Snappy: Fast On-chain Payments with Practical Collaterals

Vasilios Mavroudis
University College London
v.mavroudis@ucl.ac.uk

Karl Wüst
ETH Zurich
karl.wuest@inf.ethz.ch

Aritra Dhar
ETH Zurich
aritra.dhar@inf.ethz.ch

Kari Kostiainen
ETH Zurich
kari.kostiainen@inf.ethz.ch

Srdjan Capkun
ETH Zurich
srdjan.capkun@inf.ethz.ch

***Abstract*—Permissionless blockchains offer many advantages but also have significant limitations including high latency. This prevents their use in important scenarios such as retail payments, where merchants should approve payments fast. Prior works have attempted to mitigate this problem by moving transactions off the chain. However, such Layer-2 solutions have their own problems: payment channels require a separate deposit towards each merchant and thus significant locked-in funds from customers; payment hubs require very large operator deposits that depend on the number of customers; and side-chains require trusted validators.**

In this paper, we propose Snappy, a novel solution that enables recipients, like merchants, to safely accept fast payments. In Snappy, all payments are on the chain, while small customer collaterals and moderate merchant collaterals act as payment guarantees. Besides receiving payments, merchants also act as statekeepers who collectively track and approve incoming payments using majority voting. In case of a double-spending attack, the victim merchant can recover lost funds either from the collateral of the malicious customer or a colluding statekeeper (merchant). Snappy overcomes the main problems of previous solutions: a single customer collateral can be used to shop with many merchants; merchant collaterals are independent of the number of customers; and validators do not have to be trusted. Our Ethereum prototype shows that safe, fast (<2 seconds) and cheap payments are possible on existing blockchains.

I. Introduction

Cryptocurrencies based on permissionless blockchains have shown great potential in decentralizing the global financial system, reducing trust assumptions, increasing operational transparency and improving user privacy. However, this technology still has significant limitations, preventing it from posing as valid alternative to established transaction means such as card payments.

One of the main problems of permissionless blockchains is *high latency*. For example, in Ethereum [1], users have to wait approximately 3 minutes (10 blocks) before a new payment can be safely accepted [2], [3]. In comparison, traditional and centralized payment systems such as VISA can confirm payments within 2 seconds [4]–[6]. High latency makes permissionless blockchains unsuitable for many important applications such as point-of-sale purchases and retail payments.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24049
www.ndss-symposium.org

To improve blockchain performance, various consensus schemes have been proposed [7]. While techniques like sharding and Proof-of-Stake can increase the *throughput* of blockchains significantly, currently there are no promising solutions that would drastically decrease the latency of permissionless blockchains.

Thus, researchers have explored alternative avenues to enable fast payments over slow blockchains. Arguably, the most prominent approach are Layer-2 solutions that move transaction processing off the chain and use the blockchain only for dispute resolution and occasional synchronization. However, such solutions have their own shortcomings. For example, payment channels require a separate deposit for each channel, resulting in large locked-in funds for users such as retail customers [8], [9]. Payment networks cannot guarantee available paths and are incompatible with the unilateral nature of retail payments from customers to merchants [10], [11]. Payment hubs [12], [13] require either a trusted operator or huge collaterals that are equal to the total expenditure of all customers [14], [15]. Side-chains based on permissioned BFT consensus require 2/3 trusted validators and have high communication complexity [16], [17].

Our solution. In this paper, we introduce Snappy, a system that enables safe and fast (zero-confirmation) on-chain payments. Snappy can be used today on top of low-throughput and high-latency blockchains such as Ethereum and in the future on top of (sharded) high-throughput and mid-latency blockchains.

We tailor our solution for application scenarios such as retail payments, but emphasize that our design can be used in any scenario where a large number of users (e.g., 100,000 customers) make payments towards a moderate set of recipients (e.g., 100 merchants). In Snappy, the merchants form a joint consortium that may consist of large retailers with several stores globally or small local stores from the same neighborhood. The merchants need to communicate to be able accept fast-payments safely, but importantly neither the merchants nor the customers have to trust each other.

Snappy relies on customer collaterals that enable merchants to safely accept payments before the transaction has reached finality in the blockchain. The collaterals serve as payment guarantees and are deposited by customers to a smart contract during system enrollment. If the transaction corresponding to an accepted payment does not appear in the blockchain within a reasonable time (double-spending attack), the victim merchant can recoup the lost funds from the malicious customer's collaterals. The customer's deposit should cover the value

of the customer's purchases within the latency period of the blockchain (e.g., 3 minutes in Ethereum) which makes them small in practice (e.g., $100 would suffice for many users). Moreover, customers do not need to repeatedly replenish their collaterals, as they are used only in the case of attack.

In Snappy, the payment recipients (merchants) act as *untrusted statekeepers* whose task is to track and collectively approve incoming transactions. To initiate a fast payment, a customer creates a transaction that transfers funds to a smart contract and indicates the merchant as beneficiary. The recipient merchant sends the transaction to the statekeepers and proceeds with the sale only if a *majority* of them approves it with their signatures. Upon collecting the required signatures, the merchant broadcasts the transaction in the blockchain network to be processed by an Arbiter smart contract. Once the transaction is processed and logged by the Arbiter, the payment value is forwarded to the merchant.

Statekeepers must also deposit collaterals that protect merchants from attacks where a malicious statekeeper colludes with a customer. In such a case, the victim merchant can use the statekeeper's approval signatures as evidence to claim any lost funds from the misbehaving statekeeper's collateral. The size of statekeeper collateral is proportional to the total amount of purchases that all participating merchants expect within the blockchain latency period. Crucially, the statekeeper collaterals are independent of the number of customers which allows the system to scale. The main security benefit of statekeeper collaterals is that they enable fast and safe payments without trusted parties.

Main results. We prove that a merchant who follows the Snappy protocol and accepts a fast payment once it has been approved by the majority of the statekeepers never loses funds regardless of any combination of customer and statekeeper collusion. We also demonstrate that Snappy is practical to deploy on top of existing blockchains by implementing it on Ethereum. The performance of our solution depends primarily on number of participating statekeepers (merchants). For example, assuming a deployment with 100 statekeepers, a payment can be approved in less than 200 ms with a processing cost of $0.16 (169k Ethereum gas), which compares favorably to card payment fees.

Snappy overcomes the main problems of Layer-2 solutions in application scenarios such as retail payments. In contrast to BFT side-chains that assume that $2/3$ honest validators and require multiple rounds of communication, Snappy requires no trusted validators and needs only one round of communication. Unlike payment channels, Snappy enables payments towards many merchants with a single and small customer deposit. In contrast to payment networks, Snappy payments can always reach the merchants, because there is no route depletion. And finally, the statekeeping collaterals are practical even for larger deployments, compared to those in payment hubs, as they are independent of the number of customers in the system.

Contributions. This paper makes the following contributions:

- ❖ *Novel solution for fast payments.* We propose a system called Snappy that enables fast and secure payments on slow blockchains without trusted parties using moderately-sized and reusable collaterals that are practical for both customers and merchants.
- ❖ *Security proof.* We prove that merchants are guaranteed to receive the full value of all accepted payments, in any possible combination of double spending by malicious customers and equivocation by colluding statekeepers.
- ❖ *Evaluation.* We implemented Snappy on Ethereum and show that payment processing is fast and cheap in practice.

This paper is organized as follows: Section II explains the problem of fast payments, Section III provides an overview of our solution and Section IV describes it in detail. Section V provides security analysis and Section VI further evaluation. Section VII is discussion, Section VIII describes related work, and Section IX concludes the paper.

II. Problem Statement

In this section we motivate our work, explain our assumptions, discuss the limitations of previous solutions, and specify requirements for our system.

A. Motivation

The currently popular permissionless blockchains (e.g., Bitcoin and Ethereum) rely on Proof-of-Work (PoW) consensus that has well-known limitations, including low throughput (7 transactions per second in Bitcoin), high latency (3 minutes in Ethereum), and excessive energy consumption (comparable to a small country [18]). Consequently, the research community has actively explored alternative permissionless consensus schemes. From many proposed schemes, two prominent approaches, *Proof of Stake* and *sharding*, stand out [7].

Proof of Stake (PoS) systems aim to minimize the energy waste by replacing the computationally-heavy puzzles of PoW with random leader election such that the leader selection probability is proportional to owned staked. While the current PoS proposals face various security and performance issues [7], the concept has shown promise in mitigating the energy-consumption problem.

Sharding systems increase the blockchain's throughput by dividing the consensus participants into committees (*shards*) that process distinct sets of transactions. Recent results reported significant throughput increases in the order of thousands of transactions per second [19], [20]. Despite several remaining challenges, sharding shows great promise in improving blockchain throughput.

Sharding and PoS can also address transaction latency. Recent works such as Omniledger [19] and RapidChain [20] use both techniques and report latencies from 9 to 63 seconds, assuming common trust models like honest majority or 1/3 Byzantine nodes. However, such measurements are achieved in fast test networks (permissionless blockchains rely on slow peer-to-peer networks) and under favorable work loads (e.g., largely pre-sharded transactions).

Our conclusion is that while permissionless blockchain throughput and energy efficiency are expected to improve in the near future, latency will most likely remain too high for various scenarios such as point-of-sale payments and retail

shopping, where payment confirmation is needed within 1-2 seconds. Therefore, *in this paper we focus on improving blockchain payment latency*. We consider related problems like limited blockchain throughput as orthogonal problems with known solutions. Appendix A provides further background on permissionless consensus.

B. System Model and Assumptions

Customers and merchants. We focus on a setting where n users send payments to k recipients such that n is large and k is moderate. One example scenario is a set of $k = 100$ small shops where $n = 100,000$ customers purchase goods at. Another example is $k = 100$ larger retail stores with $n = 1$ million customers [21], [22].

We consider merchants who accept *no risk*, i.e., they hand the purchased products or services to the customers, only if they are guaranteed to receive the full value of their sale. Therefore, naive solutions such as accepting zero-confirmation transaction are not applicable [23]. The customers are assumed to register once to the system (similar to a credit card issuance processes) and then visit shops multiple times.

We assume that merchants have permanent and fairly reliable Internet connections. Customers are not expected to be online constantly or periodically (initial online registration is sufficient). At the time of shopping, customers and merchants can communicate over the Internet or using a local channel, such as a smartphone NFC or smart card APDU interface.

Blockchain. We assume a permissionless blockchain that has sufficient throughput, but high latency (see motivation). The blockchain supports smart contracts. We use Ethereum as a reference platform throughout this work, but emphasize that our solution is compatible with most permissionless blockchains with smart contracts [24]–[26]. To ensure compatibility with existing systems like Ethereum, we assume that smart contracts have access to the current state of the blockchain, but not to all the past transactions.

Adversary. The main goal of this paper is to enable secure and fast payments. Regarding payment security, we consider a strong adversary who controls an arbitrary number of customers and all other merchants besides the target merchant who accepts a fast payment. The adversary also controls the network connections between customers and merchants but cannot launch network-level attacks such as node eclipsing [27]. The adversary cannot violate the consensus guarantees of the blockchain, prevent smart contract execution, or violate contract integrity. For payment liveness, we additionally require that sufficiently many merchants are responsive (see Section V-B).

C. Limitations of Known Solutions

A prominent approach to enable fast payments on slow permissionless blockchains is so called Layer-2 solutions that move transaction processing off the chain and use the blockchain only in case of dispute resolution and occasional synchronization between the on-chain and off-chain states. Here, we outline the main limitations of such solutions. Section VIII provides more details on Layer-2 solutions and their limitations.

Payment channels transfer funds between two parties. The security of such solutions is guaranteed by separate deposits that must cover periodic expenditure in each individual channel. In our "small shops" example with $k = 100$ merchants and an average customer expenditure of $e = \$10$, the customers would need to deposit combined \$1,000. In our "large retailers" example with $k = 100$ merchants and expenditure of $e = \$250$, the total deposit is \$25,000. Payment channels require periodic deposit replenishment.

Payment networks address the above problem of having to set up many separate channels by using existing payment channels to find paths and route funds. Payments are possible only when the necessary links from the source (customer) to the destination (merchant) exist. However, payment networks are unreliable, as guaranteeing the suitable links between all parties is proven difficult [28]. Moreover, retail payments are pre-dominantly one-way from customers to merchants, and thus those links get frequently depleted, reducing the route availability even further [29].

Payment hubs attempt to solve the route-availability problem by having all customers establish a payment channel to a central hub that is linked to all merchants. The main problem of this approach is that the hub operator either has to be trusted or it needs to place a very large deposit to guarantee all payments. Since the required collateral is proportional to the number of customers, in our large retailers example, a hub operator will have to deposit \$250M to support $n = 1$M customers with an expenditure of $e = \$250$. To cover the cost of locking in such a large mount of funds, the operator is likely to charge substantial payment fees.

Commit-chains aim to improve payment hubs by reducing or eliminating operator collaterals. To do that, they rely on periodic on-chain *checkpoints* that finalize multiple off-chain transactions at once. While this improves throughput, it does not reduce latency, as users still have to wait for the checkpoint to reach finality on-chain [14], [30]. Other commit-chain variants enable instant payments, but require equally large collaterals as payment hubs. Additionally, in such variants users need to monitor the checkpoints (hourly or daily) to ensure that their balance is represented accurately [14], [31]. We focus on retail setting where customers do not have to be constantly or periodically online (recall Section II-B), and thus cannot be expected to perform such monitoring.

Side-chains rely on a small number of collectively trusted validators to track and agree on pending transactions. Typically, consensus is achieved using Byzantine-fault tolerant protocols [32] that scale up to a few tens of validators and require 2/3 of honest validators. Thus, side-chains contradict one of the main benefits of permissionless blockchains, the fact that no trusted entities are required. Additionally, BFT consensus requires several communication rounds and has high message complexity.

D. Requirements

Given these limitations of previous solution, we define the following requirements for our work.

- ❖ *R1: Fast payments without trusted validators.* Our solution should enable payment recipients such as merchants to accept fast payments assuming no trusted validators.
- ❖ *R2: Practical collaterals for large deployments.* Collaterals should be pratical, even when the system scales for large number of customers or many merchants. In particular, the customer collaterals should only depend on their own spending, not the number of merchants. The entities that operate the system (in our solution, the merchants) should deposit an amount that is proportional to their sales, not the number of customers.
- ❖ *R3: Cheap payment processing.* When deployed on top of an existing blockchain system such as Ethereum, payment processing should be inexpensive.

III. Snappy Overview

In our solution, Snappy, customer collaterals are held by a smart contract called *Arbiter* and enable merchants to safely accept payments before a transaction reaches finality on the chain. If a pending transaction does not get confirmed on the chain (double spending attack), the victim merchant can recover the lost funds from the customer's collateral.

For any such solution, it is crucial that the total value of a customer's pending transactions never exceeds the value of its collateral. This invariant is easy to ensure in a single-merchant setup by keeping track of the customers' pending transactions. However, in a retail setting with many merchants, each individual merchant does not have a complete view of each customer's pending payments, as each customer can perform purchases with several different merchants simultaneously.

A natural approach to address this problem is to assume that the merchants *collaborate* and collectively track pending transactions from all customers (see Figure 2 (right)). Below, we outline simple approaches to realize such collaboration and point out their drawbacks that motivate our solution.

A. Strawman Designs

Transaction broadcasting. Perhaps the simplest approach would be to require that all merchants broadcast all incoming payments, so that everyone can keep track of the pending transactions from each customer. Such a solution would prevent customers from exceeding their collateral's value, but assumes that all the merchants are *honest*. A malicious merchant, colluding with a customer, could mislead others by simply not reporting some of the customer's pending payments. The customer can then double spend on all of its pending transactions, thus enabling the colluding merchant to deplete its collateral and prevent other merchants from recovering their losses. The same problem would arise also in cases where a benign merchant fails to communicate with some of the merchants (e.g., due to a temporary network issue) or if the adversary drops packets sent between merchants.

We argue that such transaction broadcasting might be applicable in specific settings (e.g., large retailers that trust each other and are connected via high-availability links), but it fails to protect mutually-distrusting merchants such as small shops or restaurants, and is unsuited to scenarios where mutually-trusting merchants cannot establish expensive links.

Unanimous approval. Alternatively, merchants could send each incoming transaction to all other merchants and wait until each of them responds with a signed approval. While some of the merchants may act maliciously and equivocate, rational merchants holding pending transactions from the same customer will not, as this would put their own payments at risk. The Arbiter contract would refuse to process any claims for losses, unless the merchant can provide approval statements from all other merchants. Such unanimous approval prevents the previous attack (assuming rational actors), but it suffers from poor liveness. Even if just one of the merchants is unreachable, the system cannot process payments.

BFT consensus. A common way to address such availability concerns is to use Byzantine-fault tolerant protocols. For example, the merchants could use a BFT consensus such as [32] to stay up to date with all the pending payments. Such a solution can tolerate up to $1/3$ faulty (e.g., non-responsive) merchants and thus provides increased availability compared to the previous design. However, this solution has all the limitations of side-chains (recall Section II-C). In particular, BFT consensus requires $2/3$ trusted validators and several rounds of communication. Therefore, BFT side-chains are not ideal even in the case of mutually-trusting merchants.

B. Snappy Main Concepts

To overcome the problems of the above strawman designs, we use two primary techniques: (a) *majority approval* and (b) *merchant collaterals*, such that fast payments can be accepted safely even if all the merchants are untrusted and some of them are unreachable.

The use of majority approval generates non-deniable evidence about potentially misbehaving merchants. Together with merchant collaterals, this enables attack victims to recover their losses, in case a merchant colludes with a malicious customer. In an example attack, a malicious customer sends transaction τ to a victim merchant and τ' to a colluding merchant, and simultaneously double spends both τ and τ'. The colluding merchant claims the lost value from the customer's collateral which exhausts it and prevents the victim merchant from recovering losses from the customer's collateral. However, the victim can use the colluding merchant's signature from the majority approval process as evidence and reclaim the lost funds from the merchant's collateral. Given these main ideas, next we provide a brief overview of how Snappy works.

Collaterals. To register in the Snappy system, each customer deposits a collateral to Arbiter's account (see "Registration" in Figure 1 (left)). The value of the collateral is determined by the customer, and it should suffice to cover its expenditure e_t during the blockchain latency period (e.g., $e_t = \$100$ for 3 minutes in Ethereum).

Since merchants are untrusted, they also need to deposit a collateral. The size of the merchant collateral depends on the total value of sales that *all* participating merchants process within the latency period. For example, for a consortium of $k = 100$ small shops that process $p_t = 6$ payments of $e_t = \$5$ on average during the latency period, a collateral of \$3,000 will suffice. In a deployment with $k = 100$ larger retailers, where each one handles $p_t = 15$ purchases of $e_t = \$100$ (on average) within the latency period, each merchant will need

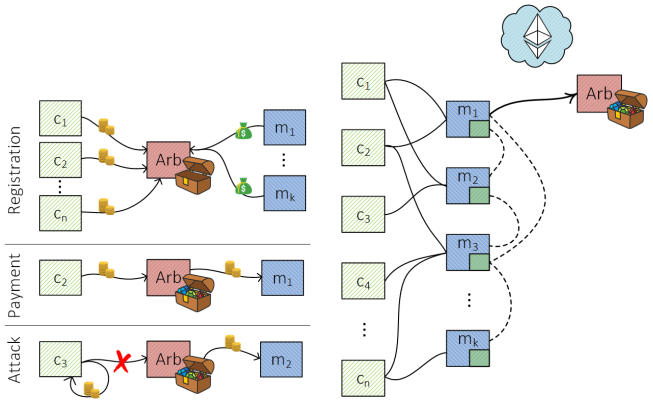

Fig. 1 (left): **Flow of funds.** Customers and merchants deposit collaterals to the arbiter smart contract. Payments flow from customers to merchants through the arbiter. In case of attack, the victim merchant can recover any losses from the arbiter.

Fig. 2 (right): **Example deployment** where each merchant operates one statekeeper. Customers make payments towards merchants, who consult a majority of the statekeepers before accepting them.

to deposit $150,000 [21], [22]. We acknowledge that this is a significant deposit, but feasible for large corporations.

Payment approval. After registering, a customer can initiate a payment by sending a payment intent to a merchant together with a list of its previous approved but not yet finalized transactions. The merchant can verify that the provided list is complete, by examining index values that are part of each Snappy transaction. If the total value of intended payment and the previously approved payments does not exceed the value of the customer's collateral, the merchant proceeds.

To protect itself against conflicting Snappy transactions sent to other merchants simultaneous, the merchant collects approval signatures from more than half of the participating merchants, as shown in Figure 2 (right). Such majority approval does not prevent malicious merchants from falsely accepting payments, but provides undeniable evidence of such misbehavior that the merchant can later use to recover losses in case of an attack. In essence, a merchant m signing a transaction τ attests that "m has not approved other transactions from the same customer that would conflict with τ". Thus, m needs to check each transaction against those it has signed in the past. The merchant who is about to accept a payment also verifies that the approving merchants have each sufficient collateral left in the system to cover the approved payment. Once these checks are complete, the merchant can safely accept the payment.

The customer construct a complete Snappy payment such that the payment funds are initially sent to Arbiter that logs the approval details into its state and after that forwards the payment value to the receiving merchant (see "Payment" in Figure 1 (left)). We route all payments through the Arbiter contract to enable the Arbiter to perform claim settlement in blockchain systems like Ethereum where smart contracts do not have perfect visibility to all past transactions (recall Section II-B). If future blockchain systems offer better transaction visibility to past transactions for contracts, payments can be also routed directly from the customers to the merchants.

Settlements. If the transaction does not appear in the blockchain after a reasonable delay (i.e., double-spending attack took place), the victim merchant can initiate a settlement process with the arbiter to recover the lost funds ("Attack in Figure 1 (left)). The Arbiter uses the logged approval evidence from its state to determine who is responsible for the attack, and it returns the lost funds to the victim merchant either from the collateral of the customer or the misbehaving merchant who issued false approval.

Separation of statekeeping. So far, we have assumed that payment approval requires signatures from merchants who track pending transactions. While we anticipate this to be the most common deployment option, for generality and separation of duties, we decouple the tracking and approval task of merchants into a separate role that we call *statekeeper*. For the rest of the paper, we assume a one-to-one mapping between merchants and statekeepers, as shown in Figure 2 (right) and in Appendix B we discuss alternative deployment options. When there are an identical number of merchants and statekeepers, the value of statekeepers' collaterals is determined as already explained for merchant collaterals.

Incentives. There are multiple reasons why merchants would want to act as statekeepers. One example reason is that it allows them to join a Snappy consortium, accept fast blockchain payments securely and save on card payment fees. To put the potential saving into perspective, considering our large retail stores example and the common 1.5% card payment fee (and a cost of $0.16 per Snappy payment). In such a case, a collateral of $150,000 is amortized in ∼37 days. Potential free-riding by a consortium member could be handled by maintaining a ratio of approvals made and received for each merchant and excluding merchants who fall below a threshold.

Another example reason is that the merchants could establish a fee-based system where each approver receives a small percentage of the transaction's value to cover the operational cost and the collateral investment needed to act as a statekeeper.

IV. Snappy Details

In this section, we describe the Snappy system in detail. We instantiate it for Ethereum, but emphasize that our solution is not limited to this particular blockchain. We start with background on aggregate signatures, followed by Snappy data structures, registration, payment and settlement protocols.

A. Background on BLS Signatures

To enable signature *aggregation* for reduced transaction size and efficient transaction verification, we utilize the Boneh-Lynn-Shacham (BLS) Signature Scheme [33], along with extensions from [34], [35]. BLS signatures are built on top of a bilinear group and a cryptographic hash function H :

Field	Symbol	Type	Description
To	τ_{to}	160-bit	Arbiters addr
From	τ_f	160-bit	Customer's addr
Value	τ_v	Integer	Transferred funds
ECDSA Sig.	v, r, s	256-bits	Tx signature triplet.
Data			
↪ *Operation*	τ_{op}	String	e.g., "Pay", "Claim"
↪ *Merchant*	τ_m	160-bit	Merchant's address
↪ *Payment Index*	τ_i	Integer	Monotonic counter
↪ *Signatures*	τ_A	512-bits	Aggregate signature
↪ *Quorum*	τ_q	256-bits	Approving parties

TABLE I: **Snappy transaction τ format.** All Snappy-specific information is encoded into the Data field of Ethereum transactions.

$\{0,1\}^* \mapsto \mathbb{G}$. A bilinear group $\mathsf{bp} = (p, \mathbb{G}, \mathbb{H}, \mathbb{G}_T, e, g, h)$ [1] consists of cyclic groups $\mathbb{G}$, $\mathbb{H}$, $\mathbb{G}_T$ of prime order p, generators g, h that generate $\mathbb{G}$ and $\mathbb{H}$ respectively, and a *bilinear map* $e : \mathbb{G} \times \mathbb{H} \to \mathbb{G}_T$ such that: The map is bilinear, i.e., for all $u \in \mathbb{G}$ and $v \in \mathbb{H}$ and for all $a, b \in \mathbb{Z}_p$ we have $e(u^a, v^b) = e(u, v)^{ab}$; The map is non-degenerate, i.e., if $e(u, v) = 1$ then $u = 1$ or $v = 1$. There are efficient algorithms for computing group operations, evaluating the bilinear map, deciding membership of the groups, and sampling generators of the groups.

The main operations are defined as follows:

- ❖ *Key generation:* A private key x_j sampled from $\mathbb{Z}_p$, and the public key is set as $v_j = h^{x_j}$. A zero-knowledge proof of knowledge π_j for x_j can generated using a Fischlin transformed sigma protocol [37] or by signing the public key with its private counterpart.
- ❖ *Signing:* Given a message m, the prover computes a signature as $\sigma_j = H(m)^{x_j}$.
- ❖ *Aggregation:* Given message m and signatures $\sigma_1, \ldots, \sigma_n$, an aggregated signature A is computed as $A = \prod_{j=1}^{n} \sigma_j$.
- ❖ *Verification:* The verifier considers an aggregated signature to be valid if $e(A, h) = e\left(H(m), \prod_{j=1}^{n} v_j\right)$.

B. Data Structures

Transactions. All interaction with the Arbiter smart contract are conducted through Ethereum transactions. We encode the Snappy-specific information in the to *Data* field, as shown in Table I. Each Snappy transaction τ carries an identifier of the operation type τ_{op} (e.g., "Payment", "Registration", "Claim"). Moreover, transactions marked as "Payments" also include the address of the transacting merchant τ_m, a monotonically increasing payment index τ_i,[2] an aggregate τ_A of the statekeepers' approval signatures, and a vector of bits τ_q indicating which statekeepers contributed to the aggregate signature. τ_v denotes the amount of funds in τ.

[1] Boneh et al.'s scheme utilises specific bilinear pairings where there is an efficiently computable isomorphism between $\mathbb{G}$ and $\mathbb{H}$ which are not implemented over Ethereum. However, Boneh et al. [36] observed that the proof of security does still apply with respect to the more commonly used pairings under a stronger cryptographic assumption. We assume the signature scheme is implemented with regards to bilinear groups with no known isomorphism.

[2] We note that the Snappy transaction index τ_i is a separate field from the standard Ethereum transaction *nonce*. Although both are monotonically increasing counters, the Snappy index counts payments *inside* the Snappy system, while the Ethereum nonce counts all transactions by the same user, also *outside* the Snappy system.

Field	Symbol	Description
Customers	C	Customers
↪ *entry*	$C[c]$	Customer c entry
↪ *Collateral*	$C[c].\mathscr{col}_c$	Customer's collateral
↪ *Clearance*	$C[c].cl$	Clearance index
↪ *Finalized*	$C[c].D$	Finalized Transactions
↪ *entry*	$C[c].D[i]$	Entry for index i
↪ *Hash*	$C[c].D[i].H(\tau)$	Tx Hash
↪ *Signatures*	$C[c].D[i].\tau_A$	Aggregate signature
↪ *Quorum*	$C[c].D[i].\tau_q$	Approving parties
↪ *Bit*	$C[c].D[i].b$	Sig. verified flag
Merchants	M	Merchants
↪ *entry*	$M[m]$	Merchant m entry
Statekeepers	S	Statekeepers
↪ *entry*	$S[s]$	Statekeeper s entry
↪ *Allocation*	$S[s].\mathscr{col}_s[m]$	Value per merchant

TABLE II: **Arbiter's state.** The arbiter smart contract maintains a record for each customer, merchant and statekeeper.

Arbiter state. The Arbiter maintains state, shown in Table II, that consist of the following items: a *Customers* dictionary C that maps customer public keys to their deposited collateral $\mathscr{col}_c$, a clearance index cl, and a dictionary D of finalized transactions. D maps transaction index values to tuples that contain the hash of the transaction H(τ), τ_A, τ_q, and a bit b indicating whether the signature has been verified. Additionally, the Arbiter maintains a *Merchants* dictionary M containing the public keys of each registered merchant. The *Statekeepers* dictionary S maps the statekeeper public keys to a tuple with their collateral $\mathscr{col}_s$ and a vector d indicating how $\mathscr{col}_s$ is distributed between the merchants (by default, evenly).

Customer state. Each customer c maintains an ordered list $\mathcal{L}$ with its past *approved* transactions. We denote $\mathcal{S}_c$ the subset of those transactions from $\mathcal{L}$ that are still pending inclusion in the blockchain ($\mathcal{S}_c \subseteq \mathcal{L}$).

Statekeeper state. Each statekeeper maintains a list $\mathcal{P}$ of all the payment intents it has approved for each customer c.

Merchant state. Each merchant maintains a table $R[s]$ indicating how much collateral of each statekeeper s it can claim. Initially, $R[s] = \mathscr{col}_s/k$ for each statekeeper.

C. Registration

Customers can join and leave the system at any time, but the set of merchants and statekeepers is fixed after the initialization (in Appendix B we discuss dynamic sets of merchants and statekeepers).

Merchant registration. Merchant m submits a registration request which is a transaction that calls a function of the Arbiter contract. Once executed, the Arbiter adds a new entry $M[m]$ to its state, where $m = \tau_f$ is the merchant's account address (public key).

Customer registration. Customer c sends a transaction to the Arbiter with the funds that it intents to deposit for its collateral $\mathscr{col}_c$. The Arbiter contract creates a new customer entry $C[c]$,

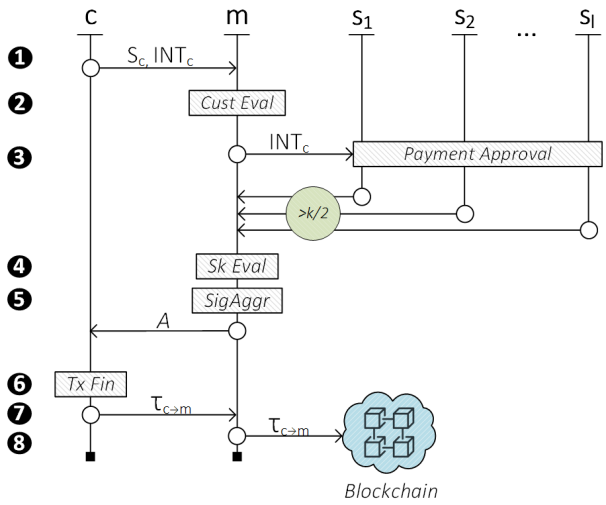

Fig. 3: **Payment-approval protocol.** Customer c initiates a payment to merchant m, who requests a majority approval from statekeepers $s_1, ..., s_l$. Merchant m aggregates the received responses and forwards them back to customer c, who creates and broadcasts the final transaction τ transferring the funds to the merchant.

sets the customer's collateral to τ_v, and initializes the clearance index to $cl = 0$. The Arbiter also initializes a new dictionary $C[c].D$ to log the registered customer's payments.

Statekeeper registration. The registration process for statekeepers is similar to that of customers. However, a statekeeper's registration includes also a proof of knowledge for their BLS private key x_j to prevent *rogue key attacks* (see [38] for details). The proof is included in the data field of the registration transaction and is verified by the Arbiter. Additionally, the statekeeper may also define how its collateral is to be distributed between the merchants. The maximum value each merchant can claim is $S[s].d$. We consider the case where each statekeeper collateral $\not{col}_s$ is equally allocated between all k merchants, and thus each merchant can recoup losses up to $\not{col}_s/k$ from statekeeper s.

D. Payment Approval

The payment-approval protocol proceeds as follows (Figure 3):

① *Payment initialization.* To initiate a payment, customer c creates a payment intent INT_c and sends it to merchant m, together with its state $\mathcal{S}_c$ that contains a list of already approved but still pending payments. The intent carries all the details of the payment (e.g., sender, recipient, amount, index) and has the same structure as normal transaction, but without the signatures that would make it a complete valid transaction.

② *Customer evaluation.* Upon reception, merchant m checks that all the index values in the interval $\{1 \dots \text{INT}_c[i]\}$ appear either in the blockchain as finalized transactions or in $\mathcal{S}_c$ as approved but pending transactions. If this check is successful, m proceeds to verify that $\not{col}_c$ suffices to compensate all of the currently pending transactions of that customer (i.e., $\not{col}_c \geq \sum \mathcal{S}_c + \text{INT}_c[v]$). Finally, merchant m verifies the approval signatures of the customer's transactions both in $\mathcal{S}_c$ and in the blockchain.

③ *Payment approval.* Next, the payment intent must be approved by a majority of the statekeepers. The merchant forwards INT_c to all statekeepers, who individually compare it with c's payment intents they have approved in the past (list $\mathcal{P}$ in their state). If no intent with the same *index* τ_i is found, statekeeper s appends INT_c in $\mathcal{P}$, computes a BLS signature σ_s over INT_c, and sends σ_s back to the merchant. If the statekeeper finds a past approved intent from c with the same index value, it aborts and notifies the merchant.

④ *Statekeeper evaluation.* Upon receiving the approval signatures from a majority of the statekeepers, merchant m uses $R[s]$ to check that each approving statekeeper s has sufficient collateral remaining to cover the current payment value τ_v. In particular, the merchant verifies that none of the approving statekeepers have previously approved pending payments, whose sum together with the current payment exceeds $R[s]$. (Merchants can check the blockchain periodically and update their R depending on the pending payments that got finalized on-chain.) Recall that initially $R[s]$ is set to $\not{col}_s/k$ for each statekeeper, but in case m has filed a settlement claims in the past against s, the remaining collateral allocated for m may have been reduced. This check ensures that in case one or more statekeepers equivocate, m is able to recover its losses in full.

⑤ *Signature aggregation.* If the statekeepers evaluation succeeds, m aggregates the approval signatures $\sigma_1, \dots, \sigma_{\lceil (k+1)/2 \rceil}$ (i.e, $A = \prod_{j=1}^{\lceil (k+1)/2 \rceil} \sigma_j$) and sends to customer c the resulting aggregate A and a bit vector q indicating which statekeepers' signatures were included in A. Otherwise, if one or more statekeepers do not have sufficient collateral to approve INT_c, the merchant can either contact additional statekeepers or abort.

⑥ *Transaction finalization.* The customer signs and returns to m a transaction τ with the details of INT_c, as well as $\tau_{to} =$ Arbiter, $\tau_m = m$, $\tau_A = A$ and $\tau_q = q$.

⑦ *Payment acceptance.* Merchant m first verifies the details and the signatures of the transaction τ, and then hands over the purchased goods or service to the customer. Finally, m broadcasts τ to the blockchain network. (Merchants have no incentive to withhold payments, but if this happens, the customer can simply broadcast it after a timeout.)

⑧ *Payment logging.* Once τ is added in a block, the Arbiter records the payment and forwards its value to merchant m (Algorithm 1). To keep the payments inexpensive in Ethereum, our smart contract does *not* verify the aggregate signature τ_A during logging. Instead, it performs the expensive signature verification only in cases of disputes, which are expected to be rare. In Section V, we show that such optimization is safe.

E. Claim Settlement

If a transaction does not appear in the blockchain within a reasonable time, the affected merchant may inspect the blockchain to determine the cause of the delay. There are three possible reasons for a transaction not getting included in the blockchain in a timely manner:

Algorithm 1: Record-and-Forward. Arbiter records the transaction and forwards the payment value to the merchant.

Actor : Arbiter (smart contract)
Input : Snappy transaction τ
Output : None

1 $c \leftarrow \tau_f$
2 **if** $c \in C$ *and* $\tau_i \notin C[c].D$ **then**
3 $\quad h \leftarrow H(\tau_f, \tau_{to}, \tau_v, \tau_i)$
4 $\quad C[c].D[\tau_i] \leftarrow \langle h, \tau_A, \tau_q, 0\rangle$
5 $\quad$ Send(m, τ_v) ▷ Forward the tx's funds to merchant
6 **else**
7 $\quad$ Send(s, τ_v) ▷ Return funds to sender

Algorithm 2: Claim-Settlement. Arbiter processes settlement claim using customers' and statekeepers' collaterals.

Actor : Arbiter (smart contract)
Input : Pending transaction τ^p
Ordered list of pending transactions $\mathbb{T}_p$
List of conflicting transaction tuples $\mathbb{T}_{\text{cnfl}}$ (optional)
Output : None

1 **if** *Verify*(τ^p_A) **then**
2 $\quad r \leftarrow$ Claim-Customer$(\tau^p, \mathbb{T}_p)$
3 $\quad$ **if** $r > 0$ **then**
4 $\quad\quad$ Claim-Statekeeper$(\tau^p, \mathbb{T}_{\text{cnfl}}, r)$
5 $\quad c \leftarrow \tau^p_f$
6 $\quad C[c].D[\tau_i] \leftarrow \langle h, \tau_A, \tau_q, 1\rangle$ ▷ Log tx as processed

1. *Benign congestion.* A transaction may be of lower priority for the miners compared to other transactions awarding higher miner fees.
2. *Conflicting transaction.* Another transaction by the same customer prevents the transaction from being included in the blockchain (doubles-pending attack). In Ethereum, two transactions from the same customer conflict when they share the same *nonce* value [1].
3. *Customer's account depletion.* A recent transaction left insufficient funds in the customer's account. Ethereum processes the transactions of each customer in increasing nonce order, and thus transactions with greater *nonce* values may be invalidated by those preceding them.

In the first case, the merchant must simply wait and check again later. In the latter two cases, the merchant can initiate a settlement claim to recoup the lost value τ_v. Such claims are sent to the Arbiter and include: (1) the pending transaction τ^p for which the merchant requests settlement, (2) a list $\mathbb{T}_p$ with all preceding (lower index) and pending Snappy transactions, and (3) optionally a list of conflicting transaction tuples $\mathbb{T}_{\text{cnfl}}$. The merchant can construct $\mathbb{T}_p$ from the $\mathcal{S}_c$ (received at payment initialization) by removing the transactions that are no longer pending ($\mathbb{T}_p \subseteq \mathcal{S}_c$).

Settlement overview. Algorithm 2 describes how the Arbiter executes settlement. First, it checks the majority approval by verifying the BLS signature aggregate τ^p_A. Then, it tries to recoup the transaction value τ^p_v from the customer's collateral using the *Claim-Customer sub-routine*. If the collateral does not suffice to cover the transaction value τ_v, it proceeds to claim the remaining amount from the collateral of the equivocating statekeeper using the *Claim-Statekeeper sub-routine*. Finally, the contract logs τ_i so that no further claims from the customer's collateral can be made for the same index τ_i.

Claim-Customer sub-routine is shown in Algorithm 3. First

Algorithm 3: Claim-Customer. Arbiter recovers lost funds from the customer's collateral or returns the remaining amount.

Actor : Arbiter (smart contract)
Input : Pending transaction τ^p
Preceding pending transactions $\mathbb{T}_p$
Output : Residual r or $\perp$

1 $c \leftarrow \tau^p_f$
2 I$^* \leftarrow C[c].D$ ▷ Pass by reference
/* Verify signatures of preceding non-pending txs. */
3 **for** $\forall\{i \in I^* | I^*[i].b = 0\}$ **do**
4 $\quad$ **if** *Verify*$(I^*[i]_A)$ **then**
5 $\quad\quad I[i].b \leftarrow 1$
6 $\quad$ **else**
7 $\quad\quad$ del I^*[i] ▷ Past tx had no approval.
/* Any pending & preceding txs missing? */
8 **if** $\exists\, i \in \{1 \dots \tau^p_{i-1}\}$ *such that* $i \notin I^*$ *and* $i \notin \mathbb{T}_p$ **then**
9 $\quad$ **return** $\perp$
/* Verify signatures of pending preceding txs. */
10 **if** !*Verify*$(\mathbb{T}_p)$ **then**
11 $\quad$ **return** $\perp$
/* Process claim. */
12 **if** $\tau^p_i \notin I^*$ **then**
13 $\quad \cancel{col}^*_c \leftarrow C[c].\cancel{col}_c$ ▷ Pass by reference
14 $\quad cov \leftarrow max(0, \cancel{col}^*_c - \sum \mathbb{T}_p)$ ▷ Max claimable amount
15 $\quad \rho \leftarrow min(cov, \tau^p_v)$
16 $\quad \cancel{col}_c \leftarrow \cancel{col}^*_c - \rho$
17 $\quad$ Send(τ^p_m, ρ)
18 $\quad$ r $\leftarrow \tau^p_v - \rho$
19 $\quad$ **return** r
20 **else**
21 $\quad$ **return** τ_v ▷ Statekeeper equivocated

(lines 3-7), it verifies the signatures of preceding finalized transactions that have not been verified already (recall that signatures are not verified during Record-and-Forward to reduce payment processing cost). Subsequently, the contract checks that the list $\mathbb{T}_p$ contains all the pending transactions preceding τ^p (lines 8-9) and verifies their signatures (lines 10-11). For every transaction that is deleted (line 7), the merchant should have included another transaction with the same index and valid approval signature in $\mathbb{T}_p$. These two checks ensure that the merchant did its due diligence and verified the approval signatures of preceding pending and non-pending transactions during "Customer evaluation" step (Section IV-D).

After that, the contract checks for another valid transaction with the same index, and if there was it returns τ^p_v to its parent process (line 21). In this case, the losses should be claimed from the equivocating statekeepers' collaterals. Otherwise, the arbiter compensates the merchant from the customer's collateral $\cancel{col}_c$. In case the customer's collateral does not suffice to fully recoup τ^p_v, then the algorithm returns the remaining amount (line 19). This amount is then used in further claims against the statekeepers.

The Snappy system supports arbitrarily many past transactions. However, in practice, an expiration window can be used to reduce the computational load of verifying the signatures of preceding transactions. For example, a 24-hour window would allow sufficient time to accept payments and claim settlements if needed, and at the same time keep the number of past-transaction verifications small. Such time window would also reduce the storage requirements of the Arbiter smart contract, as it would only need to store the most recent transactions

Algorithm 4: Claim-Statekeeper. Arbiter sends lost funds from the misbehaving statekeepers' collaterals to the affected merchant.

```
   Actor  : Arbiter (smart contract)
   Input  : Residual r
            Pending Transaction τ^p
            Conflicting transaction tuples T_cnfl
   Output : None
1  left ← r
2  while left > 0 and T_cnfl ≠ ∅ do
3      ⟨τ', τ''⟩ ← T_cnfl.pop()          ▷ Tuple of txs with the same idx
4      if Verify(τ') and Verify(τ'') and
5          τ'_i ≤ τ^p_i and τ''_i ≤ τ^p_i and τ' ≠ τ'' then
6          sk ← FindOverlap(τ', τ'')      ▷ Find who equivocated
7          if sk ≠ ∅ then
8              δ ← |τ'_v − τ''_v|
9              col*_sk ← S[sk].col_sk[τ^p_m]      ▷ Pass by reference
               /* Is there enough collateral left?
                  */
10             if col*_sk[m] − δ ≥ 0 then
11                 col*_sk[m] ← col*_sk[m] − δ
12             left ← left − δ
13 ρ ← min(τ^p_v, |τ^p_v − left|)
14 Send(τ^p_m, ρ)
```

from each user in $C[c].D$, instead of all the past transactions. Note that valid payments will appear in the blockchain within a few minutes and thus the operators' collateral will be quickly freed to be allocated to other transactions.

Claim-Statekeepers sub-routine is shown in Algorithm 4. It is executed in cases where the customer's collateral does not suffice to fully recoup the value of the transaction τ_v^p for which settlement is requested. In this case, the arbiter attempts to recover the lost funds from the equivocating statekeepers.

The Arbiter iterates over the tuples of conflicting (and preceding) transactions until τ_v^p has been fully recovered or there are no more tuples to be processed (line 2). For each of those tuples, the Arbiter does the following: First, it verifies the approval signatures of the two transactions, checks that the two transactions precede τ^p and that they are not identical (lines 4-5). Then, it uses *FindOverlap*() based on bit strings τ_q of processed transactions to identify the equivocating statekeepers (line 6). Finally, it computes the amount that the merchant was deceived for ($|\tau'_v - \tau''_v|$) and subtracts it from the collateral of one of the statekeepers (lines 8-11).

We only deduct the missing funds from one of the equivocating statekeepers in each tuple, as our goal is to recoup the τ_v^p in full. However, Snappy can be easily modified to punish all the equivocating statekeepers. Recall that each merchant is responsible for ensuring that the collateral allocated by each statekeeper for them, suffices to cover the total value of pending payments approved by that statekeeper (line 10).

F. De-registration

Customers can leave the Snappy system at any point in time, but in the typical usage de-registrations are rare operations (comparable to credit card cancellations). The process is carried out in two steps, first by clearing the customer's pending transactions and subsequently by returning any remaining collateral. This two-step process allows enough time for any pending settlements to be processed before the collateral is returned to the customer.

The customer submits a clearance request that updates the clearance field $C[c].cl$ to the current block number. At this point, the customer's account enters a clearance state that lasts for a predetermined period of time (e.g., 24 hours). During this time, customer c can no longer initiate purchases, but $\not{col}_c$ can still be claimed as part of a settlement. Once the clearance is complete, the customer can withdraw any remaining collateral by submitting a withdrawal request. The Arbiter checks that enough time between the two procedures has elapsed and then returns the remaining $\not{col}_c$ to customer c.

V. Snappy Analysis

In this section, we analyze the safety and liveness properties of Snappy.

A. Safety

First we prove that a merchant who follows the Snappy protocol to accept a fast payment is guaranteed to receive the full value of the payment (our Requirement R1) given the strong adversary defined in Section II-B.

Definitions. We use $\mathcal{BC}$ to denote all the transactions in the chain, and $\mathcal{BC}[c]$ to refer to the subset where c is the sender. We say that a customer's state $\mathcal{S}_c$ is *compatible* with a transaction τ when all τ' such that $\tau' \notin \mathcal{BC}$ and $\tau'_i < \tau_i$, $\tau \in \mathcal{S}_c$ and $\sum \mathcal{S}_c + \tau_v \leq \not{col}_c$. Less formally, compatibility means that $\mathcal{S}_c$ should include all of c's pending Snappy transactions that precede τ, while their total value should not exceed $\not{col}_c$.

Theorem 1. *Given a customer state $\mathcal{S}_c$ that is* compatible *with a transaction τ approved by a majority of the statekeepers with sufficient collaterals, merchant τ_m is guaranteed to receive the full value of τ.*

Proof: A Snappy transaction τ transfers funds from customer c to merchant m. Here, we use $\not{col}_c$ to denote the value of the c's collateral when τ was approved, and $\not{col}'_c$ to denote the collateral's value at a later point in time. Note that $\not{col}'_c$ is potentially smaller than $\not{col}_c$ as several settlement claims may have been processed in the meantime.

To prove Theorem 1, we consider both the case where τ is included in the blockchain $\mathcal{BC}$, and the case it is not. In the former case ($\tau \in \mathcal{BC}$), merchant m is guaranteed to receive the full payment value τ_v, as the adversary cannot violate the integrity of the Arbiter smart contract or prevent its execution. Once transaction τ is in the blockchain, the Arbiter contract executes Record-and-Forward (Algorithm 1) that sends the full payment value τ_v to merchant m.

In the latter case ($\tau \notin \mathcal{BC}$), the merchant sends a settlement claim request that causes the Arbiter contract to execute Claim-Settlement (Algorithm 2). There are now two cases to consider: either $\not{col}'_c$ suffices to cover τ_v or it does not. In the first case ($\tau_v \leq \not{col}'_c$), merchant m can recover all lost funds from $\not{col}'_c$. For this process, m needs to simply provide the approved τ and the list of Snappy transactions from customer c that preceded τ and are still pending. The Arbiter contract verifies the validity of the inputs and sends the payment value τ_v to merchant m from $\not{col}'_c$. In the latter case ($\tau_v > \not{col}'_c$), merchant m can still recoup any lost funds, as long as $\mathcal{S}_c$ was *compatible* with τ and m followed the Snappy payment

approval protocol and verified before payment acceptance that the approving statekeepers' remaining collaterals $R[s]$ suffice to cover τ_v and all other pending transactions previously approved by the same statekeeper.

According to Lemma 1, if $\tau_v > \not{col}'_c$, then there are more than one approved transactions from customer c with the same *index* value τ_i. As shown by Proposition 1, for this to happen, one or more statekeepers need to approve two transactions with the same index (i.e., equivocate). The arbiter contract can find the equivocating statekeepers by comparing the quorum bit vectors τ_q and τ'_q from the conflicting transaction tuples, and recoups all lost funds from their collaterals.[3]

In Snappy, it is the responsibility of the merchant to verify that each statekeeper who approves a payment has sufficient collateral remaining. Before payment acceptance, the merchant verifies that the sum of all approved but pending payments and the current payment are fully covered by $R[s]$ which is initially $\not{col}_s/k$ but can be reduced by previous settlement claims. Since one merchant cannot claim collateral allocated for another merchant, the merchant is guaranteed to be able recover the full payment value, even if one or more statekeepers equivocate to several merchants simultaneously. ■

Lemmas. We now provide proofs for Lemma 1 and Proposition 1 used above.

Lemma 1. *Let $\mathcal{S}_c$ be state of customer c that is* compatible *with a transaction τ, if at any point $\not{col}'_c < \tau_v$ and τ has not been processed, then there exists an approved τ' such that $\tau' \notin \mathcal{S}_c$, and τ' has the same index either with τ or a transaction in $\mathcal{S}_c$.*

Proof: Let $\not{col}'_c$ be the funds left in the customer's collateral after a set of claims Π were processed i.e., $\not{col}'_c = \not{col}_c - \sum \Pi$. Let's now assume that $\not{col}'_c < \tau_v$, but there is no τ' that (1) is not in $\mathcal{S}_c$ and (2) has the same index with τ or with a transaction in $\mathcal{S}_c$. In other words, let's assume that no two approved transactions have the same index i.

From $\tau_v > \not{col}'_c$, it follows that $\sum \Pi + \tau_v > \not{col}_c$. By definition $\sum \mathcal{S}_c + \tau_v \leq \not{col}_c$. Thus, it follows that:

$$\sum \mathcal{S}_c + \tau_v < \sum \Pi + \tau_v \Rightarrow \sum \mathcal{S}_c < \sum \Pi$$

From this, we derive that $\Pi \not\subset \mathcal{S}_c$. Since, Π is not a subset of $\mathcal{S}_c$, there is at least one transaction $\tau' \in \Pi$ such that $\tau' \notin \mathcal{S}_c$. $\mathcal{S}_c$ is compatible with τ, and thus $\mathcal{S}_c$ contains *all* the pending transactions up to τ_i. As a result, a pending τ' that is not included in $\mathcal{S}_c$ must have τ'_i greater than τ_i. Note that if τ' is not pending, the Arbiter contract will not process the settlement claim (line 12 in Algorithm 3). Thus, $\tau_i < \tau'_i$.

According to Algorithm 3, any claim for τ' should include all transactions preceding τ' that are still pending. Since, τ is pending and $\tau_i < \tau'_i$, the claim should also include τ. However, Line 14 in Algorithm 3 ensures that enough funds are left in the customer's collateral to cover all the preceding-pending transactions. This covers τ too, and contradicts our starting premise $\tau_v > \not{col}'_c$. We have so far shown that:

(1) If $\not{col}'_c < \tau_v$, then the arbiter processed a claim for a transaction τ' that is not included in $\mathcal{S}_c$.
(2) The collateral will always suffice to cover τ, even if other claims for transactions with greater index values (i.e., $\tau'_i > \tau_i$) are processed first.

From (1) and (2), it follows that $\tau'_i \leq \tau_i$. ■

Proposition 1. *For any two majority subsets $M_1 \subseteq S$ and $M_2 \subseteq S$, where S is the set of all the statekeepers, it holds that $M_1 \cap M_2 \neq \emptyset$.*

Proof: The proof of this proposition follows directly from the Pidgeonhole principle (or Dirichlet's box principle) which states that if n items are put in m containers for $n > m$ then at least one container contains multiple items. ■

No penalization of honest statekeepers. Above we showed that an honest merchant who accepts a Snappy payment will always receive his funds. Additionally, Snappy ensures that an honest statekeeper cannot be penalized due to benign failures such as network errors or crashes. As outlined in Section IV, a merchant is penalized only if it approves two conflicting transactions (same index, same customer). This simple policy is sufficient to protect all honest merchants in the case of benign failures.

Privacy Payment privacy is largely orthogonal to our solution and inherited from the underlying blockchain. For example, if Snappy is built on Ethereum, for most parts Snappy customers and merchants receive the level of privacy protection that Ethereum transactions provide. In Section VII, we discuss privacy in more detail.

B. Liveness

Next, we explain the liveness property of Snappy. Payment processing is guaranteed when at least $\lceil (k+1)/2 \rceil$ of the statekeepers are reachable and responsive. If we assume that all the statekeeping nodes are equally reliable then each one of them should have an availability of only $\lceil (k+1)/2 \rceil$. We note that Snappy ensures safety and liveness under slightly different conditions. In that regard Snappy is similar to many other payment systems where liveness requires that the payment processor is reachable but the same is not needed for payment safety.

VI. Snappy Evaluation

In this section we evaluate Snappy and compare it with previous solutions.

A. Latency

In Snappy, the payment approval latency depends on two factors: (a) the number of approving statekeepers and (b) the speed/bandwidth of the network links between the merchants and the statekeepers. The number of customers and merchants has no effect on the approval latency and its impact on the collateral size is discussed in Section VI-B.

[3] Lines 8-9 in the Claim-Customer sub-routine (Algorithm 3) force any actor who claims τ_v to publish a list with transactions that are pending and have *indices* preceding τ_i. This enables other merchants to identify conflicting transaction pairs, find the equivocating statekeepers and claim from their collaterals.

To evaluate our Requirement R1 (fast payments), we simulated a setup with several globally-distributed statekeepers and merchants running on Amazon EC2 instances. Both the statekeepers and the merchants were implemented as multi-threaded socket servers/clients in Python 3.7 and used low-end machines with 2 vCPUs and 2 GB of RAM. We distributed our nodes in 10 geographic regions (4 different locations in the US, 3 cities in the EU, and 3 cities in the Asia Pacific region).

As seen in Figure 4, we tested the payment approval latency for different numbers of statekeepers and various rates of incoming requests. In our first experiment, we initialized 100 merchants who collectively generate 1,000 approval requests per second. We observe that for smaller consortia of up to 40 statekeepers (i.e., at least 21 approvals needed), Snappy approves the requests within 300ms, while larger consortia require up to 550ms on average. This shows that the approval latency increases sub-linearly to the number of statekeepers and experimentally validates the low communication complexity of Snappy. In particular, the latency doubles for a 5-fold increase in the consortium's size i.e., 40 statekeepers require ∼300ms to collectively approve a request while 200 statekeepers require ∼550ms. We also tested our deployment for higher loads: 2,500 and 5,000 requests per second respectively. Our measurements show a slightly increased latency due to the higher resource utilization at the statekeeping nodes. However, the relationship between the number of statekeepers and the approval latency remains sub-linear. We observe, in Figure 4, that the variance increases both with the number of statekeepers and the throughput. However, in all cases the payment approvals were completed in less than a second.

These results demonstrate the practicality of Snappy as it remains under the 2-second approval latency mark that we consider a reliable user experience benchmark [4]–[6]. The measured latencies are consistent with past studies measuring the round-trip times (RTT) worldwide [39] and within the Bitcoin network [40], [41]. In particular, the majority of the bitcoin nodes have an RTT equal or less to 500ms, while only a tiny fraction of the network exhibit an RTT larger than 1,5 seconds.

An optimized Snappy deployment that uses more capable machines will likely achieve better timings for the aforementioned loads and even larger statekeeping consortia. We did not perform such performance optimizations, as Snappy is best suited to deployments where the number of statekeepers is moderate (e.g., $k = 100$). For larger consortia, the statekeeper collaterals grow large (Section VI-D) and a centralized-but-trustless deployment is preferable (Appendix B).

B. Scalability

We now evaluate how many customers and merchants Snappy can handle (Requirement R2–large deployments).

Regarding the number of customers, the only scalability issue is that the recent transactions of each customer (e.g., past 24 hours) need to be recorded to the Arbiter's state. Thus, its storage needs grow linearly to the number of customers. Since Ethereum contracts do not have a storage limit, our implementation can facilitate hundreds of thousands or even millions of customers. In blockchain systems that allow smart contracts to examine past transactions, there is no need for the Arbiter to log transactions into the contract's state and the required storage is reduced significantly.

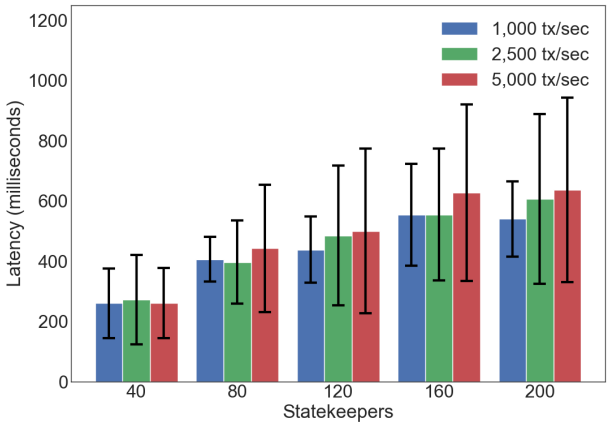

Fig. 4: **Payment approval latency.** Payment approval latency for varying rates of incoming approval requests that each corresponds to one purchase.

As shown in the previous section, Snappy can easily support 100-200 statekeeping merchants which is the intended usage scale of our solution. Moreover, due to the low communication complexity of our protocol, an optimized deployment could also support a few thousand statekeeping merchants with approval latency of less than 4 seconds. However, in cases of very large deployments with several thousands merchants, it is preferable to allow merchants to decide if they want to deposit a collateral and perform statekeeping tasks or simply be able to receive Snappy payments. Such a deployment remains trustless towards the statekeeping merchants and decentralized (but not fully), while it can support a much larger number of non-statekeeping merchants. This design is further discussed in Appendix B.

C. Processing Cost

To evaluate our Requirement R3 (cheap payments), we implemented the Arbiter smart contract in Solidity for Ethereum, and measured the Ethereum gas cost of all Snappy operations. Our cost evaluation corresponds to a case where merchants run a Snappy instance non-profit. Joining a non-profit consortium allows merchants to accept fast and safe payments without having to pay fees to external entities such as card payment processors. Additional fees may be charged in for-profit setups. Table III summarizes our results and provides the USD equivalents using the current conversion rate and a Gas price (Gwei) of 7.8.

Registration cost. The one-time registration cost for merchants and customers is very low ($67,000$ Gas that equals to $\$0.06$), while statekeepers have to pay a slightly increased cost ($\$0.48$), due to verification of the proof of knowledge for the statekeeper's BLS private key to prevent *rogue key* attacks [38]. The cost of the collateral clearance and withdrawal operations for both customers and statekeepers are also inexpensive, requiring $\$0.04$ and $\$0.02$.

Payment cost. The cost of a payment, in the absence of an attack, is $169,000$ Gas ($\$0.16$), mostly due to the cost of

TABLE III: Cost of Snappy operations.

Function	Gas	USD
Client/Merchant Registration	67,000	0.06
Statekeeper Registration	510,000	0.48
Clear Collateral	42,000	0.04
Withdraw Collateral	23,000	0.02
Process Payment	169,000	**0.16**

TABLE IV: Worst-case claim settlement cost in Gas and USD.

Minimum Majority	Pending Transactions per Customer			
	0	1	2	3
50	1.9M ($1.79)	2.7M ($2.54)	3.5M ($3.30)	4.3M ($4.05)
100	3.1M ($2.92)	4.3M ($4.05)	5.5M ($5.19)	6.6M ($6.22)
150	4.2M ($3.96)	5.8M ($5.47)	7.4M ($6.98)	9.0M ($8.49)
200	5.4M ($5.09)	7.4M ($6.984)	9.4M ($8.87)	11.4M ($10.75)
250	6.6M ($6.22)	9.0M ($8.494)	11.3M ($10.66)	13.7M ($12.93)

storing information about the transaction. This is roughly eight times as expensive as a normal Ethereum transaction that does not invoke a smart contract. In comparison, Mastercard service fees are $\sim$1.5% of the transaction value [42]. For example, a payment of \$15 will cost to the merchant \$0.22, while in a payment of \$100 the fees will rise to \$1.5. Snappy compares favorably to these charges. BLS signatures [33] enabled us to aggregate the approval signatures, significantly reducing both the transaction size and the processing costs.

Claim settlement cost. While our solution enables merchants to accept payments from customers with arbitrarily many pending transactions (constrained only by the customer's own collateral), the Ethereum VM and the block gas limits constrain the computations that can be performed during the settlement process. To examine these constraints, we consider the gas costs for different numbers of pending transactions, and statekeeper quorum sizes. While the number of the statekeepers is seemingly unrelated, it affects the cost of the settlement due to the aggregation of the public keys. In Appendix B, we discuss a special case, where each customer is allowed to have only one transaction pending which simplifies settlement and reduces its cost significantly.

Table IV shows the worst-case gas costs of settlement which is few dollars for a typical case (e.g., \$1.79 when $k = 100$ and there is no pending transactions).[4] Given Ethreum's gas limit of approximately 8M gas units per block, Snappy claim settlement can either scale up to $k = 499$ statekeepers (250 approvals) with no pending transactions (6.6M Gas), or to 3 pending transactions per customer (6.6M Gas) with $k = 199$ statekeepers (100 approvals).

D. Collateral Comparison

To evaluate our Requirement R2 (practical collaterals), we compare our deposits to known Layer-2 solutions, as shown in Table V. For our comparisons, we use example parameter values that are derived from real-life retail business cases.

[4]These costs were calculated by assuming that the adversary specifically crafts previous transactions to maximize the computational load of the settlement. To prevent an adversary from increasing the claim processing costs, the Arbiter contract could reject any transactions that have more approvals than the minimum necessary.

TABLE V: Collateral comparison.

Solution	Customer	Operator	
		Individual	Combined
Channels [8], [9]	$e \cdot k$		
↪ Small Shops	\$1,000		
↪ Large Retailers	\$25,000		
Hubs [12], [13]	e	$\sum_n e$	
↪ Small Shops	\$10	\$1M	
↪ Large Retailers	\$250	\$250M	
Snappy	$\max(e_t)$	$\sum_k \max(e_t) \cdot p_t$	$\sum_k \max(e_t) \cdot p_t \cdot k$
↪ Small Shops	\$5	\$3,000	\$300,000
↪ Large Retailers	\$100	\$150,000	\$15M

Our "small shops" scenario is based on sales numbers from [43]–[45]. The system has $n = 100,000$ customers and $k = 100$ merchants. The daily average expenditure of a customer per merchant is $e = \$10$ and the expenditure of a customer within $t = 3$ minutes blockchain latency period is $e_t = \$5$. The number payments received by a merchant within the same time-period is $p_t = 6$ (i.e., one customer payment every 30 seconds).

Our "large retailers" example corresponds to the annual sales of a large retailer in the UK [21], [22]. In this case, we have $n = 1$ million customers, $k = 100$ merchants, $e = \$250$, $e_t = \$100$ and $p_t = 15$ (i.e., one payment every 12 seconds).

Customer collateral. In payment channels, the customer collateral grows linearly with the number of merchants. This leads to large customer collaterals ($\$1,000$ and $\$25,000$) in our example cases. Payment hubs alleviate this problem, but they still require customers to deposit their anticipated expenditure for a specific duration (e.g., $e = \$250$), and replenish it frequently. Snappy requires that customers deposit a collateral that is never spent (unless there is an attack) and equals the maximum value of payments they may conduct within the blockchain latency period (e.g., $e_t = \$100$).

Merchant collateral. In payment hubs, the operator's deposit grows linearly with the number of customers in the system, since the operator must deposit funds equal to the sum of the customers' deposits [12]–[14]. That is, the operator collateral must account for all registered customers, *including the currently inactive ones*. Given our examples, this amounts to \$1M and \$250M for the "small shops" and the "larger retailers" cases, respectively.

In Snappy, each merchant that operates as a statekeeper deposits enough funds to cover the total value of sales that the merchants conduct within the latency period t. Once a transaction gets successfully finalized on the blockchain (i.e., after t), the statekeeper can reuse that collateral in approvals of other payments. Thus, the total size of this collateral is independent of the number of registered customers, and is proportional to the volume of sales that merchants handle. In other words, the statekeeper collateral accounts only for the customers that are *active* within the 3-minute latency period. Given our examples, this amounts to \$3,000 and \$150,000 which are three orders of magnitude less than in payment hubs. The combined deposit by all statekeepers (merchants) is shown on the last column of Table V. In both of our example cases, the combined deposit is smaller than in payment hubs.

Designing a payment hub where the operator collateral is proportional to only active customers (and not all registered customers) is a non-trivial task, because the deposited collateral cannot be freely moved from one customer to another [30]. Some commit-chains variants [31] manage to reduce operator collaterals compared to hubs, but such systems cannot provide secure *and* fast payments. Other commit-chain variants [31] enable fast and safe payments, but require online monitoring that is not feasible for all retail customers. Thus, we do not consider commit chains directly comparable and omit them here (see Section VIII for more details).

Cost of operation. The amount of locked-in funds by system operators allows us to approximate the cost of operating Snappy system with respect to other solutions. For example, in our retail example case, each merchant that runs a statekeeper needs to deposit \$150k. Assuming 7% annual return of investment, the loss of opportunity for the locked-in money is \$10,500 per year which, together with operational expenses like electricity and Internet, gives the operational cost of Snappy. We consider this an acceptable cost for large retailers. In comparison, a payment hub operator needs to deposit \$250M which means that running such a hub would cost \$17.5M plus operational expenses which is three orders of magnitude more.

VII. Discussion

A. Governance

Snappy has, by design, a majority-based governance model. This means that any majority of statekeeping nodes can decide to ostracize merchants or statekeepers that are no longer deemed fit to participate. For example, a majority can stop processing requests from a merchant who has equivocated (signed conflicting transactions). If more complex governance processes are needed (e.g., first-past-the-post voting) the Arbiter's smart contract can be extended accordingly.

B. Censorship Mitigation

A direct implication of Snappy's governance model is that a majority of statekeeping nodes can discriminate against a particular victim merchant by not responding to its payment approval requests or by delaying the processing of its requests. Such censorship can be addressed in two ways.

The first approach is technical. Targeted discrimination against a specific victim merchant could be prevented by hiding the recipient merchant's identity during the payment approval process. In particular, by replacing all the fields that could be used to identify the merchant (e.g., "recipient", "amount") from the payment Intent with a cryptographic commitment. Commitments conceal the merchant's identity from other merchants (statekeepers) during payment approval, but allow the merchant to claim any lost funds from the Arbiter later by opening the commitment. Moreover, if IP address fingerprinting is a concern, merchants can send their approval requests through an anonymity network (e.g., Tor would increase latency by $\sim$500ms [46]) or through the customer's device so as to eliminate direct communication between competing merchants.

The second mitigation approach is non-technical. In case of known merchant identities and a mutually-trusted authority (e.g., a merchants' association), the victim merchant can file a complaint against constantly misbehaving merchants. In cases of widespread attacks, the victim merchant can reclaim their collaterals in full, deregister from this consortium and join another consortium.

C. Transaction Privacy

For on-chain transaction privacy, Snappy inherits the privacy level of the underlying blockchain. For example, Ethereum provides transaction *pseudonymity*, and thus every transaction that is processed with Snappy is pseudonymous once it recorded on the chain.

During payment approval, the identity of the recipient merchant can be concealed from all the statekeepers using cryptographic commitments, as explained above (see Section VII-B). However, the pseudonym of the customer remains visible to the statekeepers.

Well-known privacy-enhancing practices like multiple addresses and mixing services [47], [48] can be used to enhance customer privacy. For example, a Snappy customer could generate several Ethereum accounts, register them with the Arbiter and use each one of them only for a single payment. Once all accounts have been used, the customer can de-register them, generate a new set of accounts, move the money to the new accounts through a mixing service, and register new accounts. The main drawback of this approach is that the user needs to have more collateral locked-in and will pay the registration fee multiple times.

In the future, privacy-preserving blockchains like ZCash [49] combined with private smart contracts based on Non-Interactive Zero-Knowledge proofs (NIZKs) could address the on-chain confidentiality problem more efficiently and protect the privacy of both the users and the merchants. However, realizing such a secure, efficient and private smartcontract language while achieving decent expressiveness, remains an open research problem [50].

D. Limitations

The main drawbacks of using Snappy are as follows. First, customers and merchants need to place small collaterals, and thus keep a percentage of their funds locked-in for extended periods of time. Second, Snappy can scale up to a moderate number of statekeeping merchants but cannot support hundreds of thousands or millions statekeeping nodes. In such cases, alternative deployment options can be used (see Appendix B). Third, Snappy does not move the payment transactions off the chain and thus customers still need to cover the transaction processing fees charged by the blockchain's miners.

VIII. Related Work

Payment channels enable two parties to send funds to each other off the chain, while adding only an opening and a closing transaction on the chain [8], [9], [12]. With the opening transaction the two parties lock funds in the channel, which are then used throughout the lifetime of the channel. In cases where the two parties send approximately the same amount

of funds to each other over time, a payment channel can enable almost indefinite number of near-instant payments. However, in the retail setting customers send funds unilaterally towards merchants. Moreover, customers transact with several merchants and thus each customer will need to maintain several channels and keep enough funds in them.

Payment networks utilize the payment channels established between pairs of users to build longer paths [10], [11]. While this is a straightforward idea, in practice, finding routes reliably is not a trivial task [28]. This is because the state of the individual channels changes arbitrarily over time and thus the capacity of the graph's edges fluctuate. Moreover, the unilateral nature of retail payments (customer $\rightarrow$ merchant) quickly depletes the available funds in the individual channels, preventing them from serving as intermediaries to route payments by other customers [29]. Miller et. al [8] showed that even under favorable conditions (2.000 nodes, customers replenish their accounts every 10 seconds, maximum expenditure of \$20, no attacks), approximately 2% of the payments will fail. At peak hours the ability of the network to route payments from customers to merchants is expected to degrade further. Rebalancing methods [51] have only a meager effect, primarily because credit cycles are rarely formed in real life [8].

Payment hubs introducing a single central point connecting all customers to all merchants. This eliminates the need of finding routing paths and in theory require a smaller total amount of locked funds for the customers [30]. However, this approach comes with two main drawbacks. First, it introduces a single point of failure for payment availability. And second, the hub operator needs to deposit very large amount of funds to match the total expenditure of all customers [12], [13] and thus will likely charge service fees. For instance, a hub that serves $n = 1$M customers having in total of \$250M in their channels, must also lock-in that amount in channels with merchants to be able to accommodate payments, in particular during peak hours. Hub operators would charge significant fees to cover the opportunity cost of the large locked-in funds.

Commit-chains are parallel (and not yet peer-reviewed) work [30], [31], [52] that may either reduce or eliminate the operator collaterals compared to payment hubs. The main idea of commit-chains is to maintain a second-layer ledger and make periodic commitments (called *checkpoints*) of its state transitions to the main chain. In one proposed variant [31], the central operator does not have to place any collateral, but such scheme does not enable fast and safe payments, because users need to wait for the next checkpoint which may take hours or days. Another proposed variant [31] allows safe and fast payment, but has other problems. First, the users need to monitor the blockchain (hourly or daily) and dispute checkpoints if their balance is inaccurately represented. Such monitoring assumption is problematic, especially in use cases like retail with large number of customers using various client devices. Second, although the operator's collateral is slightly lower than those of payment hubs, it still remains very large (e.g., \$200M in our "large retailers" use case) [14]. Snappy enables fast and safe payments with smaller merchants collaterals for customers that remain mostly offline.

Side-chains use a permissioned set of validators to track pending transactions, typically using a BFT consensus protocol [16], [17]. Such solutions change the trust assumptions of permissionless blockchains significantly, as BFT consensus requires that 2/3 of the validators must be trusted. Side chains also require multiple rounds of communication and have high message complexity.

Probabilistic payments such as MICROPAY1/2/3 can in certain scenarios enable efficient and fast payment approval [53], [54]. However, such solutions require that the service provided is continuous and granular so that the payments' probabilistic variance becomes negligible. In retail payments, this provides no guarantee that the merchant will be paid the right amount.

IX. Conclusion

In this paper we have presented Snappy, a novel system that enables merchants to safely accept fast on-chain payments on slow blockchains. We have tailored our solution for settings such retail payments, where currently popular cryptocurrencies are not usable due to their high latency and previous solutions such as payment channels, networks and hubs have significant limitations that prevent their adoption in practice.

Acknowledgments

The authors would like to thank the anonymous reviewers, the shepherd Stefanie Roos, Mary Maller and George Danezis. This research has been partially supported by the Zurich Information Security and Privacy Center (ZISC).

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[2] G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Čapkun, "Misbehavior in bitcoin: A study of double-spending and accountability," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, 2015.

[3] M. Lei, "Exploiting bitcoin's topology for double-spend attacks," 2015.

[4] M. Freed-Finnegan and J. Koenig, "Visa quick chip," https://usa.visa.com/visa-everywhere/security/quick-chip-interview.html, 2017.

[5] Visa, "Visa quick chip for emv," https://vimeo.com/163309180, 2017.

[6] Capgemini, "The quick emv solution," https://www.capgemini.com/wp-content/uploads/2017/07/the_quick_emv_card_processing_2016_web.pdf, 2016.

[7] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Consensus in the age of blockchains," in *ACM Advances in Financial Technologies (AFT)*, 2019.

[8] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 508–526.

[9] J. Lind, O. Naor, I. Eyal, F. Kelbert, P. R. Pietzuch, and E. G. Sirer, "Teechain: Reducing storage costs on the blockchain with offline payment channels," in *ACM International Systems and Storage Conference (SYSTOR)*, 2018.

[10] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," *draft version 0.5*, vol. 9, p. 14, 2016.

[11] R. Network, "Raiden: Cheap, scalable token transfers for ethereum," https://raiden.network/, 2018.

[12] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[13] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg, "Tumblebit: An untrusted bitcoin-compatible anonymous payment hub," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[14] L. Gudgeon, P. McCorry, P. Moreno-Sanchez, A. Gervais, and S. Roos, "Sok: Off the chain transactions," 2019.

[15] P. McCorry, S. Bakshi, I. Bentov, A. Miller, and S. Meiklejohn, "Pisa: Arbitration outsourcing for state channels," *IACR Cryptology ePrint Archive*, 2018. [Online]. Available: https://eprint.iacr.org/2018/582

[16] J. Dilley, A. Poelstra, J. Wilkins, M. Piekarska, B. Gorlick, and M. Friedenbach, "Strong federations: An interoperable blockchain solution to centralized third-party risks," *arXiv preprint arXiv:1612.05491*, 2016.

[17] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains," https://blockstream.com/sidechains.pdf, 2014.

[18] Digiconomist, "Bitcoin energy consumption index," https://digiconomist.net/bitcoin-energy-consumption, 2019.

[19] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[20] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[21] T. PLC, "Tesco's christmas in numbers," https://se-report.tescoplc.com/news/news-releases/2016/tesco-christmas-in-numbers/, 2016.

[22] Tesco, "Tesco annual report," https://www.tescoplc.com/media/474793/tesco_ar_2018.pdf, 2018.

[23] G. O. Karame, E. Androulaki, and S. Capkun, "Double-spending fast payments in bitcoin," in *ACM conference on Computer and communications security (CCS)*, 2012.

[24] T. Hanke, M. Movahedi, and D. Williams, "DFINITY Technology Overview Series, Consensus System," *arXiv preprint arXiv:1805.04548*, 2018.

[25] The Wanchain Community, "Wanchain yellow paper," https://github.com/wanchain/crypto/raw/master/Wanchain%20yellow%20paper%20English%20version.pdf, 2018.

[26] T. N. Community, "NXT Blockchain," https://www.dropbox.com/s/cbuwrorf672c0yy/NxtWhitepaper_v122_rev4.pdf, 2014.

[27] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network." in *USENIX Security Symposium*, 2015.

[28] P. Prihodko, S. Zhigulin, M. Sahno, A. Ostrovskiy, and O. Osuntokun, "Flare: An approach to routing in lightning network," http://bitfury.com/content/5-white-papers-research/whitepaper_flare_an_approach_to_routing_in_lightning_network_7_7_2016.pdf, 2016.

[29] F. Engelmann, H. Kopp, F. Kargl, F. Glaser, and C. Weinhardt, "Towards an economic analysis of routing in payment channel networks," in *ACM Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2017.

[30] R. Khalil and A. Gervais, "Nocust – a non-custodial 2nd-layer financial intermediary," *Gas*, vol. 200, 2018.

[31] R. Khalil, A. Gervais, and G. Felley, "Nocust–a securely scalable commit-chain," 2019.

[32] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, 2002.

[33] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *J. Cryptology*, vol. 17, no. 4, 2004.

[34] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2003.

[35] T. Ristenpart and S. Yilek, "The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks," in *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2007.

[36] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," *IACR Cryptology ePrint Archive*, vol. 2018, p. 483, 2018.

[37] M. Fischlin, "Communication-efficient non-interactive proofs of knowledge with online extractors," in *Annual International Cryptology Conference (CRYPTO)*, 2005.

[38] T. Ristenpart and S. Yilek, "The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2007.

[39] T. Høiland-Jørgensen, B. Ahlgren, P. Hurtig, and A. Brunstrom, "Measuring latency variation in the internet," in *International on Conference on emerging Networking EXperiments and Technologies*, 2016.

[40] M. Fadhil, G. Owen, and M. Adda, "Bitcoin network measurements for simulation validation and parameterisation," in *International Network Conference (INC)*, 2016.

[41] S. Ben Mariem, P. Casas, and B. Donnet, "Vivisecting blockchain p2p networks: Unveiling the bitcoin ip network," in *ACM CoNEXT Student Workshop*, 2018.

[42] Mastercard, "Uk - domestic interchange fees," https://tinyurl.com/ybzkcxak, 2017.

[43] S. Campion, "Transit-oriented displacement?: The san jose flea market and the opportunity costs of smart growth," 2011.

[44] A. Pelham, E. Sills, and G. S. Eisman, *Promoting Health and Wellness in Underserved Communities: Multidisciplinary Perspectives through Service Learning. Service Learning for Civic Engagement Series.* ERIC, 2010.

[45] J. A. List, "The economics of open air markets," National Bureau of Economic Research, Tech. Rep., 2009.

[46] T. T. Project, "Tor metrics," https://metrics.torproject.org/onionperf-latencies.html, 2019.

[47] T. Ruffing, P. Moreno-Sanchez, and A. Kate, "Coinshuffle: Practical decentralized coin mixing for bitcoin," in *European Symposium on Research in Computer Security (ESORICS)*, 2014.

[48] J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten, "Mixcoin: Anonymity for bitcoin with accountable mixes," in *Financial Cryptography and Data Security (FC)*, 2014.

[49] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash protocol specification," Technical report, 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep., 2016.

[50] S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. Vechev, "zkay: Specifying and enforcing data privacy in smart contracts," in *ACM Conference on Computer and Communications Security (CCS*, 2019.

[51] R. Khalil and A. Gervais, "Revive: Rebalancing off-blockchain payment networks," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[52] R. A. E. Khalil and A. Gervais, "System and method for scaling blockchain networks with secure off-chain payment hubs," May 9 2019, uS Patent App. 16/183,709.

[53] D. L. Salamon, G. Simonsson, J. Freeman, and B. J. Fox, "Orchid: Enabling decentralized network formation and probabilistic micropayments," 2018.

[54] R. Pass *et al.*, "Micropayments for decentralized currencies," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[55] J. Kwon, "Tendermint: Consensus without mining," 2014.

[56] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference (CRYPTO)*, 2017.

[57] P. Vasin, "Blackcoin's proof-of-stake protocol v2," 2014.

[58] W. Li, S. Andreina, J.-M. Bohli, and G. Karame, "Securing proof-of-stake blockchain protocols," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2017.

[59] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[60] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[61] P. Li, G. Wang, X. Chen, and W. Xu, "Gosig: Scalable byzantine consensus on adversarial wide area network for blockchains," *arXiv preprint arXiv:1802.01315*, 2018.

[62] E. Research, "Ethereum 2.0 spec–casper and sharding," https://github.

com/ethereum/eth2.0-specs/blob/master/specs/core/0_beacon-chain.md, 2018.

[63] E. Conner, "Transaction throughput under shasper," https://ethresear.ch/t/transaction-throughput-under-shasper/3467, 2018.

[64] V. Buterin, "Slasher: A punitive proof-of-stake algorithm," 2014.

[65] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[66] M. Bellare, J. A. Garay, and T. Rabin, "Fast batch verification for modular exponentiation and digital signatures," in *International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, 1998.

APPENDIX

A. Background on Permissionless Consensus

In this appendix we review additional examples of recent permissionless consensus system. We center our review around schemes that do not introduce significant additional security assumptions, and refer the reader to [7] for a more thorough survey.

Proof of Stake is proposed as an alternative to computational puzzles in Proof of Work, where the nodes commit (i.e., stake) funds in order to participate in the consensus process [55]–[58]. These solutions are based on an economic-driven model with nodes being rewarded for honest behavior and penalized for diverging from the consensus. While they provide some latency improvements, they do not reach the 3 seconds averages of centralized payment processors.

For example, Ouroboros [56] reports a throughput of approximately 300 transactions per second and a block frequency 10-16 times smaller than that of Bitcoin. Thus, a merchant will still have to wait for several minutes before a transaction can be considered finalized. As another example of Proof-of-Stake system, Algorand [59] has a throughput that is 125× higher than that of Bitcoin, and latency of at least 22 seconds, assuming a network with no malicious users.

Sharding. While Proof of Stake techniques involve the whole network in the consensus process, sharding techniques promise significant performance improvement by splitting the network in smaller groups. For example, Elastico [60] achieves four times larger throughput per epoch for network sizes similar to that of Bitcoin. However, it provides no improvement on the confirmation latency (~ 800 seconds). Similarly, Gosig [61] can sustain approximately 4.000 transactions per second with an $\sim$1 minute confirmation time. Rapidchain [20] has a throughput of $7,300$ transactions per second with a 70-second confirmation latency.

Omniledger [19] is the only proposal that reports performance (both throughput and latency) compatible with retail payments. However, this comes at a security cost. Their low-latency transactions (i.e., Trust-but-Verify) use shards comprised of only a few (or even one) "validators". In the retail setting, this enables malicious validators to launch multi-spending attacks by approving several conflicting payments towards various honest merchants. While the attack and the malicious validator(s) will be uncovered by the core validators within $\sim$1 minute, this time-period suffices for an adversary to attack multiple merchants resulting in substantial losses.

Shasper for Ethereum is expected to be capable of handling $13,000$ transactions per second, while optimistic estimates report a minumum block frequency of $\sim$8 seconds [62]–[65]. However, even with such a small block interval, the latency remains too high for retail purchases, as merchants will need to wait for several blocks for transactions to eventually reach finality.

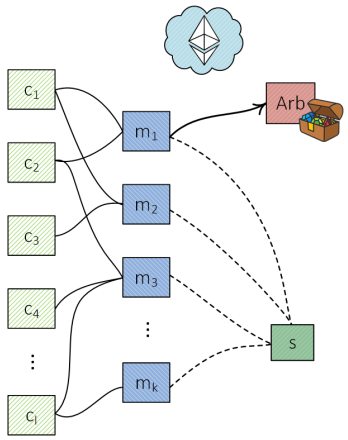

Fig. 5: **Centralized Snappy instance.** The customers c_i initiate payments towards the merchants m_j, who then consult with the central statekeeper s and either accept or reject the payment.

B. Deployment Alternatives for Snappy

In this appendix we discuss alternative deployment options.

Centralized Snappy. While decentralization is one of the main benefits of Snappy, there may be cases where having one central party is also acceptable. In this case, the merchants can appoint a single party to approve or reject payments. This simplifies our protocols as each payment requires only one approval query instead of several. However, if the central party is untrusted, it still has to deposit a collateral for the merchants to claim in case it equivocates. However, relying on a single party comes with drawbacks. In particular, such a setup adds a centralized layer on top of a fully decentralized blockchain, while the reliance on a single service provider will likely result in increased service fees.

Non-statekeeping Merchants In the previous section, we discussed a fully centralized version of Snappy that allows the system to scale even further and simplifies statekeeping. While this setup has several advantages, the liveness and quality of service of the deployment relies on the single party that can unilaterally decide on changing the service fees and processes. An alternative solution that retains most of the centralization benefits while remaining decentralized (but not fully) is allowing non-statekeeping merchants. Merchants can join the system and decide if they want to deposit a collateral and perform statekeeping tasks or they simply want to be able to receive payments.

This allows merchants who choose not to keep state to still make use of the Snappy deployment and enables the system to scale to millions of merchants. To incentivize statekeeping, a small service fee could be paid by non-statekeeping merchants. to those who have allocated a statekeeping collateral. While this goes beyond the scope of this paper, it is probably preferable if the statekeepers' set remains open to all interested

merchants (cf. to being capped or fixed) and the service fees are determined dynamically based on the offer/demand. Note that several merchants may be represented by a single statekeeping node. For example, instead of having small shops match the collateral of large chain stores, their association could maintain one node that performs the statekeeping for all of them and routes their to-be-approved payments to the rest of the statekeepers.

This setup has the advantage of being able to use any trust relationships (e.g., small merchants trust their association) when/where they exist, while still allowing a trustless setup for actors who prefer it.

One pending transaction. Much of the complexity of Snappy's protocols comes from the fact that the pending transactions of a customer should never exceed in value the collateral. One possible way to reduce this complexity is by constraining the number of allowed pending transactions to 1. Such a setup allows the customers to conduct at most one transaction per block, and greatly simplifies the settlement process, as there are no pending transaction to be provided by the merchant. We believe that such a setup is realistic and may be preferable in cases where the customers are unlikely to perform several transactions within a short period of time. However, this is an additional assumption that may reduce the utility of the system in some cases. For example, a customer, who after checking out realizes that they forgot to buy an item, will have to wait until the pending transaction is confirmed.

Signature verification batching. While the computational cost of verifying an aggregated signature (i.e., two pairings) is negligible for a personal computer, this is not true for the Ethereum Virtual Machine, where a pairing operation is considerably more expensive than a group operation. Our original scheme tackles this cost by having the arbiter verify signatures only in case of disputes. As an additional cost-reduction optimization, the arbiter can use techniques such as those in [66] to batch and check several signatures simultaneously

Let's assume that there are ℓ aggregated signatures $(\sigma_1, \ldots, \sigma_\ell)$ to be verified for the messages $(m_1, \ldots, m_\ell)$. The arbiter samples ℓ random field elements $(\gamma_1, \ldots, \gamma_\ell)$ from $\mathbb{Z}_p$. The verifier considers all the signatures to be valid if

$$e\left(\prod_{i=1}^{\ell} \sigma_i^{\gamma_i}, h\right) = \prod_{i=1}^{\ell} e\left(H(m_i)^{\gamma_i}, \prod_{j=1,\tau_{q_i}[j]=1}^{n} v_j\right).$$

This roughly halfs the verification costs. In systems where the number of transactions is considerably more than the number of statekeepers, we can reduce the costs per transaction further. Assume that there are ℓ aggregated signatures $(\sigma_1, \ldots, \sigma_\ell)$ to be verified for the messages $(m_1, \ldots, m_\ell)$ where $\ell \gg n$. The verifier samples ℓ random field elements $(\gamma_1, \ldots, \gamma_\ell)$ from $\mathbb{Z}_p$. The verifier considers all the signatures to be valid if

$$e\left(\prod_{i=1}^{\ell} \sigma_i^{\gamma_i}, h\right) = \prod_{j=1}^{n} e\left(\prod_{i=1,\tau_{q_i}[j]=1}^{\ell} H(m_i)^{\gamma_i}, v_j\right).$$

The cost of verifying a batch of ℓ signatures signed by n merchants is then $n+1$ pairing operations and 2ℓ group exponentiations in $\mathbb{G}$.

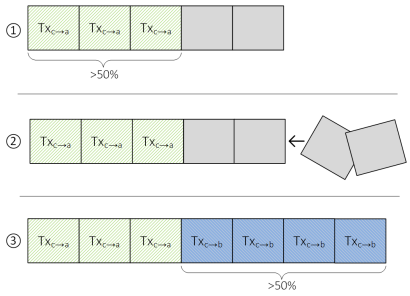

Fig. 6: **Moving majority attack** enables a customer to have two transactions with the same index value approved by disjoint statekeeper majorities.

Dynamic Statekeepers' Consortia. So far we have considered only cases where customers joined and withdrew from the system. Similarly, one can imagine that statekeepers could decide to leave or new statekeepers may want to join an existing deployment. Such functionality can be easily facilitated by modifying through the registration and de-registration algorithms available for customers. However, while churning customers do not pose a threat to the security of the system, changes in the set of statekeepers may result in attacks.

Such an attack could enable a malicious customer to have two approved transactions with the same index value. As shown in Figure 6, initially the system features 5 statekeepers $(s_1 \ldots s_5)$ and a merchant who wants to have a transaction τ approved, reaches out to s_1, s_2 and s_3. Subsequently, two new statekeepers s_6 and s_7 join the system. The malicious customer now issues another transaction τ', such that $\tau_i = \tau'_i$. The merchant receiving τ' now queries a majority of the statekeepers (i.e., s_4, s_5, s_6 and s_7) and gets the transaction approved. At the final stage of the attack c issues another transaction that invalidates τ and τ' (e.g., a doublespend), while the malicious merchant quickly claims τ'_v from the customer's collateral. Because of the way the arbiter processes claims (Line 12 in Algorithm 3), the honest merchant is now unable to claim τ_v from $\mathscr{col}_c$. Moreover, none of the statekeepers equivocated and thus no funds can be recouped from their collateral.

Snappy can safely support a dynamically changing set of statekeepers, if appropriate protection mechanisms are deployed. For example, such attacks can be prevented: 1) by giving early notice to the merchants about changes in the set (e.g., multistage registration), and 2) by waiting (a few minutes) until all past transactions are finalized in the blockchain before/after every change.