Censorship-Resilient and Confidential Collateralized Second-Layer Payments

Kari Kostiainen[1], Sven Gnap[1], and Ghassan Karame[2]

[1] ETH Zurich
kari.kostiainen@inf.ethz.ch, gnaps@student.ethz.ch
[2] Ruhr University Bochum
ghassan.karame@rub.de

Abstract. Permissionless blockchains are too slow for applications like point-of-sale payments. While several techniques have been proposed to speed up blockchain payments, none of them are satisfactory for application scenarios like retail shopping. In particular, existing solutions like payment channels require users to lock up significant funds and schemes based on pre-defined validators enable easy transaction censoring. In this paper, we develop Quicksilver, the first blockchain payment scheme that works with practical collaterals and is fast, censorship-resilient, and confidential at the same time. We implement Quicksilver for EVM-compatible chains and show that censoring-resilient payments are fast and affordable on currently popular blockchains platforms like Ethereum and Polygon.

1 Introduction

Although blockchain technology has gained wide attention, its adoption as a payment mechanism is still limited. For instance, in point-of-sale payments, permissionless blockchains are simply too slow, as safe blockchain payment acceptance takes several minutes and transactions should be completed in few seconds. Permissionless blockchains are also too transparent for retail payments. This has led to the surge of dedicated platforms like Zcash [18] and Monero [20] and 2nd-Layer systems like Zether [7] to provide improved privacy. However, none of these solutions are fast enough for retail payments.

To speed up permissionless blockchain payments, a number of techniques have been developed. Payment channels [14] and payment channel networks [14] emerge as one of the most popular solutions. Once a payment channel is established, or once a suitable route in a payment channel network is found, fast off-chain payments are possible. However, if a retail customer needs to establish separate channels with all possible merchants, he needs to lock up significant collateral. Payment networks can reduce the amount of locked collateral, but cannot guarantee that a suitable route is found and that the payment can be completed.

Side chains [19] and new collateralized payment processing techniques like Snappy [15] provide an alternative approach to speed up payments. In such

solutions, pre-defined validators approve payments off-chain. Such schemes improve latency, but suffer from a significant limitation as a by-product: Since every payment now needs to be approved by a set of validators, payment censoring becomes possible. Strong resilience to censorship is one of the primary advantages of permissionless blockchains, and known practical schemes for fast payments unfortunately eliminate that advantage.

Research question and solution. In this paper, we address the following research question: *Can one design a permissionless blockchain payment scheme that is practical, fast, censorship-resilient, and confidential at the same time?* To the best of our knowledge, no such solution exists at the moment.

We present a novel solution, dubbed Quicksilver, that answers this question positively. Quicksilver leverages two building blocks from the existing research literature. First, Quicksilver adapts the notion of fast and collateralized payment processing from Snappy [15]. Second, we use a confidential payment technique from Zether [7]. However, as we show in this paper, a simple composition of these two techniques is insufficient to solve our problem and therefore we realize Quicksilver with new ideas: First, we enable censorship-resilient payment approval by modifying payment processing such that payment details are made oblivious to the validators using cryptographic commitments and Verifiable Random Functions (VRFs). Second, we enable fast processing of confidential payment by leveraging the homomorphic property of encrypted payment values and new authorization model for confidential payments. By combining such new techniques, we build Quicksilver, the first 2nd-Layer blockchain payment scheme that is fast, censorship-resilient and confidential.

We implemented Quicksilver for EVM-compatible chains and evaluated it on Ethereum and Polygon. Our experiments show that Quicksilver payments can be approved in 2.5 seconds which is comparable to Visa [11]. One payment costs $0.3 on Polygon. Quicksilver payments that are fast and censoring-resilient (but not confidential) cost $1.9 on Ethereum. While such prices are volatile, they demonstrate practicality of our (non-optimized) solution. We also show that Quicksilver can process thousands of transactions per second and thus match the throughput of current (and near future) permissionless blockchains.

Contributions. To summarize, in this paper we make the following contributions:

- *Quicksilver system:* We design and implement the first blockchain payment scheme that is fast, censorship-resilient, and confidential (cf. Sections 4 and 5).
- *Performance evaluation:* We show experimentally that Quicksilver payments can be both fast and affordable. (cf. Section 6).
- *Security analysis:* We prove that Quicksilver payments are safe and prevent targeted censoring (cf. Appendix A).

	Key requirements					*Additional requirements*			
System	**Payment Latency**	**Censoring Resilience**	**Confiden-tiality**	**Practical Collaterals**	**No Extra Assumpt.**	**No Fund Migration**	**Transaction Fees**	**Anony-mity**	**System Throughput**
Permissionless chain	high	✓	✗	n/a	✓	✓	low	✗	limited
Sidechains [19]	low	✗	✗	n/a	✗	✓	low	✗	increased
Payment channels [14]	low	✓	✓	✗	✓	✓	low	✗	increased
Payment hubs	low	✓	✗	✗	✓	✓	low	✗	increased
Snappy [15]	low	✗	✗	✓	✓	✓	normal	✗	limited
Zether [7]	high	✓	✓	n/a	✓	✓	moderate	✓	limited
LDSP [16]	low	partial	✗	✗	✓	✓	low	partial	limited
Monero [20]	high	✓	✓	n/a	✓	✗	normal	✓	limited
Zcash [18]	high	✓	✓	n/a	✓	✗	normal	✓	limited
Quicksilver	low	✓	✓	✓	✓	✓	moderate	✗	limited

Table 1. Comparison of Quicksilver to related solutions.

2 Known Solutions and Their Limitations

Blockchains are too slow and transparent for point-of-sale payments. Below we analyse known solutions that speed up blockchain payments and improve their privacy.

Side chains. One common approach is side chains [19], where a pre-defined set of validators approve each payment, e.g., by running a consensus protocol. However, the main drawback of this approach is that it requires additional trust assumptions. In the case of BFT consensus, two-thirds of the validator nodes must be trusted. Additionally, such systems are not resilient to censoring. If the validator nodes so decide, they can easily prevent transaction processing from targeted victim users.

Payment channels and networks. Another popular set of solutions are payment channels [14]. In a typical payment channel scheme, two parties lock collateral into a smart contract and then perform fast off-chain transactions that are secured, as long as there is sufficient collateral left in the channel. Payment channels improve latency. However, when considering retail payments, the main problem of payment channels is that the user needs to setup a separate channel with each merchant which requires significant locked-in funds. Payment channels can be organized into networks, but since the flow of funds is predominantly one way (from customers to merchants), the deposited collaterals will quickly run out making it difficult to find available channels to merchants in practice [10].

Payment hubs. Another approach is that all users establish a payment channel with a centralized service, called *payment hub*, that in turn has channels with recipients. In this approach, the users need to deposit only a single collateral. However, the main drawback of such systems is that the payment hub operator would need to deposit a huge collateral that covers all payments of all users in the system. In addition, the collaterals will quickly run out. Moreover, the hub operator is in a perfect position to censor selected users.

Collateralized schemes. Recently, researchers have proposed alternative collateralized schemes such as Snappy [15] that combines ideas from payment channels and side chains. A major benefit of this approach is that users need to deposit only a single collateral and the same collateral can be re-used unlimited number of times which improves collateral practicality. Also validator collaterals remain moderate and validators do not need to be trusted. The main remaining

limitation of Snappy is that the pre-defined validators (called *statekeepers*) can censor users and merchants.

Private payments. Dedicated privacy-preserving blockchains like Monero [20] and Zcash [18] hide user identities and payment amounts. However, such schemes do not improve latency. In addition, since payment privacy requires the use of a separate chain, users need to migrate funds between multiple systems. 2nd-Layer solutions like Zether [7] enable confidential payment amounts on popular blockchains without migration. Such schemes are censoring resilient but do not improve payment latency.

3 Problem Statement and Security Goals

Based on the above discusssion, which is summarized in Table 1, we make two observations. Our first observation is that many 2nd-Layer schemes that provide improved payment latency make a significant trade-off. These systems introduce validators (or *statekeepers*) whose job is to approve payments fast. The negative side effect of this approach is that these privileged entities can easily censor customers or merchants. Thus, such solutions gain payment speed by sacrificing strong censorship-resilience of blockchains. Our second observation is that while several private blockchain payment techniques have been proposed, it is unknown whether such payments can be made fast and scalable for applications like retail payments. Motivated by these observations, our research question becomes:

Research question: *Is it possible to design a payment scheme that retains the strong censorship resilience of permissionless blockchains and provides fast and confidential payments with practical collaterals for application scenarios like retail shopping?*

To the best of our knowledge, no such solution exists at the moment. The work that probably comes closest to meet these requirements is a recent proposal called LDSP [16] that runs a Chaum-style e-cash scheme [9] as 2nd-Layer on top of Ethereum. However, LDSP provides only partial censoring resilience because payment recipient (merchant) identity and payment amount are not hidden. Also its collaterals are not practical since they need to be replenished after they have been used.[3]

Main security definitions. Before introducing our solution, we define the main security notions that we seek to achieve. We assume that customers and merchants initiating a payment do not disclose any information about each other or their payment to the public. We argue that little can be done if a customer

[3] In LDSP [16], payment validators do not learn the identity of the customer at the time of payment, and therefore cannot censor customers. However, they do learn the identity of the merchant who deposits the coins back to a validator that can censor merchants. Payment validators also learn the total value of deposited coins, and thus price-based censoring is possible. Collaterals are not reusable, because once the withdrawn coins have been used, no further payments are possible.

or merchant leaks the payment. When a customer initiates a payment, it sends a *payment intent* to the merchant. Payment intents are then processed by validators (statekeepers) before the merchant accepts the payment. Our goal is to prevent payment censoring by validators based on the payment intent.

Let P denote the set of all payment intents in the system. By $t \in P$, we mean that a payment intent t is characterized by a sender address in $\{0,1\}^n$ drawn from probability distribution S, a recipient address in $\{0,1\}^m$ drawn from probability distribution R, a payment value in $\{0,1\}^l$ drawn from probability distribution V, and a payment index (introduced in the next section) in $\{0,1\}^o$ drawn from probability distribution I. Similarly, let U denote the set of all payment intents drawn at random.

Definition 1 (Censorship resilience). *We say that a payment system is censorship-resilient if P is poly-time indistinguishable from a randomly formed payment intent in U, if $\forall$ p.p.t. distinguisher A, there exists a negligible function ϵ, s.t. $|\underset{t \in P}{Pr}(A(t)=1) - \underset{t \in U}{Pr}(A(t)=1)| < \epsilon(n)$ where n is the security parameter.*

Definition 2 (Merchant safety). *If a merchant follows the protocol and accepts a payment, he is guaranteed to receive funds matching the full amount of the accepted payment.*

Functional & non-goals. In this work, we focus on improving payment latency. Increased throughput is another common goal for 2nd-Layer solutions, but for us out of scope. However, in Section 6, we show that our solution can match the throughput of permissionless blockchains.

We explain how confidential payments can be made fast and censoring-resilient. While payment anonymity is not necessarily the focus of our work, in Appendix B, we discuss how our solution could be extended for anonymous payments in future work.

4 Quicksilver Overview

In this section, we describe two techniques that we adopt as building blocks for our solution. Then, we explain how a simple combination of these techniques fails to solve our research problem and outline our main ideas for overcoming the involved challenges.

4.1 Building Blocks

Fast payment approval. The first building block of our solution is fast payment approval using collateralized majority signing, a technique recently introduced by Snappy [15]. This technique requires a pre-defined set of validators, called *statekeepers*, that approve payments fast. During system registration, each user and statekeeper deposits a collateral into a smart contract. In a practical deployment, merchants can play the role of statekeepers.

Figure 1 illustrates the payment processing. When the user initiates a payment (step 1), it sends a data structure called *payment intent* and a list of previously approved pending transactions to the merchant. Payment intents contain the addresses of the payment sender (customer) and recipient (merchant), the amount of the payment, and a monotonically increasing *payment index* that the user increments for each payment. Using the payment index, the merchant verifies that all the previous payments by the same user either appear finalized on the chain or are approved by statekeepers, and that the user has enough collateral to cover the current and pending payments (step 2).

The merchant forwards the payment intent to all statekeepers who check that they have not approved a payment by the same user with the same index before. Then they sign the intent and send it back to the merchant (step 3). Once the merchant has collected signed intents from the *majority* of the statekeepers (step 4), it can aggregate the signatures, and forward the aggregated signature back to the user (step 5). The user will finalize the transaction by including the majority-signed intent to it (step 6). The user sends the final transaction to the merchant who can at this point consider the payment safely completed, hand over goods to the user (step 7), and broadcast the transaction to the blockchain network (step 8).

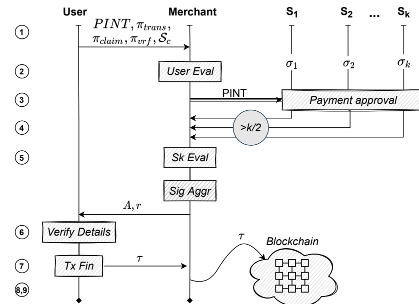

Fig. 1. Payment process in Quicksilver.

The main intuition why such payment approval process is safe is based on two arguments [15]. First, if the user double spends, the merchant can claim back the lost funds from the user's collateral by presenting the signed intent as evidence to the smart contract that controls all collaterals. This process is called *settlement*. Second, if the user colludes with a malicious statekeeper who approves multiple transactions for the same user with the same index value (double spending with another merchant), the victim merchant can claim the lost funds from the cheating statekeeper's collateral. Identification of the cheating statekeeper is always possible, because if two separate majority sets sign conflicting transactions, there will always be at least one statekeeper whose equivocation leaves undeniable evidence.

Confidential payments. The second building block of our solution is a confidential payment mechanism that leverages encrypted account balances, as recently introduced by Zether [7]. In this technique, a smart contract maintains an encrypted account balance for each user. Each account balance is encrypted with a public key that is associated with the account, and the encryption scheme is homomorphic (ElGamal encryption with the message in the exponent) such that it allows addition and subtraction of encrypted values.

To create a confidential balance, a user performs a funding operation that transfers coins to a smart contract which will create an encrypted account bal-

ance associated with that user's public key. To transfer funds, the sender first encrypts the payment amount using the public key of its own account. The resulting ciphertext can be subtracted by the smart contract from the sender's previous encrypted account balance. Then, the sender encrypts the payment amount using the public key of the recipient's account. This ciphertext can be added to the recipient's previous encrypted balance by the smart contract. The sender also produces a zero-knowledge proof that shows that the two account balances were adjusted by the same amount, the sender's updated balance is positive, and proves the knowledge of the sender's private key. The smart contract verifies the proof and updates the encrypted account balances accordingly.

4.2 Challenges and Main Ideas

While the above two techniques provide a starting point for our solution, a strawman combination of these two techniques fails to solve our research question. Below, we explained involved challenges and outline our new ideas and techniques for solving them.

Challenge 1: Merchant censoring. Recall that the majority-signed payment intent must contain the the merchant's address, so that only the correct merchant can claim settlement in case of double-spending. Because merchant's address is included to the intent, statekeepers can censor payments of chosen merchant. To prevent such censoring, the merchant's identity needs to be hidden from the statekeepers *at the time of payment approval* and the hiding mechanism must be such that the legitimate merchant can later prove its identity to the smart contract, if settlement is needed.

We prevent merchant censoring using *cryptographic commitments*. The merchant modifies the intent that it receives from the user by replacing its address with a commitment to the address (and saving the commitment opening). In case the transaction turns out to be malicious, the merchant can open the commitment, and thus prove its identity to the smart contract for correct settlement. We focus on hiding identities at the protocol level and assume that network-level anonymity can be achieved through other means.[4]

Challenge 2: Customer censoring. Fast payment approval requires that the identity of the customer and the monotonically increasing payment index are included in each payment intent. The statekeepers ensure that they sign only one intent with the same index per customer. Such a design allows statekeepers to censor payments from the chosen customer. Replacing the user's identity with a cryptographic commitment is insufficient, since this would prevent statekeepers from controlling that they sign only one intent for each payment index for each user. Instead, what is needed is a mechanism to hide the identity of the user

[4] One option is to rely on Tor between the merchant and the statekeepers. Another option is to modify the payment processing flow such that the customer, whose IP address is expected to change, can send the intent to the statekeepers on behalf of the merchant.

from the statekeepers such that they can still enforce the policy of one signature per index and user.

We solve this problem using *verifiable random functions* (VRFs) [13]. When a customer wishes to initiate a payment, he uses his VRF private key and the current payment index value as input for the VRF that will output a pseudo-random value and a proof that can be verified using the associated VRF public key. The customer creates a modified payment intent that contains the pseudo-random value as *Randomized Payment Identifier* (RPID) instead of its identity. The customer passes the payment index and VRF proof for the current and all pending transactions to the merchant who can verify that the proofs are correct using the public key that is registered to the smart contract, before passing the intent to the statekeepers. The statekeepers enforce a new policy where they sign only one intent with the same RPID value. Such payments are safe for merchants due to the *uniqueness* property of VRFs. For the same VRF input (payment index) only one correct VRF output (RPID) can be generated. Since malicious customers cannot create multiple RPIDs for the same index and statekeepers track double-spending per payment index, a malicious customer cannot double spend. Customer censoring is no longer possible due to the *pseudorandomness* of VRFs which ensures that RPID (and thus the payment intent) reveals no information about the customer.

Challenge 3: Price-based censoring. Prior research has shown that payment amounts can identify the merchant [12]. Thus, to effectively prevent censoring, we also need to hide the amount of the payment from the statekeepers. We hide payment amounts from the statekeepers using encrypted payment amounts in the payment intent.

Challenge 4: Confidential collaterals. Next, we consider deployments where also the user's collateral is confidential for increased privacy. Recall that, to safely accept a payment, the merchant needs to verify that the customer's collateral is sufficient to cover both the current payment and all the pending transactions. When the collateral is confidential and all payments amounts are encrypted, the merchant cannot perform such a check. Additionally, if the customer cheats and the merchant needs to initiate a settlement, it cannot authorize a transfer from the user's confidential collateral, because it does not know the user's private key needed to create a proof that authorizes the confidential transfer.

We solve these two problems using a new *payment authorization mechanism* that is tailored to our use case. When the customer initiates a payment, it creates *two* proofs: one that authorizes the transfer of funds from his account to the merchant's account and another that authorizes transfer of funds from his collateral to the merchant. When computing the second proof, the customer leverages the *homomorphic property* of confidential payments. The customer subtracts the (encrypted) values of all pending payments from the (encrypted) value of the collateral, and then creates the proof that shows that shows that the updated collateral value is still positive. This is possible using existing proof techniques from [7]. By verifying the proof, the merchant ensures that the customer's collateral is sufficient to cover the all the pending and current payments.

Challenge 5: Confidential statekeeper collateral. To enable confidential statekeeper collaterals, statekeepers establish separate confidential collaterals for each merchant and privately shares their value and the private key that controls them with the respective merchant. Merchants *track payments* approved by each statekeeper and control that no statekeeper is approving more transactions than their collateral allows. If settlement from a statekeeper is needed, merchant can authorize the transfer from the statekeepers collateral using the private key. Sharing the private key is safe, because the Quicksilver smart contract enforces that the statekeeper's collateral is used only when the statekeeper equivocated.

5 Quicksilver Specification

In this section, we describe Quicksilver in detail. We start by listing cryptographic primitives. After that, we explain the system initialization, payment, and settlement operations.

5.1 Cryptographic Primitives

Aggregate signatures. To enable signature aggregation for reduced transaction size and efficient verification, we rely on the Boneh-Lynn-Shacham (BLS) signature scheme [6]. We assume that the user is given functions $\sigma = \mathsf{AggrSig}(\{\sigma_1, \dots, \sigma_n\})$ and $y = \mathsf{AggrPk}(\{y_1, \dots, y_n\})$ which implement signature and public key aggregation, respectively, and a verification function $\mathsf{AggrVerify}(\sigma, y)$ that outputs valid for correct signature [5].

Verifiable Random Functions. A verifiable random function (VRF) is a public-key version of a keyed cryptographic hash [13]. Given an input value m, the owner of private key x can compute hash $h = \mathsf{VRFhash}(x, m)$ and matching proof $\pi_{\mathsf{vrf}} = \mathsf{VRFprove}(x, m)$. An important property of VRFs is that the hashing algorithm is deterministic for the same inputs (x, m). Given y, m, and π, the hash is valid if $\mathsf{VRFverify}(y, m, \pi)$ outputs valid. Anyone can deterministically obtain the VRF output h from the proof π by computing $h = \mathsf{VRFproof2hash}(\pi)$.

VRFs have the following security properties [13]. *Uniqueness* means that, for any fixed public VRF key y and for any input m, there is a unique VRF output h that can be proved to be valid. *Collision resistance* is the same as for cryptographic hash functions. *Pseudorandomness* ensures that when an adversary sees a VRF hash output h without its corresponding VRF proof π, then h is indistinguishable from a random value.

Commitments. Cryptographic commitments allow a user to commit to a value that remains hidden and reveal it later. Commitment c to message m can be created using function $c = \mathsf{Commit}(m, r)$ where r is a randomly chosen blinding factor. Commitment c can later be opened to m using the blinding factor r as input: $m = \mathsf{Open}(c, r)$. Pedersen commitments are perfectly hiding and computationally binding [17].

ElGamal encryption. We leverage an ElGamal encryption scheme variant where the message is in the exponent, as defined in [7]. Given a key-pair (x, y),

where $y = g^x$, we encrypt a message b by choosing a random secret $r \in \mathbb{Z}_p$ and by computing the ciphertext $C = (g^b y^r, g^r)$. To decrypt C, one divides $g^b y^r$ by $(g^r)^x$ which yields g^b. The extraction of b out of g^b is performed by brute force. Such encryption scheme is *additively homomorphic* under the same public key. We use notation where $C_a \leftarrow C_a \circ C_b$ adds the value of C_b to C_a. Conversely, $C_a \leftarrow C_a \circ C_b^{-1}$ deducts the value of C_b from C_a.

Σ-Bullets. Σ-Bullets is a proof system that combines efficient range proofs from Bulletproofs [8] with Sigma protocols for algebraically encoded statements. Σ-Bullets is used for confidential payments in Zether [7] as follows. Assume that the user wants to transfer an amount b^* from his account y to another account $\overline{y}$. Let C_b be the current encrypted balance associated with y. To complete such a confidential transfer, the smart contract needs to deduct b^* from y's balance and add the same amount to $\overline{y}$'s balance. To achieve this, the user will encrypt b^* under both y and $\overline{y}$ to get ciphertexts C and $\overline{C}$. After that, the user computes a zero-knowledge proof using function $\pi_{\mathsf{transfer}} = \mathsf{ProveTransfer}(C_b, C, \overline{C}, y, \overline{y}; x, b^*, b', r^*)$ that takes as inputs both public keys $(y, \overline{y})$, all three ciphertexts $(C_b, C, \overline{C})$, the sender's private key x, the payment amount b^*, the sender's remaining balance b' after the account update, and randomness used for encryption r^*. The function outputs a proof π_{transfer}, which shows that (1) ciphertexts C and $\overline{C}$ are well formed and encrypt the same amount, (2) the payment amount b^* is a positive value, (3) the sender's remaining balance b' is positive, and (4) the proof creator knows the private key x. Finally, we assume a verification function $\mathsf{true}/\mathsf{false} = \mathsf{VerifyTransfer}(y, \overline{y}, C_b, C, \overline{C}, \pi_{\mathsf{trans}})$ that takes as input the public keys, the above ciphertexts, and the proof.

5.2 System Initialization

We assume that Quicksilver smart contract is deployed on blockchain like Ethereum. All system participants (customers, merchants, statekeepers) have an existing encrypted account balance, maintained by the Quicksilver contract, that has already been funded. We call such confidential accounts *Quicksilver accounts* to differentiate them from plaintext Ethereum accounts. We assume a system deployment with C registered customers, M merchants, and S statekeepers. The state of the Quicksilver smart contract is shown in Tables 2, 3, and 4. We denote the privacy-preserving payment intent as PINT and the complete transaction as τ. The structure of τ is defined in Table 5.

Customer registration. To register, customer c creates a VRF key pair $(y_{\mathsf{vrf}}, x_{\mathsf{vrf}})$ and registers the public key y_{vrf} with their existing Quicksilver account y_c in the smart contract. The customer also performs a confidential transfer [7] that transfers collateral amount $pcol$ from y_c to the Quicksilver smart contract. The contract creates a new user entry $C[y_c]$ in its state and updates the user's confidential account balance $Acc[y_c].bal$ and the user's confidential collateral $C[y_c].pcol$ based on the confidential payment. The user initializes locally its payment index as $i = 0$.

Field	Symbol	Description
Accounts	Acc	
↪ *entry*	$Acc[y]$	
↪ *Counter*	$Acc[y].ctr$	Transaction counter
↪ *Balance*	$Acc[y].bal$	Encrypted balance
↪ *Pending*	$Acc[y].P$	Accumulated transfers

Table 2. Accounts state in smart contract.

Field	Symbol	Description
Merchants	M	
↪ *entry*	$M[y_m]$	
Statekeepers	S	
↪ *entry*	$S[y_s]$	
↪ *Allocation*	$S[y_s].y[y_m]$	Per merchant collateral

Table 3. Merchants state in smart contract.

Field	Symbol	Description
Customers	C	
↪ *entry*	$C[y_c]$	
↪ *Collateral*	$C[y_c].pcol$	Private collateral
↪ *VRF*	$C[y_c].y_{vrf}$	VRF public key
↪ *Finalized*	$C[y_c].D$	Finalized transactions
↪ *entry*	$C[y_c].D[i]$	Entry for index i
↪ *Hash*	$C[y_c].D[i].h$	Processed tx hash
↪ *Signatures*	$C[y_c].D[i].\tau_A$	Aggregate signature
↪ *Quorum*	$C[y_c].D[i].\tau_q$	Approving parties
↪ *Bit*	$C[y_c].D[i].b$	Sig. verified flag
↪ *Observed*	$C[y_c].O$	Observed approval quora
↪ *entry*	$C[y_c].O[i]$	Entry for index i
↪ *Hash*	$C[y_c].O[i].h$	Observed tx hash
↪ *Trace*	$C[y_c].T$	Past settlements
↪ *entry*	$C[y_c].T[i]$	Entry for index i
↪ *Nonce*	$C[y_c].T[i].idx$	Settled tx index
↪ *Remaining*	$C[y_c].T[i].bal$	Remaining collateral

Table 4. Customers state in smart contract.

Field	Symbol	Description
To	τ_{to}	Quicksilver contract
From	τ_f	Any address
Value	τ_v	Transaction fee
ECDSA Sig.	v, r, s	Tx signature triplet
Data		
¡ ↪ *Payment Index*	τ_i	Monotonic counter
↪ *Random Payment ID*	τ_{RPID}	VRF hash
↪ *Commitment*	τ_c	Merchant's address
↪ *Signatures*	τ_A	Aggregate signature
↪ *Quorum*	τ_q	Approving parties
↪ *Proof*	π_{trans}	Customer's Σ-Bullet
↪ *Accounts*	(y_c, y_m)	Sender/receiver address
↪ *Values*	(c_c^*, c_m^*)	Encrypted values

Table 5. Quicksilver transaction τ format.

Statekeeper registration. To register, statekeeper s must create a separate confidential collaterals accounts for each M merchants in the Quicksilver contract. To achieve this, the statekeeper picks M key pairs $((x_1, y_1), ..., (x_M, y_M))$ and sends the public keys $(y_1, ..., y_M)$ to the Quicksilver smart contract. Then, the statekeeper performs M confidential payments [7] from its account y_s to confidential accounts defined by $(y_1, ..., y_M)$. The contract saves the transferred funds in its state as new encrypted account balances $S[y_s].y[y_i]$ and updates the confidential account balance $Acc[y_s].bal$ of the statekeeper based on the payments accordingly. Finally, the statekeeper sends the private keys $(x_1, ..., x_M)$ to the respective merchants.

Merchant registration. To register, merchant m sends its user account public key y_m to the smart contract that creates a new entry $M[y_m]$ in its state. The merchant also stores all the private collateral keys $(x_i, ..., x_S)$ that it receives from each S statekeepers.

5.3 Payment Protocol

The payment process transfers funds from a customer's Quicksilver account y_c to a merchant's Quicksilver account y_m as shown in Figure 1.

Step 1. The customer creates a private payment intent PINT_c which includes (1) a random payment identifier $\mathsf{RPID} = \mathsf{VRFhash}(x_{\mathsf{vrf}}, i)$, computed using the VRF private key x_{vrf} and payment index i, (2) the address of the merchant's account y_m, and (3) the encrypted payment value c_c^* under the customer's public key y_c. The customer also creates two zero-knowledge proofs for payment and

settlement π_{trans} and π_{claim} using the $\mathsf{ProveTransfer}$ function. The first proof π_{trans} proves the usual transfer details, i.e., it is created on the user's main account balance $Acc[c].bal$ and shows that remaining balance after subtracting the encrypted payment amount remains positive. The second proof, π_{claim} is created on customer's the collateral balance $C[y_c].pcol$, and it shows that the current collateral is sufficient to cover the current payment and all the pending payments. To achieve this, the input b' for the π_{claim} proof computation is obtained by homomorphically deducting the encrypted values of all pending payments from the current collateral balance. The user sends to the merchant the current payment index i, PINT_c, y_{vrf}, (c_c^*, c_m^*) $\pi_{\mathsf{trans}}, \pi_{\mathsf{claim}}$, the VRF proofs π_{vrf} for all pending transactions, and the list $\mathcal{S}_c$ of pending transactions.

Step 2. The merchant performs the following checks to ensure that they can claim settlement in case the payment should fail. They verify the VRF proof π_{vrf} for every pending transaction index. Furthermore, for each index $j \in \{1, \ldots, \mathsf{PINT}_c[i]\}$, there must appear an *approved* transaction with index j such that is either finalized on the blockchain or contained in $\mathcal{S}_c$. The received y_{vrf} must be registered on the Quicksilver contract. All details of the confidential payment, in particular π_{trans}, must be correct. The additional π_{claim} must be verified to ensure that the user's private collateral is sufficient to cover all pending transactions and the current transaction. To verify π_{claim}, the merchant homomorphically deducts all the encrypted amounts b_i^* of all pending payments from the customer's current encrypted collateral value $C[y_c].pcol$ to obtain input b' for the $\mathsf{VerifyTransfer}$ function.

Step 3. To prevent censoring, the merchant hides their address in the payment intent PINT_c by replacing their address with a commitment $c = \mathsf{Commit}(y_m, r)$, where r is a randomly chosen blinding factor and saves r. The merchant broadcasts the modified PINT_c to all the statekeepers.

Step 4. Each statekeeper evaluates the received PINT_c. They check within their local list that they have not already approved a payment with the same RPID. If a matching RPID is found, they notify the merchant. Otherwise, they approve the payment by computing a BLS signature $\sigma_i = \mathsf{Sign}(\mathsf{PINT}_c, x_s)$ and send it back to the merchant. The statekeeper appends the approved RPID value to their local list.

Step 5. If a majority of statekeepers approves the payment intent, the merchant checks that all signatures are correct and upon success, aggregates them into $A = \mathsf{AggrSig}(\{\sigma_1, \ldots, \sigma_n\})$. Otherwise, the merchant aborts and informs the user. The merchant additionally checks that each statekeeper who has approved the intent allocated enough collateral to the merchant. This can be done because the merchant knows their corresponding allocated collateral balances and the corresponding secret keys x_i. The merchant stores the payment intent PINT_c together with the blinding factor r, all VRF proofs π_{vrf}, π_{claim} and π_{claim} for possible later settlement. Finally, the merchant sends the aggregated signature A, the blinding factor r, and the majority-signed PINT_c to the customer.

Step 6. The customer verifies the aggregate signature A and checks that the merchant's commitment c in the signed intent opens to $y_m = \mathsf{Open}(c, r)$.

Step 7. The customer creates the final transaction τ containing the details exchanged in the payment process, as described in Table 5. The final transaction τ can be signed by an arbitrary Ethereum account that has sufficient funds to cover the transaction fees.

Step 8. The merchant verifies that the customer correctly constructed and signed τ (i.e., the customer has not replaced, omitted, or modified any of the values). The payment can now be considered safely accepted and the merchant broadcasts τ in the blockchain network.

Step 9. Once τ is included to a block by miners, the Quicksilver smart contract executes *Record-and-Transfer* process (Algorithm 1 in Appendix D) that records the payment in its state and completes the confidential payment to merchant m. This operation verifies that the customer c is registered and that there is no transaction with the same index i already recorded. It then stores the hash of the transaction together with the approval signature and quorum. Finally, the contract verifies the zero-knowledge proof π_{trans} and completes the confidential transfer.

5.4 Settlement

Transaction may not be received and processed by the miners even after a reasonable amount of time due to the following reasons:

1. *Benign congestion:* The transaction may be of lower priority to the miners, e.g. due to having a lower gas price.
2. *Conflicting transaction:* Another transaction by the same user prevents the current transaction to be accepted by the Quicksilver smart contract (e.g., double-spending).
3. *User's blockchain account depletion:* The user's Ethereum account was depleted due to a previous transaction and has insufficient funds to cover the gas fees of the user's pending Quicksilver transactions.
4. *User's Quicksilver account depletion:* The user's Quicksilver account was depleted due to a previous transaction which invalidates the zero-knowledge proof π_{trans}.

In case (1), the transaction is valid but fails to be processed by the blockchain's miners. Hence, the merchant can either wait longer or resubmit the transaction to the blockchain network with a higher gas price (recall that a Quicksilver transaction can be submitted by any Ethereum account). In cases (2-4), the transaction is invalid and the merchant can recover the lost funds by claiming settlement. If there are conflicting transactions in the system (case 2), the merchant must initiate the *Claim-Statekeeper* process (Algorithm 2 in Appendix D) to be refunded from the equivocating statekeeper's collateral. Otherwise, the merchant can initiate the *Claim-Customer* process (Algorithm 3 in Appendix D).

Claim statekeeper. In the following, we denote by τ^p the transaction that is being claimed, and by τ', τ'' majority-approved transactions that conflict either with τ^p or with each other. To create the settlement transaction, the merchant creates a confidential payment proof π'_{trans} using $\mathsf{ProveTransfer}$ function with the inputs being the statekeeper's public key y_m, the merchant's public key y_m, the statekeeper's collateral secret key x_s, the statekeeper's collateral balance b, the payment amount b^*, and ciphertexts (c_s^*, c_m^*) that encrypt the payment amount over the public keys of the statekeeper's collateral account and the merchant's account. Note that this proof is different than the customer's π_{trans}, because it proves the transfer from the statekeeper's collateral to the merchant's account.

The victim merchant sends to the smart contract the pending transaction τ^p, the conflicting transactions τ', τ'', the equivocating statekeeper's public key y_s, the VRF proof π_{vrf} for each above transactions, r matching the pending transaction, the payment amount encrypted under the equivocating statekeepers public key c_s^*, and π'_{trans}. The smart contract executes Algorithm 2 that first verifies that τ^p, τ', τ'' are valid and the merchant provided correct conflicting transactions. In particular, apart from the BLS signature, it verifies the VRF proof and the commitment of the pending transaction. Next, it obtains the set of statekeepers who signed both conflicting transactions and ensures that the statekeeper being claimed is included in that set. Finally, the contract verifies the details of the zero-knowledge proof π'_{trans} and transfers the disputed funds upon success to the merchant.

Claim customer. If there is no conflicting transaction and the merchant followed the Quicksilver payment protocol, then the payment failed due to a malicious user. In this case, the merchant is guaranteed to be refunded by the Quicksilver smart contract from the user's collateral by triggering Algorithm 3. The merchant includes the ciphertexts that encrypt the payment value with the customer collateral's public key, and the zero-knowledge proofs $\pi_{\mathsf{claim}}, \pi_{\mathsf{trans}}$ that the customer created during the payment process. Furthermore, the merchant includes the pending disputed transaction τ^p and all transactions that were pending at the time of payment approval $\mathbb{T}_p$. The smart contract verifies that all attached transactions submitted by the merchant are valid and signed by a majority of statekeepers; the state is compatible for the settlement request, in particular that there exists no conflicting transaction in the system; the past collateral balance was sufficient to cover all pending payments; and ciphertexts for the claim encrypt opposite amounts. In this case, the smart contract transfers the disputed funds and update the current collateral balance, and stores the observed pending transaction in the contract state.

5.5 Brief Security Analysis

Next, we provide a brief security analysis by stating how Quicksilver safisfies our main security guarantees (censorship resilience and merchant safety). Due to limited space, we defer the full reasoning and proof sketches to Appendix A. In the same appendix, we also show that Quicksilver provides safety for customers

and statekeepers, ensures payment confidentiality, and enables a strong liveness guarantee.

Theorem 1 (Censorship resilience). *Given the pseudorandomness property of VRFs, assuming that the DDH assumption holds, and that the used commitment scheme is perfectly hiding, Quicksilver is censoring-resilient (cf. Definition 1).*

Theorem 2 (Merchant safety). *Given collision-resistance of VRFs, computationally binding commitments, and sound zero-knowledge proof system, Quicksilver provides merchant safety (cf. Definition 2).*

The main intuition why Thereom 2 above holds is that Quicksilver satisfies the below listed necessary conditions for merchant safety. We explain this in detail in Appendix A.

- **Condition 1:** The majority of the statekeepers have signed a payment intent that binds the pair (customer address, payment index) to the pair (merchant address, payment amount).
- **Condition 2:** At the time of payment acceptance, the customer has sufficient collateral to cover both the current payment and the list of already approved but pending payment provided by the user.
- **Condition 3:** The merchant can authorize the settlement of correct payment amount from the customer's collateral if needed.
- **Condition 4:** Each statekeeper that has signed the intent has enough collateral to cover the amounts of the current and all pending payments approved by the same statekeeper.
- **Condition 5:** The merchant can authorize the settlement of correct payment amount from the statekeeper's collateral if needed.

6 Performance Evaluation

6.1 Methodology and Experimental Setup

We implemented the Quicksilver system as a combination of an off-chain protocol in Rust/JavaScript and an on-chain smart contract in Solidity. Throughout our implementation, we used the `alt_bn128` elliptic curve since it is natively supported in the EVM. For the off-chain payment protocol, we relied Anonymous Zether [1], vrf-rs [4] and bn [2] libraries to construct the zero-knowledge and VRF proofs, and to perform the elliptic curve operations. In the Quicksilver smart contract, we additionally used the solidity-BN256G2 primitives [3].

We ran the customers, merchants and statekeepers on low-end machines with 2 vCPUs and 2 GB of RAM. Statekeepers were implemented as simple webservers. We deployed these merchants and statekeepers to cloud instances in 10 different locations (Mumbai, Toronto, Singapore, Dallas, Fremont, Atlanta, Newark, London, Sydney, Frankfurt).

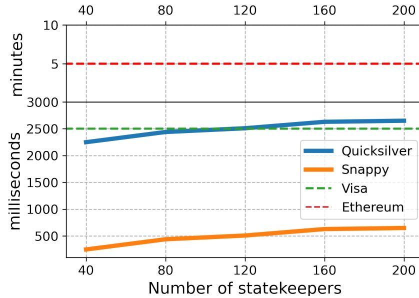

Table 6. Payment latency in Quicksilver.

Function	Gas	Ethereum	Polygon
Open Account	131,000	$1.4	$0.01
Fund Account	213,000	$2.3	$0.01
Conf. Transfer	4,798,000	$52	$0.3
Customer Registration	4,997,000	$54	$0.3
Merchant Registration	72,000	$0.8	$0.1
Statekeeper Registration			
50 merchants	5,489,000	$60	$0.3
100 merchants	8,307,000	$90	$0.5
Payment Process	5,028,000	$56	$0.3
Claim Statekeeper			
50 merchants	6,310,000	$68	$0.4
100 merchants	7,555,000	$82	$0.5

Table 7. Cost of Quicksilver operations.

For latency evaluation, we assume a scenario where the communication between the user and merchant is negligible (e.g., NFC at point-of-sale). To measure throughput, we deployed 200 merchants that collectively generate approval requests to a majority of statekeepers. Each merchant performed a series of back-to-back operations (requests) and we measured the end-to-end time taken by each operation. We then computed the total number of approval requests processed per second as an indicator for the system's throughput. To evaluate gas costs, we relied on the Solidity compiler (v0.7.0) and the Ganache client (v6.12.2) to create and deploy smart contracts. When measuring gas costs, we relied on operations and precompiled contracts available in the Ganache client.

6.2 Evaluation Results

Latency. The total time it takes to create and accept a Quicksilver payment is approximately 2.5 seconds, as shown in Figure 6. This is equal to the Visa contactless payment approvals [11] and significantly faster than standard blockchain payments.

Our measurements consist of intent creation time (steps 1-3) and the statekeeper latency (steps 4-5). We observe that the proof computation (π_{trans} and π_{claim}) needed for confidential payment and settlement dominates this latency taking more than 90% of the total latency. The required computational overhead of our censorship preventions (π_{vrf} and commitments) constitute only a minor component of the total delay. The intent signing round-trip protocol between the merchants and statekeepers takes approximately 700 ms. The statekeeper approval latency increases slightly as the number of statekeepers increases (from 40 to 200 in our experiments). Our latency measurements were obtained on a throughput of $5,000$ approval requests per second. We note that our current prototype uses an unoptimized JavaScript implementation from [1] for π_{trans} and π_{claim} computation. We expect a significant performance boost with an optimized implementation.

Throughput. We tested our prototype on throughput of $1,000$ and $2,500$ requests per second with only a negligible difference in latency. Only the overall variance was observed to increase with the throughput. Given that currently chains like Ethereum support orders of magnitude lower throughputs, we con-

Number of Merchants	Pending Transactions per User 0	1	2	3
50	9.9M	10,7M	11.4M	15.4M
100	10.5M	11.7M	13.0M	19.8M

Table 8. Confidential settlement gas cost.

Number of Merchants	Pending Transactions per User 0	1	2	3
100	1.9M	3.2M	4.7M	5.9M

Table 9. Non-confidential settlement gas cost.

clude that Quicksilver provides more than sufficient throughput for the underlying blockchain.

Gas usage. In Table 7, we measure the gas cost of each Quicksilver operation. We translate these costs to USD for two currently popular blockchain platforms. First, we use Ethereum as an example of a currently expensive platform (8.5 GWEI gas price, 1,277 USD/ETH), and second, we use Polygon as an example of a cheaper alternative (77 GWEI gas price, 0.78 USD/MATIC) based on gas prices and exchange rates from October 2, 2022.

We observe that one Quicksilver payment costs 5.03M gas that corresponds to $0.3 on Polygon and $56 on Ethereum. We note that recent Ethereum improvement proposals may reduce the payment costs significantly in near future.[5] Most of the gas costs for payment comes from the processing of confidential transactions. Our solution can be also deployed to support fast and censorship-resilient transactions that are not confidential. In this case, a Quicksilver payment takes 176K gas ($1.9 on Ethereum and $0.01 on Polygon).

The settlement processes are evaluated in Tables 8 (confidential payments) and 9 (non-confidential payments). The cost for claiming a statekeeper scales with the total number of registered merchants who act as statekeepers. The cost for claiming a user scales with the number of pending transactions. An example settlement for a confidential payment with no pending transactions and 50 merchants costs 9.9M gas ($0.6 on Polygon). To summarize, all Quicksilver operations are feasible on affordable blockchain platforms like Polygon today. On more expensive platforms like Ethereum, payments that are fast and censoring-resilient (but not confidential) are also feasible.

7 Concluding Remarks

In this paper, we presented Quicksilver, the first 2nd-Layer blockchain payment scheme that is fast, censorship-resilient and confidential. We designed and implemented Quicksilver for EVM compatible blockchain platforms and tested it on Ethereum and Polygon. Our prototype evaluation shows that payment approvals can be conducted approximately in 2.5 seconds which matches the requirements of most point-of-sale use cases.

Roadmap for the rest of the paper. In Appendix A, we provide sketches of security proofs for Quicksilver. We discuss collateral scalability and payment anonymity in Appendix B. In Appendix C, we explain gas fees could be reduced

[5] Additions and multiplications in the $\mathbb{G}_2$ group are not natively supported in Ethereum and cost $20,000$ and 2.5M gas, respectively. EIP-2537 [21] proposes precompiled contracts for additions/multiplications which would reduce these gas costs by orders of magnitude.

using customized zero-knowledge proofs. And finally, in Appendix D, we include the pseudocode listings for the Quicksilver smart contract algorithms.

Acknowledgments

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and by the Zurich Information Security and Privacy Center (ZISC).

References

1. Anonymous zether extension. Work in Progress.
2. Bn: Pairing cryptography with the barreto-naehrig curve.
3. solidity-bn256g2. Work in Progress.
4. Vrf implementation in rust. Work in Progress.
5. Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. *IACR Cryptol. ePrint Arch.*, 2018.
6. Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4), 2004.
7. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. *Zether: Towards Privacy in a Smart Contract World*, pages 423–443. 07 2020.
8. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Report 2017/1066, 2017.
9. David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, 1983.
10. Felix Engelmann, Henning Kopp, Frank Kargl, Florian Glaser, and Christof Weinhardt. Towards an economic analysis of routing in payment channel networks. In *Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2017.
11. Marc Freed-Finnegan and Joseph Koenig. Visa quick chip, 2017.
12. Arthur Gervais, Hubert Ritzdorf, Mario Lucic, Vincent Lenders, and Srdjan Capkun. Quantifying location privacy leakage from transaction prices. In *Computer Security – ESORICS 2016*, 2016.
13. Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Včelák. Verifiable Random Functions (VRFs). Internet-Draft draft-irtf-cfrg-vrf-09, 2021. Work in Progress.
14. Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Layer-two blockchain protocols. In *Financial Cryptography and Data Security (FC)*, 2020.
15. Vasilios Mavroudis, Karl Wüst, Aritra Dhar, Kari Kostiainen, and Srdjan Capkun. Snappy: Fast on-chain payments with practical collaterals. In *Network and Distributed System Security Symposium (NDSS)*, 2020.
16. Lucien KL Ng, Sherman SM Chow, Donald PH Wong, and Anna PY Woo. Ldsp: Shopping with cryptocurrency privately and quickly under leadership. In *International Conference on Distributed Computing Systems (ICDCS)*, 2021.
17. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO '91*, 1992.

18. Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.
19. Amritraj Singh, Kelly Click, Reza M Parizi, Qi Zhang, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Sidechain technologies in blockchain networks: An examination and state-of-the-art review. *Journal of Network and Computer Applications*, 149, 2020.
20. Nicholas van Saberhagen. Cryptonote v 2.0, October 2013.
21. Alex Vlasov and Kelly Olson. Eip-2537: Precompile for bls12-381 curve operations. *Ethereum Improvement Proposals*, (2537), 2020.

A Security Proofs

In this appendix, we provide sketches of security proofs for our solution Quicksilver.

A.1 Censorship Resilience

We now show that Quicksilver is censorship-resilient (Theorem 1). We assume that users and merchants initiating a payment do not disclose any information about each other or the payment intent to the public. We argue that little can be done if users/merchants leak the payment intent.

We prove censorship-resilience by showing that various constituents (price, customer address, merchant address, index number) of any legitimate payment intent in Quicksilver cannot be distinguished from the constituents of any random payment intent.

(a) *Price-based censorship:* Since users and merchants cannot leak the price, the only constituents of a payment intent that are dependent on the price comprise the ElGamal-encrypted value.
(b) *User-based censorship:* The only constituents of the payment intent that are derived from the user is the VRF hash.
(c) *Merchant-based censorship:* The only constituents of the payment intent that are derived from the merchant address are the Pedersen commitment of the merchant's Quicksilver account.

We therefore conclude that any p.p.t. distinguisher A can only distinguish a legitimate payment intent in Quicksilver from a random one with negligible probability. Otherwise, A can break the DDH assumption (to distinguish the value encrypted from ElGamal), or the pseudorandomness property of VRFs or the perfect hiding property of Pedersen commitments (to distinguish the merchant address).

A.2 Merchant Safety

Next, we show that Quicksilver provides safety for merchants (Theorem 2). For this proof sketch, we leverage the fact that the Snappy [15] paper already proves that if a merchant checks the following five conditions (also listed in Section 5.5) prior to payment acceptance, it is guaranteed to receive funds that are equal to the full amount of the accepted payment (Definition 2).

- **Condition 1:** The majority of the statekeepers have signed a payment intent that binds the pair (customer address, payment index) to the pair (merchant address, payment amount).
- **Condition 2:** At the time of payment acceptance, the customer has sufficient collateral to cover both the current payment and the list of already approved but pending payment provided by the user.
- **Condition 3:** The merchant can authorize the settlement of correct payment amount from the customer's collateral if needed.
- **Condition 4:** Each statekeeper that has signed the intent has enough collateral to cover the amounts of the current and all pending payments approved by the same statekeeper.
- **Condition 5:** The merchant can authorize the settlement of correct payment amount from the statekeeper's collateral if needed.

In Quicksilver, these five conditions are satisfied as follows:

- **Condition 1:** Recall that the PINT structure signed by the statekeepers contains commitment to the merchant address, encrypted payment amount, and a VRF hash computed using the customer's private key and the payment index as input. This condition holds, since otherwise the customer would be able to violate the binding property of commitments or violate the collision-resistance of VRFs.
- **Condition 2:** Prior to accepting a payment, the merchant verifies the proof π_{claim} which shows that the customer's collateral is sufficient to cover the current and approved pending payment. To ensure that the correct set of pending payments is considered in the above proof computation, the merchant also verifies the VRF proofs π_{vrf} for each pending payment. These proofs show that the signed RPID matches the payment index i in each pending payment. Therefore, to violate this condition, the merchant would need to forge π_{claim} which is proven secure in [7] or forge π_{vrf} by violating the collision-resistance property of VRFs.
- **Condition 3:** This condition holds since the merchant only accepts a payment if it receives a valid π_{claim} from the user. If a settlement from the user's collateral is needed, the merchant can present π_{claim} to the Quicksilver contract which can then execute the transfer from the user's collateral to the merchant. The Quicksilver contract enforces that the ciphertext used in the settlement proof is the same as the one used in the failed payment, and therefore the merchant is guaranteed to receive the correct payment amount (see Line 23 in Algorithm 3).

- **Condition 4:** This condition holds since in Quicksilver the merchant knows the private key for the statekeeper's encrypted collateral account, and thus it can easily verify that the statekeeper has sufficient collateral to cover the current and all pending payments approved for the merchant.
- **Condition 5:** The last condition holds, since the merchant knows the private key so that it can create the proof π'_{trans} that can be presented to the Quicksilver contract that will then execute the transfer of funds. The Quicksilver contract enforces that the ciphertext used as input in π'_{trans} is the same as the one in the failed payment, and therefore the merchant is guaranteed to receive the correct payment amount (see Line 6 in Algorithm 2).

We conclude that Quicksilver provides merchant safety, when VRFs are collision resistant, the used commitments are computationally binding, and the used zero-knowledge proof system is sound.

A.3 User Safety

Next, we define what it means that a payment system is safe for users.

Definition 3 (User safety). *If a user signs a complete transaction τ with payment amount b^*, the user is guaranteed to lose at most b^* of its funds.*

To satisfy Definition 3, we need to show that the following two conditions hold. First, if τ is included on the chain, no settlement will be performed from the user's collateral. Conversely, if τ is not included in the chain, at most b^* will be settled from the user's collateral.

The first condition holds since the Quicksilver smart contract verifies that the claimed transaction is not already accepted and recorded on the state of the Quicksilver contract (Line 2 in Algorithm 3). On the other hand, the second condition holds since the user creates the settlement proof π_{trans} and the Quicksilver contract enforces that the settled amount is the same ciphertext used in the failed payment τ. Recall that only the user knows the private key of his collateral account, and thus only him can create proofs to authorize transfers from this account.

A.4 Statekeeper Safety

Since, in Quicksilver, statekeepers also have a collateral account, we also need to show that Quicksilver provides safety guarantees for statekeepers.

Definition 4 (Statekeeper safety). *If a statekeeper follows the protocol, no funds will be transferred from any of his M collateral accounts $(y_1, ..., y_M)$.*

Recall that the statekeeper shares the private keys $(x_1, ..., x_M)$ of these collateral accounts with the merchants. This means that each merchant is able to create arbitrary proofs π'_{trans} that would—without other enforcements—transfer

funds from the statekeeper's collateral account. However, the Quicksilver contract ensures (Line 3 in Algorithm 2) that transfers from the statekeeper's collateral are only possible, if the statekeeper has equivocated (signed more than one PINT with the same RPID value). Thus, no funds will be transferred from any of the statekeepers collaterals, unless the statekeeper deviates from the protocol.

A.5 Payment Confidentiality and Liveness

In Quicksilver, the payment amount appears only in encrypted format on the finalized transaction τ—using similar techniques to Zether. Therefore, Quicksilver inherits the payment confidentiality guarantees of Zether that are shown in [7]. In Quicksilver, the only information that depends on the payment amount and is shared beyond the user and the merchant, is the commitment of the payment amount included in PINT. Thus, the payment confidentiality of Quicksilver also relies on the hiding property of the used commitment scheme.

Finally, Quicksilver inherits the liveness guarantee of Snappy [15]. If a majority of the statekeepers are reachable and sign the payment intent, the merchant can safely accept the payment.

B Discussion

Collateral scalability. Customer collaterals in Quicksilver are small, since they only need to account for payments the customer does within the latency period of the underlying blockchain platform (e.g., 3 minutes in Ethereum). During the latency period, all Quicksilver payments are assumed to be pending, and hence the collateral must be able to cover the customer's expenditure during this time. For instance, if the customer spends at most $100 during the latency period, a collateral of $100 will be sufficient to cover all pending payments. The customer's collateral is independent of the total number of registered users and the number of merchants.

The collateral size of statekeepers is independent of the number of registered customers, and only scales with the total value of merchant's sales during the latency period. For instance, in a deployment of 100,000 customers, 100 merchants, and a latency period of 3 minutes, the statekeepers only account for customers that are active during the block latency period. If customers spend $5 during the latency period on the average and the merchant accepts one payment every 30 seconds, this results in a collateral of $3,000 per statekeeper. A more detailed collateral analysis is presented in [15].

Payment anonymity. Quicksilver supports payment confidentiality based on encrypted account balances that are updated using zero-knowledge proofs. The Zether paper [7] outlines how such payments can be extended to support payment anonymity. The main idea is to choose an anonymity set (i.e., a set of encrypted account balances) for the sender and the recipient, and construct a zero-knowledge proof that shows that the balance of one of the sender accounts

and one of the recipient accounts was updated with a non-zero value without revealing which accounts were updated.

The same technique could be applied to Quicksilver. However, just using anonymity sets for the sender and recipient accounts would not be sufficient to provide full payment anonymity. In a system like Quicksilver, de-anonymization could be feasible by correlating payment index values between subsequent payments. An anonymous variant of Quicksilver would, therefore, need to hide payment index values and similar sources of de-anonymization from the on-chain transactions. Another challenge is that an anonymous payment scheme would need Ethereum transaction signatures from a fresh address to prevent identity leakage. Obtaining unlinkable Ethereum addresses that have the required funds to cover the transaction fees is a common challenge for anonymous applications on systems like Ethereum. A third challenge is related to high transaction processing costs, because zero-knowledge proofs over large anonymity sets are significantly more expensive compared to the proofs used in Quicksilver. Overcoming such technical challenges and thus extending Quicksilver with practical payment anonymity would be an interesting direction for future work.

C Optimized Statekeeper Registration

In Quicksilver, during registration, a statekeeper needs to make M confidential payments to establish M separate collateral accounts (one fore each registered merchant). The use of this technique prevents merchants from making claims outside their assigned collateral allocation. However, such registration process is expensive—essentially M times the cost of a confidential payment. Next, we outline how the statekeeper registration operation could be optimized for reduced gas use.

As before, we assume that the statekeeper already has an existing (encrypted) account balance. To register, the statekeeper could assign an (encrypted) allocation from his account for each merchant and disclose the plaintext allocation to each corresponding merchant. The statekeeper would then need to create a separate ciphertext (encrypted under the public key of the statekeeper) for each merchant and share the randomness used in the computation of this ciphertext with the corresponding merchant. Once such ciphertexts are computed, the statekeeper can add them together (since they are encrypted under the same public key) and use the ProveTransfer function to create a proof π_{trans} that shows that the statekeeper's account balance exceeds the sum of the above collateral allocations altogether. The Quicksilver smart contract would then verify π_{trans}, and store each ciphertext in $S[y_s].\mathit{pcol}[y_m]$ that corresponds to the encrypted allocation for merchant y_m.

The statement that is proven in zero knowledge to create π_{trans} in the case of statekeeper registration looks as follows [7]:

$$\begin{aligned}\mathsf{st}_{trans} :\{&(y, \overline{y}, C_L, C_R, C, \overline{C}, D, g; x, b^*, b', r^*)|\\ &C = g^{b^*} y^{r^*} \wedge \overline{C} = g^{b^*} \overline{y}^{r^*} \wedge D = g^{r^*} \wedge\\ &C_L/C = g^{b'}(C_R/D)^x \wedge y = g^x \wedge\\ &b^* \in \{0, \cdots, \mathrm{MAX}\} \wedge b' \in \{0, \cdots, \mathrm{MAX}\}\},\end{aligned}$$

where x is the statekeepers's secret key, b^* is the total collateral value, b' is the statekeeper's balance after all collateral allocations have been deducted from it, and r^* is the combined randomness used for each encryption.

If settlement is needed, a merchant could then claim funds from the statekeeper's account by constructing a novel proof that we call π_{ClaimSk}. This proof would prove in zero knowledge the following statement that would allow the merchant to authorize settlement from the statekeeper to the merchant:

$$\begin{aligned}\mathsf{st}_{ClaimSk} :\{&(y, \overline{y}, C_L, C_R, C, \overline{C}, D, g; x, b^*, b', r, r^*)|\\ &C = g^{b^*} y^{r^*} \wedge \overline{C} = g^{b^*} \overline{y}^{r^*} \wedge D = g^{r^*} \wedge\\ &C_L/C = g^{b'} y^{r-r^*} \wedge C_R = g^r \wedge \overline{y} = g^x \wedge\\ &b^* \in \{0, \cdots, \mathrm{MAX}\} \wedge b' \in \{0, \cdots, \mathrm{MAX}\}\},\end{aligned}$$

where x is the merchant's secret key, b^* is the payment value to be settled, b' is the updated statekeepers's collateral after settlement, r is the randomness of the statekeeper's encrypted collateral allocations, and r^* is the randomness used for the claimed confidential payment. The main intuition here is that, unlike in normal confidential transfers, the private key of the sender (in this case, statekeeper) is not required to create the zero-knowledge proof, since

$$C_L/C = g^{b'}(C_R/D)^x = g^{b'} g^{x(r-r')} = g^{b'} y^{r-r^*} = C_L/C.$$

Only $C_R = g^r$ must be verified additionally to ensure that the ciphertext is correct.

The merchant knows both r^* and r, since statekeepers disclose the plaintext of the allocated collateral amount and its randomness to the corresponding merchant. Since only the corresponding merchant can claim from his assigned allocated collateral, he can easily keep track of random values used.

The registration is cheaper with this approach because statekeepers now need to perform only one confidential payment of 4.8M gas to transfer their collateral to only one account (instead of M confidential payments previously) which significantly improves collateral scalability with regards to number of registered merchants/statekeepers.

D Smart Contract Algorithms

The main algorithms supported by the Quicksilver smart contract are called Record-and-Transfer, Claim-Statekeeper and Claim-Customer. These algorithms are listed below.

Algorithm 1: Record-and-Transfer. The Quicksilver smart contract records the payment and then completes the confidential transfer from user to merchant.

```
Actor  : Quicksilver (smart contract)
Input  : Quicksilver transaction τ
Output : ⊤ or ⊥
1 y_c ← τ_{y_c}
2 if y_c ∈ C and τ_i ∉ C[y_c].D then
      /* Record payment */
3     h ← H(τ_RPID, τ_c, τ_i)
4     C[y_c].D[τ_i] ← ⟨h, τ_A, τ_q, 0⟩
      /* Complete confidential transfer */
5     if VerifyTransferProof(τ_{y_c}, τ_{y_m}, Acc[y_c].bal, τ_{c*_c}, τ_{c*_m}, π_trans) then
6         Acc[y_m].P ← Acc[y_m].P ∘ τ_{c*_m}              ▷ Add recipient
7         Acc[y_c].bal ← Acc[y_c].bal ∘ τ_{c*_c}^{-1}     ▷ Deduct sender
8         return ⊤
9 return ⊥
```

Algorithm 2: Claim-Statekeeper. Quicksilver smart contract sends lost funds from the misbehaving statekeepers' collaterals to the affected merchant.

```
Actor  : Quicksilver (smart contract)
Input  : Pending Transaction τ^p
         Statekeeper's public key y_s
         Statekeeper's ciphertext c*_s
         VRF proofs Π_vrf, ZKP π'_trans, blinding value r
         Conflicting transactions ⟨τ', τ''⟩
Output : ⊤ or ⊥
/* Verify all transactions */
1 𝕋 ← {τ^p, τ', τ''}
2 if τ^p_c = Open(τ^p_{y_m}, r) and VerifyBLS(𝕋) and VerifyVRF(𝕋, Π_vrf) and τ'_i = τ''_i ≤ τ^p_i and
    τ' ≠ τ'' and τ^p_{y_c} = τ'_{y_c} = τ''_{y_c} and τ' ≠ τ'' then
      /* Find set of equivocating statekeepers */
3     SK ← FindOverlapSet(τ', τ'')
      /* Ensure claimed statekeeper is in that set */
4     if SK ≠ ∅ and ∃y'_s ∈ SK : y'_s = y_s ∧ S[y_s] ≠ ⊥ then
5         y_col ← S[y_s].y[y_m]
          /* Verify ZKP */
6         if VerifyTransfer(y_col, τ^p_{y_m}, Acc[y_col].bal, c*_s, τ^p_{c*_m}, π'_trans) then
              /* Transfer lost funds */
7             Acc[y_m].P ← Acc[y_m].P ∘ c*_m              ▷ Add to merchant
8             Acc[y_col].bal ← Acc[y_col].bal ∘ c*_s^{-1}  ▷ Deduct
9             return ⊤
10 return ⊥
```

Algorithm 3: Claim-customer. Quicksilver recovers lost funds from the customer's collateral if the request is correct.

Actor : Quicksilver (smart contract)
Input : Pending transaction τ, pending transactions $\mathbb{T}_p$
VRF proof π_{vrf}, ZKPs π_{claim}, π_{trans}, blinding value r
Output : $\top$ or $\bot$

1 $\mathrm{I}^* \leftarrow C[y_c].D$ ▷ Pass by reference
/* Verify tx, ensure not yet processed. */
2 **if** $!\mathit{VerifyTx}(\tau, \pi_{\mathsf{vrf}}, r)$ *or* $\tau_i \in I^*$ **then**
3 | **return** $\bot$
/* Verify signatures of preceding non-pending txs. */
4 **for** $\forall\{i \in I^* | I^*[i].b = 0\}$ **do**
5 | **if** $\mathit{Verify}(I^*[i]_A)$ **then**
6 | | $I[i].b \leftarrow 1$
7 | **else**
8 | | del I^*[i] ▷ Past tx had no approval.
/* Any pending & preceding txs missing? */
9 **if** $\exists\, i \in \{1 \ldots \tau_{i-1}\}$ *such that* $i \notin I^*$ *and* $i \notin \mathbb{T}_p$ **then**
10 | **return** $\bot$
/* Verify signatures of pending preceding txs. */
11 **if** $!\mathit{VerifyBLS}(\mathbb{T}_p)$ **then**
12 | **return** $\bot$
/* Ensure there are no conflicting transactions. */
13 **for** $\forall \tau' \in \{\tau\} \cup \mathbb{T}_p$ **do**
14 | **if** $C[y_c].O[\tau'_i].h \neq H(\tau')$ **then**
15 | | **return** $\bot$
16 $J^* \leftarrow C[y_c].T$ ▷ Pass by reference
/* Find past compatible collateral. */
17 **for** $\forall\{j \in J^*\}$ **do**
18 | **if** $J^*[j].idx < \tau_i \vee J^*[j].idx = \bot$ *and* $\nexists \tau' : \tau' \in \mathbb{T}_p \wedge J^*[j].idx = \tau_i$ **then**
19 | | $bal' \leftarrow J^*[j].bal$
/* Ensure collateral covers all pending txs. */
20 $c_p^* \leftarrow (1, 1)$
21 **for** $\forall\{c^* \in \tau' | \tau' \in \{\tau\} \cup \mathbb{T}_p\}$ **do**
22 | $c_p^* \leftarrow c_p^* \circ c^*$
/* Verify both the transfer and claim proofs */
23 **if** $!\mathit{VerifyTransfer}(\tau_{y_c}, \tau_{y_m}, bal', {c_p^*}^{-1}, \pi_{\mathsf{claim}})$ *or*
24 $!\mathit{VerifyTransfer}(\tau_{y_c}, \tau_{y_m}, bal', \tau_{c^*}^{-1}, \tau_{c_m^*}, \pi_{\mathsf{trans}})$ **then**
25 | **return** $\bot$
/* Process claim. */
26 $Acc[\tau_{y_m}].P \leftarrow Acc[\tau_{y_m}].P \circ c_m^*$
27 $C[\tau_{y_c}].\mathit{pcol} \leftarrow C[\tau_{y_c}].\mathit{pcol} \circ \tau_{c_c^*}^{-1}$
/* Log the updated collateral for the payment index */
28 $\mathrm{Append}(J^*, (\tau_i, C[\tau_{y_c}].\mathit{pcol}))$
/* Store observed txs that are not yet finalized */
29 **for** $\forall \tau' \in \mathbb{T}_p : \tau' \notin I^*$ **do**
30 | $C[c].O[\tau'_i].h \leftarrow H(\tau')$
31 **return** $\top$