layer (layer $\ell-1$), which can be further encoded to $m_{\ell-1} = m_\ell/(qr)$ coded symbols. We aggregate and code the symbols iteratively until the number of symbols in a layer decays to t, which is the size of the root.

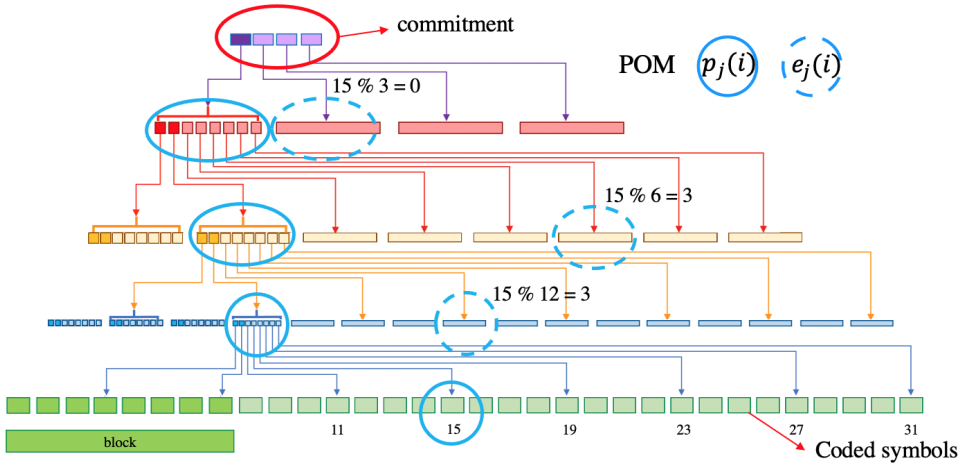

Figure 2: (1) CIT construction process of a block with $s_\ell = 8$ data symbols, applied with erasure codes of coding ratio $r = \frac{1}{4}$. The batch size $q = 8$ and the number of hashes in root is $t = 4$. (2) Circled symbols constitute the 15th base layer coded symbol and its POM. The solidly circled symbols are the base layer coded symbol and its Merkle proof (intermediate data symbols), the symbols circled in dash are parity symbols sampled deterministically.

For all layers j except for root, $1 \leq j \leq \ell$, denote the set of all m_j coded symbols as M_j, which contains two disjoint subsets of symbols: data symbols S_j and parity symbols P_j. The number of data symbols is $s_j = rm_j$. Specifically, we set $S_j = [0, rm_j)$ and $P_j = [rm_j, m_j)$. Given a block of s_ℓ data symbols in the base layer, the aggregation rule for the k-th data symbol in layer $j-1$ is defined as follows:

$$Q_{j-1}[k] = \{\mathsf{H}(M_j[x]) \mid x \in [0, M_j), k = x \bmod rm_{j-1}\} \tag{1}$$

$$M_{j-1}[k] = \mathsf{H}(\mathsf{concat}(Q_{j-1}[k])) \tag{2}$$

where $1 \leq j \leq \ell$ and H is a hash function. $Q[k]$ is the tuple of hashes that will be used to generate k-th symbol in the parent layer and concat represents the string concatenation function which will concatenate all elements in an input tuple.

Generating a POM for a base layer symbol can be considered as a layer by layer sampling process as captured by the following functions:

$$f_\ell : [m_\ell] \to \binom{m_{\ell-1}}{2}, \cdots, f_2 : [m_\ell] \to \binom{m_1}{2};$$

Each function maps a base layer symbol to two symbols of the specified layer: one is a systematic symbol and the other is a parity symbol. We denote the two symbols with a tuple of functions $f_j(i) = (p_j(i), e_j(i))$, where $p_j(i)$ is the sampled systematic symbol and $e_j(i)$ is the sampled parity symbol, each is defined as follows:

$$p_j(i) = i \bmod rm_{j-1}; \quad e_j(i) = rm_{j-1} + (i \bmod (1-r)m_{j-1}) \tag{3}$$

where $p_j : [m_\ell] \to [0, rm_{j-1})$ and $e_j : [m_\ell] \to [rm_{j-1}, m_{j-1})$. Note that the sampled non-base layer systematic symbols are automatically the Merkle proof for both systematic and parity symbols in the lower layer.

There are two important properties of CIT. (1) It guarantees that if at least $\eta \leq 1$ ratio of distinct base layer symbols along with their POM are sampled, then in every intermediate layer, at least η ratio of distinct symbols are already picked out by the collected POM. It ensures the reconstruction of each layer of CIT. (2) All sampled symbols at each layer have a common parent, which ensures the space efficiency of sampling.

Payment Channels

Payment channels support users to make payments for multiple times off chain while only submitting transactions on-chain to start the channel and handle a dispute. It locks parts of the state of the blockchain when starting a channel, processes transactions associated with this locked state in an application layer, and finally unlocks the channel with the updated state.

In a UTXO system like Bitcoin, a transaction contains a sequence of inputs and outputs. Outputs serve as *locks* for a specific amount of funds, while inputs serve as keys of corresponding outputs to unlock the funds and transfer them between accounts. There are three new types of cryptographic primitives that allow "flexible locks", so that the trust can be extracted outside the blockchain. In particular there are three important cryptographic primitives we explore here.

- Multisig. the locking transaction needs to be signed by k out of n public keys
- Hashlock. unlocking requires the owner's public key and a secret
- Timelock. requirement is related to block height and other time-based conditions

According to the payment direction, payment channels can be classified into three categories, one-way payment channels, two-way payment channels and payment networks. Here we discuss the first two types.

One-way payment channels

One-way payment channels only allow funds to flow in one direction, from payer to recipient. For example, suppose Alice wants to pay Bob in increments of 0.1 BTC up to a maximum of 1 BTC. Each of the payments can be made "on-chain", however the payment channel proceeds in the following steps (see Figure 3).

1. **Creating the channel.** Alice signs a *funding transaction* and posts it on the blockchain to create the channel. The funding transaction contains a single input with 1 BTC that is signed by Alice, the output can either:

   - contain an address that is derived from both Alice and Bob's public keys ($\mathsf{multiaddr} = \mathsf{GetAddrbyAccount}(\mathsf{pk}_1, \mathsf{pk}_2)$). And it can only be unlocked with both Alice and Bob's secret keys, neither one can unlock it alone.
   - contain Alice's address, and a timelock that specifies the expiration time of the channel. If Bob is online, he can receive the payment through posting a closing transaction, otherwise the coins return to Alice after the time expires.

2. **Updating the payments.** After creating the channel, Alice can make payments to Bob and each payment creates an intermediate off-chain transaction, which is called *commitment transaction*. A commitment transaction uses the output of funding transaction as input, and specifies how the funds held by Alice are sent to Bob through the output, e.g. 0.9 BTC to Alice, 0.1 BTC to Bob. Alice can continue paying Bob through the payment channel in this fashion. Each time she wants to pay another 0.1 BTC to Bob, she constructs a new commitment transaction from the same funding transaction output and sends it to Bob. Those commitment transactions are held by Bob, who can later decide to post any one (and only one) of them to the blockchain to receive the funds.

3. **Closing the channel.** To close the channel, a *closing transaction* is generated and posted on blockchain to close the channel and update the state. There are two possible ways to close the channel corresponding to the funding transaction,

   - Cooperative: Bob posts a commitment transaction, whose input contains a signature signed with both Alice and Bob's secret keys (2-of-2 $\mathsf{multisig}$). The signature will be verified on-chain and once it matches the output address of a funding transaction, Alice successfully makes the payments.
   - Non-cooperative: Bob is offline and channel expires (e.g. one-day $\mathsf{timelock}$), Alice posts the closing transaction on chain whose input contains Alice's signature and an expired timelock, Alice recovers her money.

Two-way payment channels

Two-way payment channels enable two parties to transfer funds to each other. Suppose now Alice and Bob both have 0.5 BTC to make payments. Consider the following steps (see Figure 4).

1. **Creating the channel.** The funding transaction now contains two inputs, each with 0.5 BTC, the first is signed by Alice and the second is signed by Bob. The output contains 1 BTC signed by both Alice and Bob using 2-of-2 $\mathsf{multisig}$. Before starting making payments, Alice and Bob will each create a secret and exchange the hashes of the secrets with each other. The opening transaction will not be signed and posted on-chain until Alice and Bob receive a special commitment transaction respectively. The commitment transaction here will contain an input with 1 BTC signed by both Alice and Bob (the output of a funding transaction), and two outputs. For the transaction that is held by Alice, the outputs are (1) 0.5 BTC to Bob, and (2) 0.5 BTC with timelock to Alice or to Bob if he knows Alice's secret, these outputs are signed by Bob. Similar for Bob, the outputs are signed by Alice and contain (1) 0.5 BTC to Alice, and (2) 0.5 BTC with one-week timelock to Bob or to Alice if she knows Bob's secret.

2. **Updating the payments.** Alice and Bob start generating a normal commitment transaction. Suppose Alice wants to send Bob 0.1 BTC, then she will sign a transaction with outputs: (1)

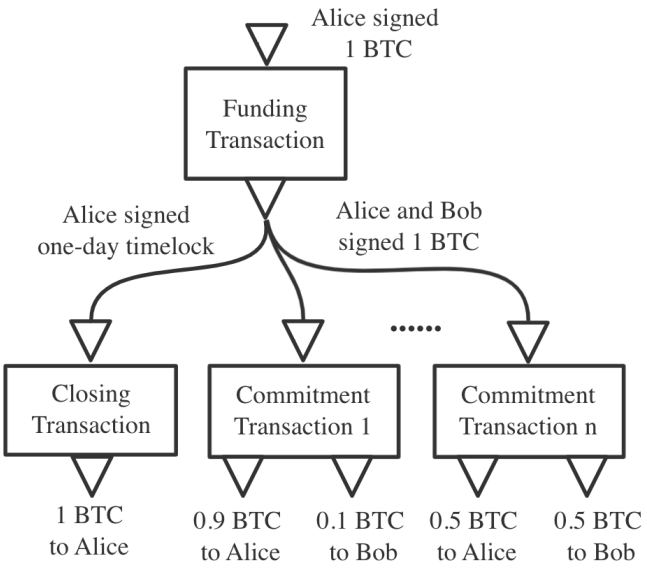

Figure 3: A one-way payment channel example.

0.4 BTC to Alice, and (2) 0.6 BTC with one-week timelock to Bob or to Alice if she knows Bob's secret. For each payment, Alice and Bob will generate a new secret and exchange the older secret to ensure that the older transaction will not be posted on chain.

3. **Closing the channel.** When one of the parties is offline, the channel can be closed by revealing any one of the commitment transactions in a non-cooperative way. And in the cooperative situation, Alice and Bob can create a transaction sending the settled balance to each party.

References

[1] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities. *arXiv preprint arXiv:1809.09044*, 2018.

[2] Shu Lin and Daniel J Costello. *Error control coding*, volume 2. Prentice hall, 2001.

[3] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[4] Mingchao Yu, Saeid Sahraei, Songze Li, Salman Avestimehr, Sreeram Kannan, and Pramod Viswanath. Coded merkle tree: Solving data availability attacks in blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 114–134. Springer, 2020.

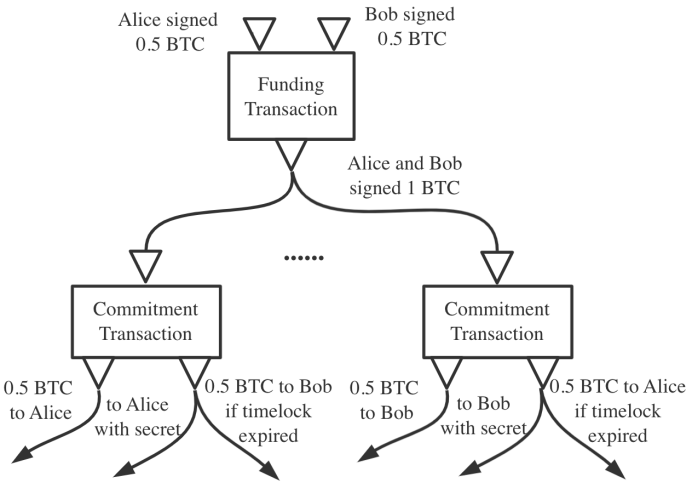

Figure 4: A two-way payment channel example.