Quisquis: A New Design for Anonymous Cryptocurrencies

Prastudy Fauzi[1], Sarah Meiklejohn[2], Rebekah Mercer[3], and Claudio Orlandi[4]

[1] Simula UiB, Norway
[2] University College London, UK
[3] O(1) Labs, USA
[4] Department of Computer Science, DIGIT, Aarhus University, Denmark

Abstract. Despite their usage of pseudonyms rather than persistent identifiers, most existing cryptocurrencies do not provide users with any meaningful levels of privacy. This has prompted the creation of privacy-enhanced cryptocurrencies such as Monero and Zcash, which are specifically designed to counteract the tracking analysis possible in currencies like Bitcoin. These cryptocurrencies, however, also suffer from some drawbacks: in both Monero and Zcash, the set of potential unspent coins is always growing, which means users cannot store a concise representation of the blockchain. Additionally, Zcash requires a common reference string and the fact that addresses are reused multiple times in Monero has led to attacks to its anonymity.

In this paper we propose a new design for anonymous cryptocurrencies, Quisquis, that achieves provably secure notions of anonymity. Quisquis stores a relatively small amount of data, does not require trusted setup, and in Quisquis each address appears on the blockchain at most twice: once when it is generated as output of a transaction, and once when it is spent as input to a transaction. Our result is achieved by combining a DDH-based tool (that we call updatable keys) with efficient zero-knowledge arguments.

1 Introduction

Bitcoin was introduced in 2008 [30], and at a high level it relies on the use of *addresses*, associated with a public and private key pair, to keep track of who owns which coins. Users of the system can efficiently create and operate many different addresses, which gives rise to a form of pseudo-anonymity. As is now well known, however, Bitcoin and other cryptocurrencies relying on this level of pseudo-anonymity can, in practice, have these addresses linked together and even linked back to their real-world identities with little effort [33,34,3,26,37,28].

Due to this, there has now been an extensive body of work aiming to provide privacy-enhanced solutions for cryptocurrencies, although even some of these new solutions have also been subjected to empirical analyses pointing out the extent to which they can be de-anonymized as well [29,25,27,20,19,18,40]. These solutions typically fall into two main categories.

First, *tumblers* (also known as mixers or mixing services) act as opt-in overlays to existing cryptocurrencies such as Bitcoin [23,36,16] and Ethereum [24], and achieve enhanced privacy by allowing senders to mix their coins with those of other senders. While these are effective and arguably have a high chance of adoption due to their integration with existing cryptocurrencies, they also have some limitations. In particular, they are generally either dependent on trusting a central mixer, which leaves users vulnerable to attacks on availability, or they require significant coordination amongst the parties wishing to mix, which leads to higher latency as users must wait for other people to mix with them.

Second, there are cryptocurrencies with privacy features built in at the protocol level. Of these, the ones that have arguably achieved the most success are Dash [1], Monero [31], and Zcash [6]. Dash is derived from a tumbler solution, Coinjoin [23], and thus inherits the properties discussed there. In Monero, senders specify some number of addresses to "mix in" to their own transaction, and then use this list of public keys to form a ring signature and hide which specific address was theirs. Observers of the blockchain thus learn only that some unknown number of coins have moved from one of these input public keys. In Zcash, users can put coins into a "shielded pool." When they wish to spend these coins, they prove in zero-knowledge that they have the right to spend some specific coins in the pool, without revealing which ones.

Between Monero and Zcash, there are already several differences. For example, because users in Monero specify rings themselves, they achieve a form of *plausible deniability*: no one can tell if a user meant to be involved in a given transaction, or if their address was simply used in a ring without their consent. In Zcash, in contrast, every other user in a user's anonymity set has no such deniability, as they at one point intentionally put coins into the shielded pool.

One limitation central to both cryptocurrencies, however, is the information that peers in the network are required to keep. In Bitcoin, the list of all addresses with a positive balance can be thought of as a set of *unspent transaction outputs* (UTXOs). When a sender spends coins, their address ceases to be a UTXO, so is replaced in the set with the address of their recipient. Full nodes can thus collapse the blockchain into this UTXO set, and check for double spending simply by checking if a given input address is in the set or not. In other words, it acts as a concise representation of the entire history of the blockchain. In October 2017, for example, there had been over 23 million Bitcoin transactions and the total size of the blockchain was over 130 GB, but the size of the UTXO set was only 3 GB [12].

In Monero and Zcash, however, addresses can (essentially) never be removed from the UTXO set, as it is never clear if an address has spent its contents or was simply used as part of the anonymity set in the transaction of a different sender. The size of the UTXO set is thus monotonically increasing: with every transaction, it can only grow and never shrink. This has a significant impact on full nodes, as they must effectively store the entire blockchain without the option of the concise representation possible in Bitcoin.

Our Contributions We present Quisquis, a new design for anonymous cryptocurrencies that resolves the limitations outlined above for existing solutions. In particular, users are able to form transactions on their own, so do not need to wait for other interested users and incur the associated latency. They can also involve the keys of other users without their permission, which gives the same degree of plausible deniability as Monero. Finally, each transaction acts to replace all the input public keys in the UTXO set with all the output public keys, thus allowing the UTXO set to behave in the same manner as in Bitcoin. Furthermore, our transactions are relatively inexpensive to compute and verify, taking around 471ms to compute and 71ms to verify for an anonymity set of size 16, with proofs of size approximately 13kB.

As a brief technical overview, Quisquis achieves anonymity using a primitive that we formalize in Section 3 called *updatable* public keys, which allows users to create updated public keys, indistinguishable from ones that are freshly generated, without changing the underlying secret key. After formally defining our threat model in Section 4, we present our full construction of Quisquis in Section 5. Roughly, senders take the keys of other users, including their intended recipients, to form a list of public keys that act as the input to a transaction. A sender can now "re-distribute their wealth" among these input keys, acting to move some of their own coins to the recipient and keeping the (hidden) balances of the other members of the anonymity set the same. To ensure anonymity, the output public keys are all updated, and all balances and amounts are given only in committed form. Thus, by design, in Quisquis every address can only ever appear at most twice on the blockchain: once when it is generated in the output of the transaction, and once when it is spent as input to a different transaction. This greatly reduces (compared, e.g., to Monero) the ability of an attacker to perform de-anonymization attacks based on how often a certain address participates in transactions.

To ensure integrity, the sender proves in zero-knowledge that they have correctly updated the keys and have not taken money away from anyone except themselves. Crucially, because the witness for the zero-knowledge proof is limited to this single transaction (as opposed to encompassing other parts of the blockchain), we can use standard discrete-log-based techniques as opposed to the heavyweight zk-SNARKs required in Zcash. This means that security can depend entirely on DDH, and no trusted setup is required (as we use the random oracle model to make the proofs non-interactive and to generate other system parameters and random values using "nothing up my sleeves" methods). As the design of Quisquis is modular, other tradeoffs could be achieved as well: for instance, it could be possible to instantiate Quisquis with zk-SNARKs as well, thus achieving even smaller transactions and faster verification at the cost of much slower transaction generation and the stronger assumptions underlying zk-SNARKs.

To demonstrate the efficiency of Quisquis, we implement it and present performance benchmarks in Section 7. We then provide a thorough comparison with existing solutions in Section 8 before concluding in Section 9.

2 Cryptographic Primitives

2.1 Notation

Let $\log_g h$ be the discrete log of h with respect to g. Define $(a,b)^c := (a^c, b^c)$ and $(a,b) \cdot (c,d) := (ac, bd)$. For vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, let $\boldsymbol{a} \circ \boldsymbol{b}$ be the Hadamard product of $\boldsymbol{a}$ and $\boldsymbol{b}$; i.e., the vector $\boldsymbol{c}$ such that $c_i = a_i b_i$. We use $y \leftarrow A(x)$ to denote assigning to y the output of a deterministic algorithm A on input x, and $y \xleftarrow{\$} A(x)$ if A is randomized; i.e., we sample a random r and then run $y \leftarrow A(x;r)$. We use $[A(x)]$ to denote the set of values that have non-zero probability of being output by A on inputs x. We use $r \xleftarrow{\$} R$ for sampling an element r uniformly at random from a set R. If $\boldsymbol{y} = (y_1, \ldots, y_n) \xleftarrow{\$} A(x)$ then we often denote y_i by $\boldsymbol{y}_i$.

2.2 Zero-knowledge arguments of knowledge

Let R be a binary relation for instances x and witnesses w, and let L be its corresponding language; i.e., $L = \{x \mid \exists w \colon (x,w) \in R\}$. An interactive proof is a protocol where a prover P tries to convince a verifier V, by an exchange of messages, that an instance x is in the language L. The set of messages exchanged is known as a *transcript*, from which a verifier can either accept or reject the proof. The proof is *public-coin* if an honest verifier generates his responses to P uniformly at random. An interactive proof is a special honest-verifier zero-knowledge argument of knowledge if it satisfies the following properties:

- Perfect completeness: if $x \in L$, an honest P always convinces an honest V.
- Special honest-verifier zero-knowledge (SHVZK): there exists a simulator S that, given $x \in L$ and an honestly generated verifier's challenge c, produces an accepting transcript which has the same (or indistinguishably different) distribution as a transcript between honest P, V on input x.
- Argument of knowledge: if P convinces V of an instance x, there exists an extractor with oracle access to P that runs in expected polynomial-time to extract the witness w.

A public-coin SHVZK argument of knowledge can be turned into a non-interactive zero knowledge (NIZK) argument of knowledge using the Fiat-Shamir heuristic. Essentially, non-interactivity is achieved by replacing the verifier's random challenge with the output of a hash function, which in the security proof is modeled as a random oracle.

2.3 Commitments

We use a commitment scheme Commit relative to a public key pk that, given a message $m \in \mathcal{M}$ and randomness $r \in \mathcal{R}$, computes $\mathsf{com} \leftarrow \mathsf{Commit}_{\mathsf{pk}}(m;r)$. Our commitments must satisfy two properties: first, they are computationally hiding, meaning for any two messages m_0, m_1, an adversary has negligible advantage in

distinguishing between $\mathsf{Commit}_{\mathsf{pk}}(m_0; U_{\mathcal{R}})$ and $\mathsf{Commit}_{\mathsf{pk}}(m_1; U_{\mathcal{R}})$, where $U_{\mathcal{R}}$ is the uniform distribution over the randomness space. Second, they are unconditionally binding, meaning even given the sk relative to pk, a commitment cannot be opened to two different messages.

Beyond these two basic properties, we require two extra properties from our commitments. First, they must be homomorphic in the sense that for some operation $\odot$ it holds that $\mathsf{Commit}_{\mathsf{pk}}(m) \odot \mathsf{Commit}_{\mathsf{pk}}(m') = \mathsf{Commit}_{\mathsf{pk}}(m + m')$ (for appropriate randomness). Second, they must be *key-anonymous*, meaning that for any honestly generated keys $\mathsf{pk}_0, \mathsf{pk}_1$ and adversarially chosen m, the tuple $(m, \mathsf{pk}_0, \mathsf{pk}_1, \mathsf{Commit}_{\mathsf{pk}_0}(m))$ is indistinguishable from $(m, \mathsf{pk}_0, \mathsf{pk}_1, \mathsf{Commit}_{\mathsf{pk}_1}(m))$.

We can construct such commitments in a group $(\mathbb{G}, g, p)$ where the DDH problem is hard, by essentially performing an ElGamal encryption in the exponent relative to public keys of the form $\mathsf{pk} = (g_i, h_i)$ (which are what we use in our later constructions). In particular, $\mathsf{Commit}_{\mathsf{pk}}(v; r)$ returns $\mathsf{com} = (c, d)$ where $c = {g_i}^r$ and $d = g^v {h_i}^r$. It is easy to verify that this commitment scheme is unconditionally binding, computationally hiding, key-anonymous, and additively homomorphic.

Finally, we also use *extended Pedersen commitments* in the constructions of our zero-knowledge (ZK) arguments; i.e., schemes that commit to a vector of values using a single group element.

3 Updatable Public Keys

This section introduces the notion of an updatable public key (UPK), in which public keys can be updated in a public fashion, and such that they are indistinguishable from freshly generated keys. This idea has been considered before in the context of several cryptographic primitives, such as signatures [14,4] and public-key encryption [39], but we wish to define it solely for keys, regardless of the primitive they are used to support.

We begin by defining security for UPKs. Our definitions of indistinguishability and unforgeability resemble those that have already been used for Bitcoin *stealth keys* [24] and in the context of other cryptographic primitives [4,39,22].

Indeed, we could continue to be inspired by stealth keys in our construction of a UPK scheme, but given their reliance on hash functions this would render us unable to prove statements about the keys using discrete log-based techniques, as we would like to do in our construction of Quisquis in Section 5. We thus present instead a purely algebraic UPK scheme based on DDH, inspired by "incomparable public keys" [39].

3.1 Security definitions

An *updatable public key system* (UPK) is described by the following algorithms:

- $\mathsf{params} \xleftarrow{\$} \mathsf{Setup}(1^\kappa)$ outputs the parameters of the scheme, including the public and secret key spaces $\mathcal{PK}, \mathcal{SK}$. These are given implicitly as input to all other algorithms.

- $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\kappa)$ takes as input a security parameter κ and outputs a public key $\mathsf{pk} \in \mathcal{PK}$ and a secret key $\mathsf{sk} \in \mathcal{SK}$.
- $(\{\mathsf{pk}'_i\}_{i=1}^n) \xleftarrow{\$} \mathsf{Update}(\{\mathsf{pk}_i\}_{i=1}^n)$ takes as input public keys $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ and outputs a new set of public keys $(\mathsf{pk}'_1, \ldots, \mathsf{pk}'_n)$.
- $0/1 \leftarrow \mathsf{VerifyKP}(\mathsf{pk}, \mathsf{sk})$ takes as input $\mathsf{pk} \in \mathcal{PK}$ and $\mathsf{sk} \in \mathcal{SK}$ and checks whether or not $(\mathsf{pk}, \mathsf{sk})$ is a valid key pair.
- $0/1 \leftarrow \mathsf{VerifyUpdate}(\mathsf{pk}', \mathsf{pk}, r)$ takes as input public keys $\mathsf{pk}', \mathsf{pk}$, and randomness r and checks if pk' was output by $\mathsf{Update}(\mathsf{pk}; r)$.

We require a UPK to satisfy the following properties.

Definition 1 (Correctness). *A UPK satisfies perfect* correctness *if the following three properties hold for all* $(\mathsf{pk}, \mathsf{sk}) \in [\mathsf{Gen}(1^\kappa)]$*: (1) the keys verify, meaning* $\mathsf{VerifyKP}(\mathsf{pk}, \mathsf{sk}) = 1$*; (2) the update process can be verified, meaning* $\mathsf{VerifyUpdate}(\mathsf{Update}(\mathsf{pk}; r), \mathsf{pk}, r) = 1$ *for all* $r \in \mathcal{R}$*; and (3) the updated keys verify, meaning* $\mathsf{VerifyKP}(\mathsf{pk}', \mathsf{sk}) = 1$ *for all* $\mathsf{pk}' \in [\mathsf{Update}(\mathsf{pk})]$.

We next define indistinguishability, which says that an adversary cannot distinguish between a freshly generated public key and an updated version of a public key it already knows.

Definition 2 (Indistinguishability). *Consider the following experiment:*

1. $(\mathsf{pk}^*, \mathsf{sk}^*) \xleftarrow{\$} \mathsf{Gen}(1^\kappa)$*;*
2. $\mathsf{pk}_0 \xleftarrow{\$} \mathsf{Update}(\mathsf{pk}^*)$*;*
3. $(\mathsf{pk}_1, \mathsf{sk}_1) \xleftarrow{\$} \mathsf{Gen}(1^\kappa)$.

A UPK satisfies indistinguishability *if for any PPT adversary* $\mathcal{A}$*:*

$$|\Pr[\mathcal{A}(\mathsf{pk}^*, \mathsf{pk}_0) = 1] - \Pr[\mathcal{A}(\mathsf{pk}^*, \mathsf{pk}_1) = 1]| \leq \mathsf{negl}(\kappa).$$

Finally, we require that an adversary should not be able to learn the secret key of an updated public key (unless it already knew the secret key for the original public key). This is formalized by saying that the adversary cannot produce a public key for which it knows both the secret key and the randomness needed to explain this public key as an update of an honestly generated public key.

Definition 3 (Unforgeability). *A UPK satisfies* unforgeability *if for any PPT adversary* $\mathcal{A}$*:*

$$\Pr[\mathsf{VerifyKP}(\mathsf{pk}', \mathsf{sk}') = 1 \wedge \mathsf{VerifyUpdate}(\mathsf{pk}', \mathsf{pk}, r) = 1 \mid (\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\kappa); (\mathsf{sk}', \mathsf{pk}', r) \xleftarrow{\$} \mathcal{A}(\mathsf{pk})] \leq \mathsf{negl}(\kappa).$$

3.2 UPKs from DDH

We present a construction of UPK based over a prime-order group $(\mathbb{G}, g, p)$ where the DDH assumption is believed to hold. Thus, our Setup outputs only publicly verifiable parameters, and does not need to be run by a trusted party. The rest of the algorithms are as follows:

- $\mathsf{Gen}(1^\kappa)$: Sample $r, \mathsf{sk} \xleftarrow{\$} \mathbb{F}_p$ and output $\mathsf{pk} = (g^r, g^{r \cdot \mathsf{sk}})$.
- $\mathsf{Update}(\{\mathsf{pk}_i\}_{i=1}^n)$: Parse $\mathsf{pk}_i = (g_i, h_i)$. Sample $r \xleftarrow{\$} \mathbb{F}_p$ and compute $\mathsf{pk}'_i = \mathsf{pk}_i^r = (g_i^r, h_i^r)$ for all i.
- $\mathsf{VerifyKP}(\mathsf{pk}, \mathsf{sk})$: Parse $\mathsf{pk} = (g', h')$ and output $(g')^{\mathsf{sk}} \stackrel{?}{=} h'$.
- $\mathsf{VerifyUpdate}(\mathsf{pk}', \mathsf{pk}, r)$: Output $\mathsf{Update}(\mathsf{pk}; r) \stackrel{?}{=} \mathsf{pk}'$.

Lemma 1. *The scheme above is a UPK satisfying Definitions 1 - 3 if the DDH assumption holds in* $(\mathbb{G}, g, p)$.

Proof. Correctness is straightforward to verify. To prove indistinguishability, our reduction receives a DDH challenge $\mathsf{chl} = (g, g^x, g^y, g^z)$, samples a value $r \xleftarrow{\$} \mathbb{F}_p$, and defines $\mathsf{pk}^* = (g^r, g^{xr})$ and $\mathsf{pk}' = (g^{yr}, g^{zr})$. It then invokes the indistinguishability adversary $\mathcal{A}$ on input $(\mathsf{pk}^*, \mathsf{pk}')$. If chl is a DDH tuple then pk' is distributed identically to pk_0, and if chl is not a DDH tuple then pk' is distributed identically to pk_1. Therefore, our reduction has the same (non-negligible) advantage in the DDH game as the $\mathcal{A}$ has in the indistinguishability game.

To prove unforgeability, our reduction receives a DL challenge $\mathsf{chl} = (g, h)$, picks a random $t \xleftarrow{\$} \mathbb{F}_p$, and sets $(g_0, h_0) = (g^t, h^t)$. The reduction now runs $(s, (g_1, h_1), r) \xleftarrow{\$} A(g_0, h_0)$, and outputs s. The input to the adversary in the reduction is distributed identically as in the definition of security. The winning condition of the security definition requires that $h_1 = g_1^s$ and $(g_1, h_1) = (g_0^r, h_0^r) = (g^{rt}, h^{rt})$ thus implying that $g^{srt} = h^{rt}$ or equivalently that $h = g^s$, meaning s is a valid solution to the DL oracle.

4 Threat Model

In this section, we present our model for cryptocurrency transactions, in which we view a transaction not as just transferring value from a sender to a recipient but as participants "re-distributing wealth" amongst themselves. Before presenting this model in Section 4.2, we first present the notion of an *updatable account* in Section 4.1, which is an extension of updatable public keys that associates them with a (hidden) balance; this is mainly done as a way to simplify notation in future sections. We then present the relevant notions of security in Section 4.3, focusing on *anonymity* (meaning no one can identify the "true" sender and recipient within the set of participants in a transaction) and *theft prevention* (meaning no one can steal the coins of other people or otherwise inflate their own wealth).

4.1 Updatable accounts

To represent an *account* in a cryptocurrency, we use pairs $\mathsf{acct} = (\mathsf{pk}, \mathsf{com})$ of public keys, which act as the pseudonym for a user, and commitments, which represent the balance associated with that public key.

In more detail, each account carries a balance $\mathsf{bl} \in \mathcal{V}$, where $\mathcal{V} \subset \mathcal{M}$; i.e., the domain of values is a subset of the messages that can be committed to using Commit. To create a new account with initial balance $\mathsf{bl} \in \mathcal{V}$, one can

run $(\mathsf{acct}, \mathsf{sk}) \xleftarrow{\$} \mathsf{GenAcct}(1^\kappa, \mathsf{bl})$, which internally runs $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\kappa)$ and $\mathsf{com} \xleftarrow{\$} \mathsf{Commit}_{\mathsf{pk}}(\mathsf{bl})$, sets $\mathsf{acct} = (\mathsf{pk}, \mathsf{com})$, and returns $(\mathsf{acct}, \mathsf{sk})$.

To verify that an account has a certain balance, it is necessary to be able to open a commitment using the secret key corresponding to pk. This also allows the owner of sk to open a commitment or prove statements about the committed message even without knowing the randomness used. We use the notation $\mathsf{VerifyCom}(\mathsf{pk}, \mathsf{com}, \mathsf{sk}, m)$, and require the commitment to be binding also with respect to this function; i.e., that no PPT adversary can output $(\mathsf{pk}, \mathsf{com}, \mathsf{sk}, m, \mathsf{sk}', m')$ with $m \neq m'$ but such that $\mathsf{VerifyCom}(\mathsf{pk}, \mathsf{com}, \mathsf{sk}, m) = \mathsf{VerifyCom}(\mathsf{pk}, \mathsf{com}, \mathsf{sk}', m') = 1$. With this algorithm in place, one can run $0/1 \leftarrow \mathsf{VerifyAcct}(\mathsf{acct}, (\mathsf{sk}, \mathsf{bl}))$, which parses $\mathsf{acct} = (\mathsf{pk}, \mathsf{com})$ and outputs 1 if $\mathsf{VerifyCom}(\mathsf{pk}, \mathsf{com}, (\mathsf{sk}, \mathsf{bl})) = 1$ and $\mathsf{bl} \in \mathcal{V}$ and 0 otherwise.

For an account $\mathsf{acct} = (\mathsf{pk}, \mathsf{com})$, observe that the output of $\mathsf{VerifyAcct}$ is agnostic to updates of the public key; i.e.,

$$\mathsf{VerifyAcct}((\mathsf{pk}, \mathsf{com}), (\mathsf{sk}, \mathsf{bl})) = \mathsf{VerifyAcct}((\mathsf{Update}(\mathsf{pk}), \mathsf{com}), (\mathsf{sk}, \mathsf{bl})).$$

Additionally, $\mathsf{VerifyAcct}$ is agnostic to re-randomizations of the commitment; i.e., $\mathsf{VerifyAcct}((\mathsf{pk}, \mathsf{com}), (\mathsf{sk}, \mathsf{bl})) = \mathsf{VerifyAcct}((\mathsf{pk}, \mathsf{com} \odot \mathsf{Commit}_{\mathsf{pk}}(0; r)), (\mathsf{sk}, \mathsf{bl}))$.

Thanks to these observations, we are able to "update" accounts using the following notation:

- $\{\mathsf{acct}'_i\}_{i=1}^n \xleftarrow{\$} \mathsf{UpdateAcct}(\{\mathsf{acct}_i, v_i\}_{i=1}^n; r_1, r_2)$ takes as input a set of accounts $\mathsf{acct}_i = (\mathsf{pk}_i, \mathsf{com}_i)$ and values v_i such that $|v_i| \in \mathcal{V}$, and outputs a new set of accounts $(\mathsf{acct}'_1, \ldots, \mathsf{acct}'_n)$ where $\mathsf{acct}'_i \xleftarrow{\$} (\mathsf{Update}(\mathsf{pk}; r_1), com \odot \mathsf{Commit}_{\mathsf{pk}}(v_i; r_2))$.
- $0/1 \leftarrow \mathsf{VerifyUpdateAcct}(\{\mathsf{acct}'_i, \mathsf{acct}_i, v_i\}_{i=1}^n; r_1, r_2)$ outputs 1 if $\{\mathsf{acct}'_i\}_{i=1}^n = \mathsf{UpdateAcct}(\{\mathsf{acct}_i, v_i\}_{i=1}^n; r_1, r_2)$ and $|v_i| \in \mathcal{V}$, and 0 otherwise.

4.2 The cryptocurrency setting

Modeling the security of a cryptocurrency is a complex problem, as there are many different actors operating at different layers of the protocol: a user wishing to send some coins creates a transaction, which is then broadcast to their peers in a peer-to-peer network. Those peers in turn perform some cryptographic validation of the transaction, and if satisfied broadcast it to their peers. Eventually, it reaches a miner or validator, who engages in some form of consensus protocol to confirm the transaction into the blockchain.

For the sake of simplicity, we focus solely on the *transaction layer* of a cryptocurrency, and assume network-level or consensus-level attacks are out of scope; i.e., we assume that the system is free from eclipse attacks [17] or other deanonymization attacks that depend on network-level information (such as IP addresses) and that an adversary is not sufficiently powerful to prevent honest transactions from being added to the blockchain or to add malicious transactions of their own.

Rather than use the traditional model of having a sender, in possession of some secret key and a coin, send this coin to a recipient, we instead consider

a set of participants who want to *redistribute wealth* amongst themselves. This means we now model a transaction as taking place amongst a set of participants $\mathcal{P}$ who act as both the senders and the recipients in the transaction, and who each come in with some initial balance $\mathsf{bl}_{0,i}$ and end with some balance $\mathsf{bl}_{1,i}$.

This still captures the traditional model of keeping senders and recipients separate, because for a sender S sending one coin to a recipient R we can use $\mathcal{P} = (\mathsf{pk}_S, \mathsf{pk}_R)$, $\mathsf{bl}_0 = (1, 0)$, and $\mathsf{bl}_1 = (0, 1)$. The natural question, however, is who is required to authorize this transaction; for efficiency reasons we do not want every participant to have to do so, but to ensure that parties cannot simply steal each others' money we do need permission on behalf of the "true" senders. The simple way to provide both these properties is to require authorization only on behalf of the public keys whose associated balance has gone down; i.e., for every $\mathsf{pk}_i \in \mathcal{P}$ such that $\mathsf{bl}_{1,i} - \mathsf{bl}_{0,i} < 0$.

Again, this model fully captures the traditional model of senders and recipients, but crucially makes it easier to reason about cryptocurrencies designed to provide anonymity. More formally, a transaction layer for cryptocurrencies consists of $(\mathsf{Setup}, \mathsf{Trans}, \mathsf{Verify})$, as defined below.

The setup algorithm $\mathsf{state} \xleftarrow{\$} \mathsf{Setup}(1^\kappa, \mathbf{bl})$ generates the initial state of the system. The vector $\mathbf{bl}$ represents the initial balance of the accounts in the system and it must be such that $\mathsf{bl}_i \in \mathcal{V}$ and $\sum_i \mathsf{bl}_i \in \mathcal{V}$. We assume that Setup runs $(\mathsf{acct}_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{GenAcct}(1^\kappa, \mathsf{bl}_i)$ at some point, and that the state contains a set UTXO consisting of all accounts acct_i. All other algorithms take as input the (current) state even when omitted, and the state is updated in ways other than through these algorithms (e.g., by miners producing blocks at the network layer).

To create a transaction, a sender in posession of a secret key sk runs $\mathsf{tx} \leftarrow \mathsf{Trans}(\mathsf{sk}, \mathcal{P}, A, \boldsymbol{v})$.[5] The vector of values $\boldsymbol{v} \in \mathcal{V}$ represents the desired change in balance for each participant, meaning they should end up with $\mathsf{bl}_{1,i} = \mathsf{bl}_{0,i} + v_i$ (where $\mathsf{bl}_{0,i}$ is their initial balance according to state). In creating a transaction, the sender may want to achieve some degree of anonymity, meaning they want to hide the link between their accounts and those of the recipient. To this end, we introduce an anonymity set A, which consists of other accounts used to hide information about the sender. It is important that these accounts are "eligible" in some way (where this depends on the concrete system, but can mean for example that they have not yet spent their contents). If A is not explicitly specified, it is picked at random from the set of eligible accounts. We denote by $\mathsf{tx}[\mathsf{inputs}] = \mathcal{P} \cup A$ the input accounts in a transaction, and by $\mathsf{tx}[\mathsf{outputs}]$ the output accounts.

Finally, $0/\mathsf{state} \leftarrow \mathsf{Verify}(\mathsf{state}, \mathsf{tx})$ checks if a transaction is valid given the current state. If so, it outputs an updated state, and if not it outputs 0.

We say a state is *valid* if it is output by Setup or if it was the output of $\mathsf{Verify}(\mathsf{state}', \mathsf{tx})$ for a valid state' and a transaction tx output by Trans. We say a transaction layer *preserves value* if for any valid $\mathsf{state}' \neq \bot$ derived from a valid

[5] For simplicity we consider a single sender but the notation can easily be generalized to allow for arbitrarily many.

state, $\mathsf{ValueOf}(\mathsf{state.UTXO}) = \mathsf{ValueOf}(\mathsf{state}'.\mathsf{UTXO})$, where $\mathsf{ValueOf}$ computes the number of coins associated with the UTXO set induced by a state.

4.3 Security

Intuitively, an anonymous cryptocurrency should provide *anonymity* for both the sender and the recipient, meaning that even they cannot identify which accounts belong to whom. From an integrity perspective, it is also important to guarantee *theft prevention*, meaning an adversary can transfer value only from accounts for which it knows the secret key (and thereofore the adversary cannot reduce the balance of the honest parties either).

Regardless of the goal, the basic outline of our security experiment is the same, in order to capture the different ways an adversary can interact with honest participants in the system. For example, the adversary can instruct honest participants to engage in transactions, or form arbitrary (i.e., fully adversarial) transactions itself, as long as they are valid.

Intuitively, the adversary begins by specifying the initial balances $\mathbf{bl}$ of all participants in the protocol. We continue this full control by allowing the adversary to direct honest parties to make specific transactions (via `transact` queries), and to inject fully malicious transactions in the system (via `verify` queries). It can also learn the secret key for any account in the system (via `disclose` queries), although here we must be careful to prevent "trivial" attacks resulting from these disclosures in `challenge` queries (in which the adversary specifies two different senders, recipients, and values, and tries to guess between transactions involving them).

These trivial attacks include: (1) the adversary controls the secret key of one or both of the senders; (2) the adversary controls the secret key of a recipient, and (3) the adversary specified a sender who does not have enough funds to complete the specified amount (meaning the output of Trans is $\perp$ in this case but not the other). Formally, our game is defined as follows:

1. $b \xleftarrow{\$} \{0,1\}$;
2. $\mathbf{bl} \xleftarrow{\$} \mathcal{A}(1^\kappa)$;
3. $\mathsf{state} \xleftarrow{\$} \mathsf{Setup}(1^\kappa, \mathbf{bl})$;
4. $b' \xleftarrow{\$} \mathcal{A}^{O(\cdot)}(\mathsf{state})$.

Part of Setup involves running $(\mathsf{acct}_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{Gen}(1^\kappa, \mathsf{bl}_i)$, and we assume that this results in the values $(i, \mathsf{acct}_i, \mathsf{sk}_i, \mathsf{bl}_i)$ being stored in memory available to the oracle.

For several of the oracle queries, there is some *bookkeeping* required to update the keys and balances associated with these records. We define this bookkeeping subroutine with respect to a transaction tx and two sets honest and $\mathsf{corrupt}$ as follows: For every $\mathsf{acct}_j \in \mathsf{tx}[\mathsf{outputs}]$ identify the corresponding $\mathsf{acct}_i \in \mathsf{tx}[\mathsf{inputs}]$ such that $\mathsf{sk}_j = \mathsf{sk}_i$. For every such j, create a new record of the form $(j, \mathsf{acct}_j, \mathsf{sk}_i, \mathsf{bl}_i + v_i')$, where v_i' is either (1) v_i if $i \in \mathcal{P}$ or (2) 0 if $i \in A$. Then, reset the value for every $\mathsf{acct}_i \in \mathsf{tx}[\mathsf{inputs}]$; i.e., save the record $(i, \mathsf{acct}_i, \mathsf{sk}_i, 0)$.

Finally, for every pair (i, j) as above: if $i \in \mathsf{honest}$ add j to honest, else add j to $\mathsf{corrupt}$.

Initialize honest to be the set of all indices i in memory, and $\mathsf{corrupt}$ to be the empty set. The oracle $O(\cdot)$ allows the following queries:

- $(\texttt{disclose}, i)$: If $(i, \mathsf{acct}_i, \mathsf{sk}_i, \mathsf{bl}_i)$ was stored, call J the set of all j such that there is a record $(j, \mathsf{acct}_j, \mathsf{sk}_j, \mathsf{bl}_j)$ with $\mathsf{sk}_i = \mathsf{sk}_j$. Remove i and J from honest, add them to $\mathsf{corrupt}$, and return $(\mathsf{sk}_i, \mathsf{bl}_i, J, \{\mathsf{bl}_j\}_{j \in J})$ to the adversary.
- $(\texttt{transact}, i, \mathcal{P}, A, \boldsymbol{v})$: If $(i, \mathsf{acct}_i, \mathsf{sk}_i, \mathsf{bl}_i)$ was not stored return $\perp$. Otherwise run $\mathsf{tx} \xleftarrow{\$} \mathsf{Trans}(\mathsf{sk}_i, \mathcal{P}, A, \boldsymbol{v})$, and $\mathsf{state}' \leftarrow \mathsf{Verify}(\mathsf{state}, \mathsf{tx})$. If $\mathsf{state}' \neq \perp$ update $\mathsf{state} = \mathsf{state}'$, run the bookkeeping for tx, and return tx.
- $(\texttt{verify}, \mathsf{tx})$: run $\mathsf{state}' \leftarrow \mathsf{Verify}(\mathsf{state}, \mathsf{tx})$. If $\mathsf{state}' \neq \perp$ update $\mathsf{state} = \mathsf{state}'$, run the bookkeeping for tx, and return state'.
- $(\texttt{challenge}, b, (i_0, i_1, j_0, j_1, A, v_0, v_1))$: Let $A_0 = A_1 = A$. If (1) $i_0 \in \mathsf{corrupt}$ or $i_1 \in \mathsf{corrupt}$, (2) $j_0 \in \mathsf{corrupt}$ or $j_1 \in \mathsf{corrupt}$ (except if $j_0 = j_1$ and $v_0 = v_1$), (3) $\mathsf{bl}_{i_0} < v_0$ or $\mathsf{bl}_{i_1} < v_1$, then halt and return 0 (i.e., the adversary lost the game). Otherwise, for $x \in \{0, 1\}$, if $i_0 \neq i_1$ add i_{1-x} to A_x, and if $j_0 \neq j_1$ add j_{1-x} to A_x. Now compute $\mathsf{tx}_x \leftarrow \mathsf{Trans}(\mathsf{sk}_{i_x}, \{\mathsf{acct}_{i_x}, \mathsf{acct}_{j_x}\}, A_x, (-v_x, v_x))$. If $\mathsf{Verify}(\mathsf{state}, \mathsf{tx}_x) = \perp$, then again we say the adversary lost the game. Otherwise, run the bookkeeping for tx_b.
  After a $\texttt{challenge}$ query, the oracle halts; i.e., it outputs $\perp$ as the response to all future queries.

In terms of the concrete security notions discussed above, we say that the adversary wins the *anonymity* game if $b' = b$ and the adversary did not lose the game as the result of some invalid query during the game. We define the *advantage* of the adversary as the probability that the adversary wins subtracted by $1/2$, and say that:

Definition 4. *Anonymity holds if no PPT $\mathcal{A}$ has non-negligible advantage in the* anonymity game.

Note that our definition of anonymity *does not* depend on the size of the anonymity set. Instead, our definition guarantees that, from the point of view of the adversary, a transaction is as likely to have been generated by any of the accounts in the input of the transaction (excluding those that the adversary owns or has corrupted).

We say that the adversary wins the *theft prevention* game if, as a result of any $\texttt{verify}$ query: (1) there exists an account $j \in \mathsf{honest}$ whose balance decreases or (2) the total wealth of the adversary increases; i.e., the sum of the balance of accounts in the set $\mathsf{corrupt}$ increases. (For this property, we could modify the game so that the adversary just outputs $\perp$ and does not need to make any $\texttt{challenge}$ queries). Again, we say that:

Definition 5. *Theft prevention holds if no PPT $\mathcal{A}$ can win the* theft prevention game *with non-negligible probability.*

Note that theft prevention as defined above trivially implies protection from *double spending attacks*.

Finally, we address several seeming limitations in our definition, which have all been introduced for ease of notation and the sake of readability but which are not necessary for our construction. First, our `challenge` queries consider only a single recipient, but could be generalized to handle sets of recipients. Second, we do not consider adversarially generated keys (allowing the adversary only to corrupt honest keys), but we could capture this by changing the second step to allow the adversary to output a list of its own accounts. We would then have to process these accounts into records (in order to keep track of their balances) and restrict which keys could be used for which oracle queries; requiring, e.g., that `transact` only be used for non-adversarial keys. Finally, our current definition has the "IND-CCA1"-style requirement that after the first `challenge` query, the adversary cannot make any other queries. To generalize the definition to allow for this, the oracle would have to keep track of two balances bl_0 and bl_1 for each account after the `challenge` query, where bl_b represents the balance of each account in the "world" in which transaction tx_b was performed. This is necessary to prevent an additional type of trivial attack, in which the adversary made a `transact` query requiring the sender to transfer more than $\min(\mathsf{bl}_0, \mathsf{bl}_1)$: in one of the two worlds this would force the oracle to return $\perp$, which would trivially leak b. Again, all of these limitations were adopted to simplify presentation, but (as should be made clear in the next two sections) our construction would also satisfy the stronger definition relative to a modified game without these restrictions.

5 Our Quisquis Construction

5.1 Overview and intuition

To get a sense of how Quisquis works, let's suppose that Alice wants to anonymously send 5 coins to Bob, and start with a strawman solution in which values are visible in the clear and associated with updatable public keys. To form a transaction, Alice identifies $n-1$ unspent keys with exactly 5 coins associated with them. She then uses these keys, in combination with her own, as the input to the transaction. To form the output keys, she replaces her key with Bob's key, and updates all the other keys. Finally, she forms a ZK proof that she has created the output keys properly; i.e., that she knew the private key for any public keys that were replaced, and that she formed the other output keys by performing a valid update of the input ones. The final transaction consists of the lists of input and output keys, their associated values, and the ZK proof.

This solution allows Alice to use the other input keys as an anonymity set, but only in the restrictive setting in which she has the exact value she wants to send to Bob stored in one of her keys, and she can find multiple other keys with that same value. To address these issues, we first shift to the "re-distribution of wealth" model introduced in Section 4. Rather than replace her own key with Bob's key, she instead adds Bob's key to the list of input keys. If she picks two

others keys pk_0 and pk_1 and forms $\mathcal{P} = (\mathsf{pk}_0, \mathsf{pk}_1, \mathsf{pk}_A, \mathsf{pk}_B)$, then even if she has 9 coins stored in her key she can still send Bob 5 coins by using $\boldsymbol{v} = (0, 0, -5, +5)$.

The problem with this new solution, of course, is that it has no anonymity: anyone can look at $\boldsymbol{v}$ and see who the real senders and recipients are. To hide these values, we switch to using the updatable accounts described in Section 4.1, which means including only commitments to the account balances. The main additional complexity is now in proving that the transaction has been formed correctly, and in particular proving that it does not take money away from anyone other than the real sender. Intuitively, Alice can do this by proving that for every output key, either she knows the secret key for the corresponding input key, or the balance corresponding to that key did not decrease; i.e., the difference between its balance and the balance of its input key is non-negative.

This also supports the case in which Alice wants to consolidate the coins associated with multiple account, as she can include these accounts in both the input and output lists but re-distribute her money so that it all ends up in one of them. This exposes an issue for efficiency, however, which is that once an account has a balance of 0 it is wasteful to leave it in the UTXO set. Thus, to "destroy" an output account, Alice can prove that its committed balance is 0, which signals to others to remove it from the UTXO set.

Conveniently, the technique of proving that a committed value is 0 can also be used to create a new account. This has a positive effect on Bob's anonymity (and communication overhead), as he can now send Alice a regular key once rather than providing a new account every time she wants to send him money. To use this key in the input list, Alice can first update it (to get a new random-looking key), generate a commitment relative to this public key (i.e., generate a new account for it), and prove that its committed balance is 0.

5.2 Transactions in Quisquis

Before describing the algorithms needed to form and verify transactions, we first describe how to instantiate the updatable accounts introduced in Section 4.1. Combining the commitment scheme from Section 2.3 and the UPK scheme from Section 3.2, we get accounts of the form $(\mathsf{pk}_i, \mathsf{com}_i) = ((g_i, g_i^{\mathsf{sk}}), (g_i^r, g^v g_i^{\mathsf{sk}\cdot r}))$. This already gives us most of the properties we need, and guarantees that $|\mathcal{V}| \ll |\mathcal{M}|$ as long as we use $\mathcal{V} = \{0, \ldots, V\}$, where V is an upper bound on the maximum possible number of coins in the system (e.g., the limit of $V = 2.1 \times 10^{15} < 2^{51}$ satoshis in Bitcoin, compared to $\mathcal{M} = \{0, \ldots, p-1\}$ for a 256-bit prime p in the commitment scheme). All it thus remains to show is that the owner of the secret key corresponding to $\mathsf{pk} = (g, h)$ can open the commitment. To do this, we can define the additional algorithm $\mathsf{VerifyCom}(\mathsf{pk}, \mathsf{com}, \mathsf{sk}, v)$ as parsing $\mathsf{com} = (c, d)$ and then checking that $\mathsf{VerifyKP}(\mathsf{pk}, \mathsf{sk}) = 1$ and $d = g^v \cdot c^{\mathsf{sk}}$. For every $(\mathsf{pk}, \mathsf{com})$ there exists exactly one pair (sk, v) for which $\mathsf{VerifyCom}$ outputs 1, so the commitment is unconditionally binding even with respect to this type of opening.

Setup On input 1^κ, Setup returns as state the output of Setup for the UPK scheme, and a list of all current accounts (which may be empty).

Trans As discussed in the overview, Quisquis allows a sender to "re-distribute" their wealth to one or more recipients, by including their accounts in both the input and output lists that comprise the transaction. In what follows we assume that transactions have a fixed number n of both inputs and outputs.

Suppose a transaction is meant to transfer v coins from a sender to a recipient. To hide the identity of the sender and recipient, the Trans algorithm picks an anonymity set A of size $n-2$ uniformly at random from the set of all unspent transaction outputs, and creates a vector $\boldsymbol{v} = (v, -v, 0, \ldots, 0)$. It then updates all these accounts by running UpdateAcct. Intuitively, the properties of updatable accounts guarantee that the individual accounts that are generated as output of UpdateAcct cannot be tied to the input of the function. However, the ordering still reveals the link between the input and outputs. We thus simply present the input and output lists in some canonical (e.g., lexicographical) order. Because the updated keys are distributed uniformly at random, this can be thought of as applying a random permutation ψ to shuffle the updated accounts.

Finally, to ensure that malicious parties cannot steal funds from honest users, the transaction must contain a NIZK proof π that the output of the transaction has been computed following the protocol specification.

To summarize, $\mathsf{tx} \xleftarrow{\$} \mathsf{Trans}((s, \mathsf{sk}_s, \mathsf{bl}_s), \mathcal{P}, A, \boldsymbol{v})$ performs the following steps:

1. First, check that the input is valid by parsing $\mathcal{P} = \{\mathsf{acct}_1, \ldots, \mathsf{acct}_t\}$ and checking that $\mathsf{VerifyAcct}(\mathsf{acct}_s, \mathsf{sk}_s, \mathsf{bl}_s) = 1$. Then check that the vector $\boldsymbol{v}$ satisfies: (1) $\sum_i v_i = 0$, (2) $\forall i \neq s : v_i \in \mathcal{V}$ (i.e., is positive), (3) $-v_s \in \mathcal{V}$ and (4) $\mathsf{bl}_s + v_s \in \mathcal{V}$.
2. Let $\mathsf{inputs} = \mathcal{P} \cup A$ in some canonical order and $\boldsymbol{v}'$ be the permutation of $\boldsymbol{v}$ under the same order. Let $s^*, \mathcal{R}^*, A^*$ denote the indices of the respective accounts of the sender, the recipients, and the anonymity set in this list; i.e., it now holds that $-v'_{s^*} \in \mathcal{V}$, $v'_i \in \mathcal{V}\ \forall i \in \mathcal{R}^*$ and $v'_i = 0\ \forall i \in A^*$.
3. Let outputs be the output of $\mathsf{UpdateAcct}(\mathsf{inputs}, \boldsymbol{v}'; r)$ in some canonical order.
4. Let $\psi : [n] \to [n]$ be the implicit permutation mapping inputs into outputs; i.e., such that accounts inputs_i and $\mathsf{outputs}_{\psi(i)}$ share the same secret key.
5. Form a zero-knowledge proof π of the relation $\mathsf{R}(x, w)$, where $x = (\mathsf{inputs}, \mathsf{outputs})$, $w = (\mathsf{sk}, \mathsf{bl}, \boldsymbol{v'}, r = (r_1, r_2), \psi, s^*, \mathcal{R}^*, A^*)$, and $\mathsf{R}(x, w) = 1$ if for all $i \in [n], j = \psi(i)$, $\mathsf{acct}_i \in \mathsf{inputs}$, $\mathsf{acct}_j \in \mathsf{outputs}$:

$$
\begin{aligned}
&\mathsf{VerifyUpdateAcct}(\mathsf{acct}_j, \mathsf{acct}_i, r, 0) = 1\ \ \forall i \in A^* \\
&\wedge (\mathsf{VerifyUpdateAcct}(\mathsf{acct}_j, \mathsf{acct}_i, r, v'_i) = 1 \wedge v'_i \in \mathcal{V})\ \ \forall i \in \mathcal{R}^* \\
&\wedge \mathsf{VerifyUpdateAcct}(\mathsf{acct}_{\psi(s^*)}, \mathsf{acct}_{s^*}, r, v'_{s^*}) = 1 \\
&\wedge \mathsf{VerifyAcct}(\mathsf{acct}_{\psi(s^*)}, \mathsf{sk}, \mathsf{bl} + v'_{s^*}) = 1 \\
&\wedge \sum_i v'_i = 0.
\end{aligned}
$$

Then the final transaction is $\mathsf{tx} = (\mathsf{inputs}, \mathsf{outputs}, \pi)$.

Due to the way transactions are generated, every address appears at most twice in Quisquis: once when it is created in the output of some transaction, and once when it appears as the input of some other transaction (regardless of whether it is the real sender or just an account added for anonymity). In particular, unlike in Monero the same account cannot be used as part of the anonymity set for two different transactions, since it will have been updated in the meantime and thus replaced in the UTXO set.

Verify The Verify algorithm ensures the validity of a transaction by checking that all the accounts in $\mathsf{tx}[\mathsf{inputs}]$ are considered unspent in the current state, and by running the verification algorithm for the NIZK argument.

Additionally, upon receiving a transaction in which one of their accounts was included in $\mathsf{tx}[\mathsf{inputs}]$, it is necessary for users to identify which (if any) of the accounts in outputs belongs to them. (If no such account appears in the inputs then they do not need to process the transaction further.) To do this, they first identify the secret key sk corresponding to their account in $\mathsf{tx}[\mathsf{inputs}]$. They then go through every $(\mathsf{pk}, \mathsf{com}) \in \mathsf{tx}[\mathsf{outputs}]$ and run $b \leftarrow \mathsf{VerifyKP}(\mathsf{pk}, \mathsf{sk})$. If $b = 1$, they replace their own existing record of that account with $\mathsf{acct} = (\mathsf{pk}, \mathsf{com})$.

The user should then figure out whether their address was an actual recipient of the transaction or whether it had only be used as part of the anonymity set. They can start by running $\mathsf{VerifyAcct}(\mathsf{acct}, \mathsf{sk}, \mathsf{bl}) = 1$, where bl was their balance before the transaction; if this passes, then their account was used as part of the anonymity set so their balance is unchanged. Otherwise, they need to find out the value v by which their balance was increased. For simplicity here we assume that the values v are small enough, say 32 bits (for comparison, the total number of satoshis that will ever exist is 2^{51}), so that computing v from g^v is computationally easy, and therefore the user can "brute force" their new account. Again, this is necessary only in the case of transactions that include their accounts as part of the input (and transactions creating new accounts); no other transactions can change the balance of a user's account.

The design can be easily extended for larger values of v: for instance, we can (1) require that senders communicate the value v_i to their recipients off-chain or (2) append to the transaction an encryption of v_i under the public key of the receiver, together with a proof that the encryption contains the correct value (using, e.g., a similar approach to Zether [10]).

Creating and removing accounts The described scheme above supports the basic functionality of making anonymous payments, but as described in the overview in Section 5.1 it is possible to improve on the efficiency of this basic protocol. In particular, newly created accounts and fully spent accounts both have a (provable) balance of 0. Allowing users to create new accounts improves the overall communication overhead and anonymity of the system, since users can send one long-term key to potential senders rather than a new account every time (which would also reveal to the sender the transaction in which this

account was created). Allowing users to destroy empty accounts reduces the storage overhead of the system, since other users do not have to keep track of accounts that have no contents left to spend.

We denote the respective algorithms used to perform creating and removing accounts by $\mathsf{CreateAcct}$ and $\mathsf{DestroyAcct}$.

- $\mathsf{acct} = (\mathsf{pk}', \mathsf{com}), \pi) \xleftarrow{\$} \mathsf{CreateAcct}(\mathsf{pk})$ is such that $\mathsf{pk}' \in [\mathsf{Update}(\mathsf{pk})]$, $\mathsf{com} = \mathsf{Commit}_{\mathsf{pk}'}(0; r)$ for some r, and π is a ZK proof for the relation $R(x, w)$, where $x = (\mathsf{pk}', \mathsf{com})$, $w = r$, and $R(x, w) = 1$ if $\mathsf{com} = \mathsf{Commit}_{\mathsf{pk}'}(0; r)$. Again, this algorithm can be run by anyone in possession of a public key for a user, which allows senders to send money to recipients without requiring their participation.
- $\pi \xleftarrow{\$} \mathsf{DestroyAcct}(\mathsf{sk}, \mathsf{acct})$ is such that π is a ZK proof for the relation $R(x, w)$, where $x = \mathsf{acct}$, $w = \mathsf{sk}$, and $R(x, w) = 1$ if $\mathsf{VerifyAcct}(\mathsf{acct}, (\mathsf{sk}, 0)) = 1$.

Proofs of this type can optionally be included in transactions, and have the effect that upon verification users remove the corresponding acct from the list of active accounts. The zero-knowledge proofs involved in both $\mathsf{CreateAcct}$ and $\mathsf{DestroyAcct}$ are standard proofs of relations between discrete logarithms, so we do not include descriptions of them here.

Mining fees As currently described, Quisquis does not provide any incentives for miners to include transactions, due to the lack of fees. More crucially, it assumes the total balance of the system is fixed during Setup, so does not capture the ability to mine new coins.

To add transaction fees to the Trans algorithm, we can add the fee f as an input and change the requirement on the vector $\boldsymbol{v}$ to be $\mathsf{f} + \sum_i v_i = 0$. Assuming the fee is public (as it is in other privacy-enhanced currencies like Zcash), this does not add any complexity to the zero-knowledge proof. So, let $(\mathsf{tx}_1, \mathsf{f}_1, \ldots, \mathsf{tx}_m, \mathsf{f}_m)$ be a set of transaction that a miner wants to add to the blockchain. To collect the fees and add a block reward rwd, the miner can simply generate a new account $(\mathsf{acct}, \mathsf{sk}) \leftarrow \mathsf{GenAcct}(1^\kappa, \mathsf{rwd} + \sum_i \mathsf{f}_i)$ and a proof that the balance of this account is equal to the block reward plus the sum of fees present in the block. The initial balance is thus public, but as soon as it is used in any further transaction the usual anonymity guarantee is preserved.

Concurrent transactions Although it is somewhat out of scope of the core cryptographic design, we briefly discuss here how a cryptocurrency based on the Quisquis design might deal with concurrent transactions, in which two users both try to use the same account in their anonymity set at roughly the same time (and there is thus a non-empty intersection between the two $\mathsf{tx}[\mathsf{inputs}]$). Since each address can appear only once as input in Quisquis, this requires at least one of the two transactions to be rejected by the system. We propose here two simple approaches for dealing with this, although the probability of having such a collision could be quite low (depending on system parameters such as the frequency of transactions, the network latency, etc.).

The first heuristic is "reject and wait": if two conflicting transactions are received in the same time period, they are both rejected and the users are instructed to wait and attempt the transaction again. The second heuristic is "first come first serve": the transaction that is first received is approved and the second one is rejected. The sender of the second transaction is free to send a new transaction as soon as they want.

The first proposal might be better for anonymity, since – thanks to the waiting time – many (or even all) addresses in the original anonymity sets might have left the UTXO set (after being chosen as part of the anonymity set of other transactions) and been replaced by new random-looking accounts. The second proposal ensures lower latency, but might reduce the privacy of the second transaction: if all accounts in the intersection were part of the anonymity set, the sender might simply replace those and effectively run a transaction with a smaller anonymity set. On the other hand, if any actual sender (or receiver) of the transaction disappeared from the UTXO set, this would require the sender to use the new version of those accounts that was created in the $\mathsf{tx}[\mathsf{outputs}]$ of the approved transaction.

5.3 Proofs of security

Proof of anonymity The full proof of anonymity of Quisquis is given in the full version of the paper [13], but we sketch the main intuition here. Informally, we claim that any $\mathcal{A}$ that can determine b from tx can be used to break either the indistinguishability property of UPK, the hiding property of Commit, or the zero-knowledge property of the NIZK. That is, any $\mathcal{A}$ that can determine b can distinguish between tx_0 and tx_1. Since $\mathsf{tx}_0[\mathsf{inputs}] = \mathsf{tx}_1[\mathsf{inputs}]$ (by inspection), it must be the case that the adversary either distinguishes between the transactions based on the proof π or the set of accounts in $\mathsf{outputs}$. The first option is ruled out due to the zero-knowledge property of π. To see why the adversary cannot distinguish based on $\mathsf{outputs}$, note that in both cases $\mathsf{outputs}$ is obtained by updating all the accounts in inputs, and the only differences between $\mathsf{outputs}_0$ and $\mathsf{outputs}_1$ are (1) the amounts by which the accounts have been increased or decreased and (2) which accounts are included in $\mathcal{P}$ and which are included in A. Since the amounts are only present in committed form, we conclude that the adversary cannot distinguish based on (1) due to the hiding property of the commitment. Since all the accounts are updated (both those in $\mathcal{P}$ and in A), and they are then randomly permuted, the adversary cannot distinguish based on (2) either.

Proof of theft prevention To win the theft prevention game, the adversary needs to submit a transaction tx that increases the total balance corresponding to the corrupted accounts or decreases the balance for the honest accounts. Intuitively, this can happen only in two ways: (1) if the adversary manages to transfer value from an honest account (to a corrupted account or to an "unspendable" account) and (2), if the adversary manages to transfer a value higher than the

balance of a corrupted account. Due to the extractability of the zero-knowledge proof system, we know that the tx will be accepted only if the adversary has a valid witness. This means that: in case (1) we can use the adversary to compute a secret key sk for an honest account (thus breaking the unforgeability property of UPK); in case (2) we can use the adversary to compute an opening of a commitment with a balance different from the real one, thus breaking the binding property of the commitment scheme.

6 Instantiating the Zero-knowledge Proof

In this section we will instantiate the zero-knowledge proof that inputs and $\mathsf{outputs}$ satisfy the relation described in the Trans algorithm. First consider the simplified case where Trans does not do any lexicographic ordering or any type of permutation of the public keys. Then a prover essentially has to prove that (1) accounts in $\mathsf{outputs}$ are proper updates of inputs, (2) the updates satisfy preservation of value, (3) balances in the recipient accounts do not decrease, and (4) the sender account in $\mathsf{outputs}$ contain a balance in $\mathcal{V}$. Properties (3) and (4) require a tool called *range proofs*. We choose to use the most efficient implementation of range proofs, which is the Bulletproofs of Bootle et al. [11]. The main requirement to use Bulletproofs is to have a public commitment key (g, h) such that the DL relation between them is unknown.

We now explain how to check properties (1) and (2). Let inputs have balances $\mathbf{bl}$, and $\mathsf{outputs}$ have balances $\mathbf{bl}'$. Let $v_i = \mathsf{bl}'_i - \mathsf{bl}_i$ be the change in value from inputs to $\mathsf{outputs}$. Additionally, let the sender be inputs_1 and the recipients be $\mathsf{inputs}_2, \ldots, \mathsf{inputs}_t$.

To be able to easily verify that the update is done correctly, the prover creates accounts $\mathbf{acct}_{\boldsymbol{\delta}}$ that contain values $\boldsymbol{v}$. Since we need preservation of value, there needs to be a way to verify that $\sum_i v_i = 0$. To do this, recall that we can regard an account acct as two parts (pk, com) where pk is a UPK and com is a commitment to the balance. The idea is then to use the homomorphic property of the commitment. This is done by first creating $\mathbf{acct}_{\boldsymbol{\epsilon}}$ that also contains values $\boldsymbol{v}$ but where $\mathsf{pk}_{\epsilon,i} = (g, h)$ for all i. (Hence $com_{\delta,i}$ and $com_{\epsilon,i}$ can be seen as two commitments of the same value under different public keys $\mathsf{pk}_{\delta,i}$ and $\mathsf{pk}_{\epsilon,i}$.) Then $\sum_i v_i = 0$ iff $\prod_i com_{\epsilon,i}$ is a commitment of 0 under public key (g, h). The values $\mathsf{acct}_{\epsilon,2}, \ldots, \mathsf{acct}_{\epsilon,t}$ will also be used to prove that the recipient's increase in values $v_2, \ldots, v_t$ are in $\mathcal{V}$.

Note however that the simplified case does not hide where the sender and recipient accounts are in both inputs and $\mathsf{outputs}$. To get full anonymity, the input accounts are shuffled into a list inputs' before the updates, then shuffled again after the updates to get the output accounts in an arbitrary order. The first shuffle uses a permutation so that the sender is always in position inputs'_1 and the recipients are $\mathsf{inputs}'_2, \ldots, \mathsf{inputs}'_t$, while the second shuffle uses a random permutation. This will help making the proof more efficient (otherwise, for every account in the transaction, we would have to prove the disjunction of the conditions for the sender and the recipients).

Function	Description
$\mathsf{CreateDelta}$	Creates a set of accounts that contains the difference (say v_i) between balances in the input and output accounts, and another set of accounts that also contains v_i but all with the global public key (g, h).
$\mathsf{VerifyDelta}$	Verifies that accounts created using $\mathsf{CreateDelta}$ are consistent.
$\mathsf{VerifyNonNegative}$	Verifies that an account contains a balances in $\mathcal{V}$.
$\mathsf{UpdateDelta}$	Updates the input accounts by v_i, but with left half unchanged.
$\mathsf{VerifyUD}$	Verifies that $\mathsf{UpdateDelta}$ was performed correctly.

Table 1: Additional functions to perform a transaction.

6.1 The auxiliary functions

To realize the ideas above, we require some auxiliary functions defined as follows (see Table 1 for a summary).

$\mathsf{CreateDelta}(\{\mathsf{acct}_i\}_{i=1}^n, \{v_i\}_{i=1}^n)$**:** Parse $\mathsf{acct}_i = (\mathsf{pk}_i, com_i)$. Sample $r_1, \ldots, r_{n-1} \xleftarrow{\$} \mathbb{F}_p$ and set $r_n = -\sum_{i=1}^{n-1} r_i$. Set $\mathsf{acct}_{\delta,i} = (\mathsf{pk}_i, \mathsf{Commit}_{\mathsf{pk}_i}(v_i; r_i))$. Set $\mathsf{acct}_{\epsilon,i} = (g, h, \mathsf{Commit}_{(g,h)}(v_i; r_i))$. Output $(\{\mathsf{acct}_{\delta,i}\}_{i=1}^n, \{\mathsf{acct}_{\epsilon,i}\}_{i=1}^n), \boldsymbol{r})$.
$\mathsf{VerifyDelta}(\{\mathsf{acct}_{\delta,i}\}_{i=1}^n, \{\mathsf{acct}_{\epsilon,i}\}_{i=1}^n, \boldsymbol{v}, \boldsymbol{r})$**:** Parse $\mathsf{acct}_{\delta,i} = (\mathsf{pk}_i, com_i)$ and $\mathsf{acct}_{\epsilon,i} = (\mathsf{pk}'_i, com'_i)$. If $\prod_{i=1}^n com'_i = (1, 1)$ and for all i, $com_i = \mathsf{Commit}_{\mathsf{pk}_i}(v_i; r_i) \wedge \mathsf{acct}_{\epsilon,i} = (g, h, \mathsf{Commit}_{(g,h)}(v_i; r_i))$ output 1. Else output 0.
$\mathsf{VerifyNonNegative}(\mathsf{acct}, v, r)$**:** If $\mathsf{acct} = (g, h, g^r, g^v h^r) \wedge v \in \mathcal{V}$ output 1. Else output 0.
$\mathsf{UpdateDelta}(\{\mathsf{acct}_i\}_{i=1}^n, \{\mathsf{acct}_{\delta,i}\}_{i=1}^n)$**:** Parse $\mathsf{acct}_i = (\mathsf{pk}_i, com_i)$ and $\mathsf{acct}_{\delta,i} = (\mathsf{pk}'_i, com'_i)$. If $\mathsf{pk}_i = \mathsf{pk}'_i$ for all i output [6] $\{(\mathsf{pk}_i, com_i \cdot com'_i)\}_{i=1}^n$, else output $\perp$.
$\mathsf{VerifyUD}(\mathsf{acct}, \mathsf{acct}', \mathsf{acct}_\delta)$**:** Parse $\mathsf{acct} = (\mathsf{pk}, com)$, $\mathsf{acct}' = (\mathsf{pk}', com')$ and $\mathsf{acct}_\delta = (\mathsf{pk}_\delta, com_\delta)$. Check that $\mathsf{pk} = \mathsf{pk}' = \mathsf{pk}_\delta \wedge com' = com \cdot com_\delta$.

6.2 The proof system

Let (g, h) be a *global public key* output by the Setup algorithm, such that the DL relation between them is unknown. The NIZK system $\mathsf{NIZK.Prove}(x, w)$ performs the following:

1. Parse $x = (\mathsf{inputs}, \mathsf{outputs})$, $w = (\mathsf{sk}, \mathsf{bl}, \boldsymbol{v}, (\boldsymbol{u}_1, \boldsymbol{u}_2), \psi : [n] \to [n], s^*, \mathcal{R}^*, A^*)$. If $\mathsf{R}(x, w) = 0$ abort;
2. Let ψ_1 be a permutation such that $\psi_1(s^*) = 1$, $\psi_1(\mathcal{R}^*) = [2, t]$ and $\psi_1(A^*) = [t + 1, n]$;
3. Sample $\boldsymbol{\tau_1} \xleftarrow{\$} \mathbb{F}_p^n$, $\rho_1 \xleftarrow{\$} \mathbb{F}_p$;
4. Set $\mathsf{inputs}' = \mathsf{UpdateAcct}(\{\mathsf{inputs}_{\psi_1(i)}, 0\}_i; (\boldsymbol{\tau_1}, \rho_1))$;
5. Set the vector $\boldsymbol{v'}$ such that $v'_i = v_{\psi_1(i)}$;

[6] Note that if $\mathsf{acct} = (\mathsf{pk}, com)$ and $\mathsf{acct}_\delta = (\mathsf{pk}, \mathsf{Commit}_{\mathsf{pk}}(v; r)))$, then $\mathsf{UpdateDelta}(\mathsf{acct}, \mathsf{acct}_\delta) = \mathsf{UpdateAcct}(\mathsf{acct}, v; 1, r)$.

6. Set $(\{\mathsf{acct}_{\delta,i}\}, \{\mathsf{acct}_{\epsilon,i}\}, \boldsymbol{r}) \xleftarrow{\$} \mathsf{CreateDelta}(\mathsf{inputs}', \boldsymbol{v'})$;
7. Update $\mathsf{outputs}' \leftarrow \mathsf{UpdateDelta}(\mathsf{inputs}', \{\mathsf{acct}_{\delta,i}\})$;
8. Let $\psi_2 = \psi_1^{-1} \circ \psi$, $\tau_{2,i} = \frac{u_{1,i}}{\tau_{1,\psi_2(i)}}$ and $\rho_2 = \frac{u_{2,i-\rho_1}}{\tau_{1,\psi_2(i)}} - r_{\psi_2(i)}$; (So that $\psi_1 \circ \psi_2 = \psi$ and $\mathsf{outputs} = \mathsf{UpdateAcct}(\{\mathsf{outputs}'_{\psi_2(i)}, 0\}_i; \boldsymbol{\tau_2}, \rho_2)\}$).
9. Generate a ZK proof $\pi = (\mathsf{inputs}', \mathsf{outputs}', \mathsf{acct}_\delta, \mathsf{acct}_\epsilon, \pi_1, \pi_2, \pi_3)$ for the relation $\mathsf{R}_1 \wedge \mathsf{R}_2 \wedge \mathsf{R}_3$, where

$$
\begin{aligned}
\mathsf{R}_1 = \{&(\mathsf{inputs}, \mathsf{inputs}', (\psi_1, \boldsymbol{\tau}_1, \rho_1)) \mid \\
&\mathsf{VerifyUpdateAcct}(\{\mathsf{inputs}'_i, \mathsf{inputs}_{\psi_1(i)}, 0\}_i; \boldsymbol{\tau}_1, \rho_1) = 1\}, \\
\mathsf{R}_2 = \{&((\mathsf{inputs}', \mathsf{outputs}', \mathsf{acct}_\delta, \mathsf{acct}_\epsilon), (\mathsf{sk}, \mathsf{bl}_{s^*}, \boldsymbol{v'}, \boldsymbol{r})) \mid \\
&\mathsf{VerifyUD}(\mathsf{inputs}'_i, \mathsf{outputs}'_i, \mathsf{acct}_{\delta,i}) = 1 \quad \forall i \\
&\wedge \mathsf{VerifyUpdateAcct}(\mathsf{inputs}'_i, \mathsf{outputs}'_i, 0; 1, r_i) = 1 \ \ \forall i \in [t+1, n] \\
&\wedge \mathsf{VerifyNonNegative}(\mathsf{acct}_{\epsilon,i}, v'_i, r_i) = 1 \quad \forall i \in [2, t] \\
&\wedge \mathsf{VerifyAcct}(\mathsf{outputs}'_1, (\mathsf{sk}, \mathsf{bl}_{s^*} + v'_1)) = 1 \\
&\wedge \mathsf{VerifyDelta}(\{\mathsf{acct}_{\delta,i}\}, \{\mathsf{acct}_{\epsilon,i}\}, \boldsymbol{v'}, \boldsymbol{r}) = 1\}, \\
\mathsf{R}_3 = \{&(\mathsf{outputs}', \mathsf{outputs}, (\psi_2, \boldsymbol{\tau}_2, \rho_2)) \mid \\
&\mathsf{VerifyUpdateAcct}(\{\mathsf{outputs}_i, \mathsf{outputs}'_{\psi_2(i)}, 0\}_i; \boldsymbol{\tau_2}, \rho_2) = 1\}.
\end{aligned}
$$

Instantiating the shuffle argument The zero-knowledge argument of knowledge for R_1 and R_3 uses a shuffle argument $\Sigma_1 = \Sigma_{sh}(\psi_1)$, which is required to prove that inputs' is a correct shuffle of inputs and $\Sigma_3 = \Sigma_{sh}(\psi_2)$, which is required to prove that $\mathsf{outputs}$ is a correct shuffle of $\mathsf{outputs}'$.

Let (g, h) be the *global public key* output by the Setup algorithm, and let $\mathsf{ck} = (\bar{g}, \bar{g}_1, \ldots, \bar{g}_n)$ be the commitment key of the extended Pedersen commitment scheme $\mathsf{XCom}_{\mathsf{ck}}(\boldsymbol{a}; r) = \bar{g}^r \prod_i \bar{g}_i^{a_i}$. In the following, we just write this as $\mathsf{XCom}(\boldsymbol{a}; r)$.

Recall that an update of $\mathsf{acct}_i = (\mathsf{pk}_i, com_i)$ using randomness (τ_i, ρ_i) is $\mathsf{acct}'_j = (\mathsf{pk}'_j, com'_j) = (\mathsf{pk}_i^{\tau_i}, com_i \cdot \mathsf{pk}_i^{\rho_i})$. The public key pk_i is updated by just exponentiation, so its proof of correct shuffle is a slight modification of the Bayer-Groth [5] shuffle. For this we define the generalized commitments to a matrix $A = (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) \in \mathbb{F}_p^{m \times n}$ to be the commitments of its n columns. That is, $\mathsf{XCom}(A; \boldsymbol{r}) = (\mathsf{XCom}(\boldsymbol{a}_1; r_1), \ldots, \mathsf{XCom}(\boldsymbol{a}_n; r_n))$. Additionally, a Hadamard product of matrices A and B, denoted $C = A \circ B$, is simply the matrix such that $c_{ij} = a_{ij} b_{ij}$.

The shuffle argument uses the following sub-arguments [5]:

- The multi-exponentiation argument, π_{MExp}: Given a vector $\boldsymbol{C'}$ and a commitment $\boldsymbol{C'_B}$, the prover shows knowledge of a witness $w = (\boldsymbol{b'}, \boldsymbol{r})$ such that $\boldsymbol{C'_B} = \mathsf{XCom}(\boldsymbol{b'}; \boldsymbol{r})$, and for a fixed $T \in \mathbb{G}$, it holds that $\prod_{i=1}^n C_i'^{b'_i} = T$. In the shuffle argument, $T = \prod_{i=1}^n C_i^{x^i}$, where x is the second message of the protocol.
- The product argument, π_{prod}: Given a commitment $\boldsymbol{C_A}$, the prover shows knowledge of a witness $w = (\boldsymbol{a}, \boldsymbol{r})$ such that $\boldsymbol{C_A} = \mathsf{XCom}(\boldsymbol{a}; \boldsymbol{r})$, and for a

fixed $t \in \mathbb{F}_p$, it holds that $\prod_{i=1}^{n} a_i = t$. In the shuffle argument, $t = \prod_{i=1}^{n}(y \cdot i + x^i - z)$, where (y, z) is the fourth message of the protocol.
- The Hadamard product argument, π_{Had}: Given extended Pedersen commitments $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$, the prover shows knowledge of an opening to vectors $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ such that $\boldsymbol{a} \circ \boldsymbol{b} = \boldsymbol{c}$.

A proof of correct shuffle for com_i uses the following invariant, provided we set all ρ_i to be the same value ρ. Let $\mathsf{pk}_i = (g_i, h_i)$, $(G, H) = (\prod_{i=1}^{N} g_i^{X^i}, \prod_{i=1}^{N} h_i^{X^i})$ and $(G', H') = (G^\rho, H^\rho)$. For a random variable X, $\prod_{i=1}^{N}(com'_{\psi(i)})^{X^{\psi(i)}} = \prod_{i=1}^{N} com_i^{X^i} \cdot (G', H')$. Hence we can also use a multi-exponentiation argument (this time with $T = \prod_{i=1}^{n} com_i^{x^i} \cdot (G', H')$), with an additional proof of correct update Σ_{vu} for the tuple (G, H, G', H').

Note that using the same $\rho_i = \rho$ to update the com_i is secure under the indistinguishability of UPK and computational hiding of Commit. (An adversary that can distinguish if two accounts are updated using the same ρ, can be used to break DDH.)

The full shuffle argument Σ_{sh} is shown in Figure 1.

The following lemma is similar to the one in [5], and the full proof is deferred to the full version of the paper [13].

Lemma 2. *Let the product argument π_{prod}, the Hadamard product argument π_{Had}, the verify update argument π_{vu} and the multi-exponentiation argument π_{MExp} be public-coin SHVZK arguments of knowledge. Then Σ_{sh} is a public-coin SHVZK argument of knowledge of $(\psi, \boldsymbol{\tau}, \rho)$ such that $(\mathsf{pk}'_i, com'_i) = (\mathsf{pk}^{\tau_i}_{\psi(i)}, com_{\psi(i)} \cdot \mathsf{pk}^{\rho}_{\psi(i)})$.*

Instantiating the other sub-arguments To prove statements related to the function $\mathsf{VerifyNonNegative}$ we use Bulletproofs, which we denote by the argument $\Sigma_{range}(\mathsf{acct}, v, r)$. $\mathsf{VerifyAcct}$ also uses Bulletproofs but since the sender may not know the randomness used to open his commitment (for example, if the account was previously updated by someone else), we need a separate argument $\Sigma_{range,sk}(\mathsf{acct}, v, \mathsf{sk})$. This argument first creates acct_ϵ, proves knowledge of (v, r) such that $\mathsf{acct}_\epsilon = (g, h, \mathsf{Commit}_{(g,h)}(v; r))$, then calls $\Sigma_{range}(\mathsf{acct}_\epsilon, v, r)$.

The zero-knowledge argument of knowledge Σ_2 for the non-shuffle parts consists of the following sub-protocols:

1. Σ_{vud}: trivial check of $\mathsf{VerifyUD}$.
2. Σ_{Com}: prover shows knowledge of $\boldsymbol{v}', \boldsymbol{r}$ such that $\mathsf{VerifyDelta}(\{\mathsf{acct}_{\delta,i}\}), \{\mathsf{acct}_{\epsilon,i}\}, \boldsymbol{v}', \boldsymbol{r}) = 1$.
3. Σ^i_{zero}: prover shows knowledge of r_i such that $\mathsf{VerifyUpdateAcct}(\mathsf{inputs}'_i, \mathsf{outputs}'_i, 0, (1, r_i)) = 1$.
4. Σ_{NN}: $(\bigwedge_{i=2}^{t+1} \Sigma_{range}(\mathsf{acct}_{\delta,i}, v'_i, r_i)) \wedge (\bigwedge_{i=t+2}^{n} \Sigma^i_{zero})$.

Hence $\Sigma_2 = \Sigma_{\mathsf{vud}} \wedge \Sigma_{Com} \wedge \Sigma_{NN} \wedge \Sigma_{range,sk}(\mathsf{outputs}'_1, \mathsf{bl}_{s^*} + v'_1, \mathsf{sk})$. The proof of the next lemma follows from the properties of AND-proofs, and is thus omitted.

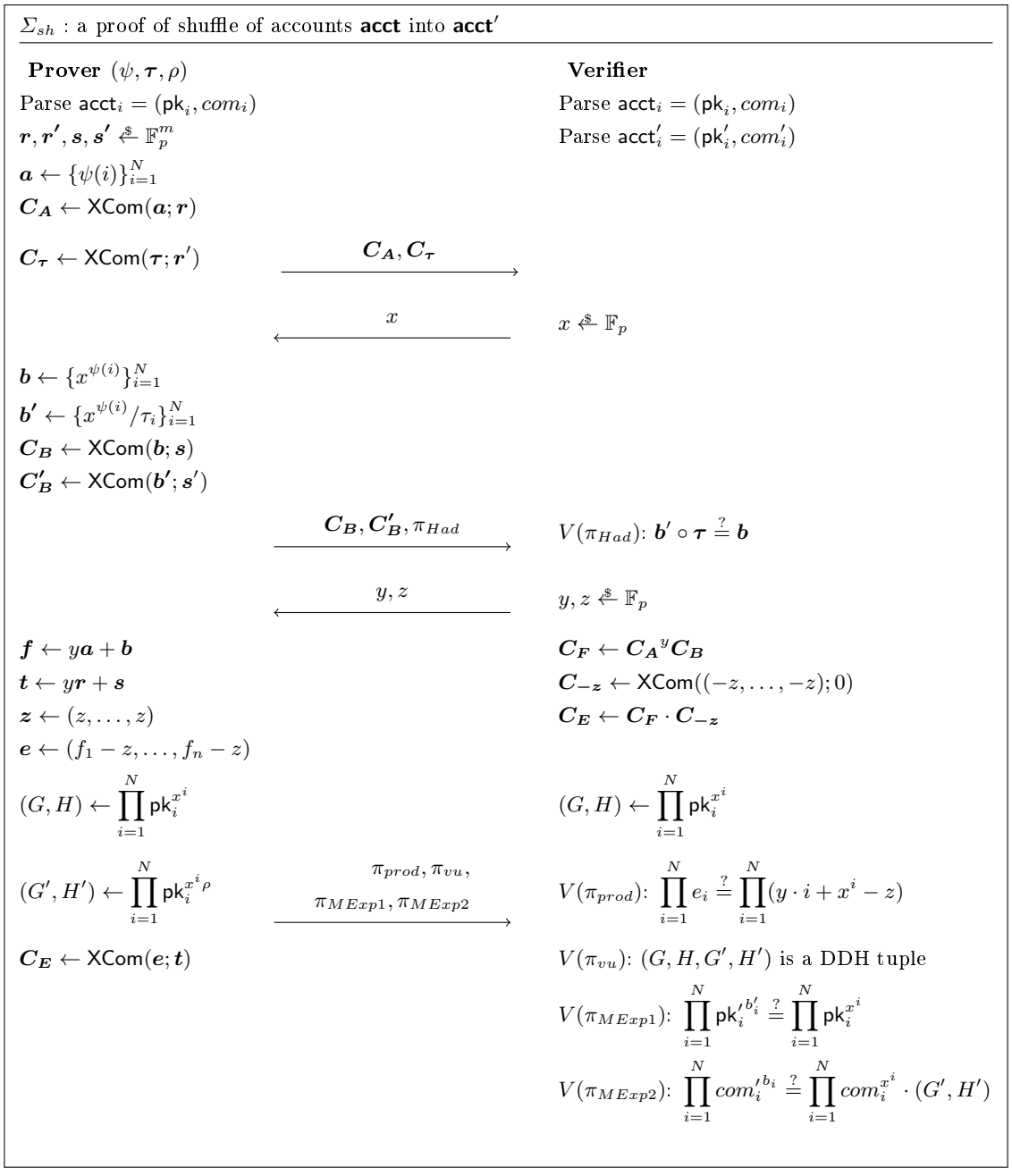

Fig. 1: The full shuffle argument Σ_{sh}. Here $V(\pi) : x$ means that statement x should be verified using the argument π.

Lemma 3. *Σ_2 is a public-coin SHVZK argument of knowledge of the relation R_2.*

The full SHVZK argument of knowledge for Quisquis is then $\Sigma := \Sigma_1 \wedge \Sigma_2 \wedge \Sigma_3$. The proof of the following theorem is deferred to the full version of the paper [13].

Theorem 1. *Σ is a public-coin SHVZK argument of knowledge of the relation $\mathsf{R}(x, w)$ defined in Section 5.2.*

7 Performance

We now describe a prototype implementation of Quisquis, written in roughly 2000 lines of Go and interfaced with an existing Rust implementation for producing Bulletproofs,[7] to demonstrate that it is competitive in terms of both communication and computational costs.

As a reminder, transactions in Quisquis are made up of: (1) input and output account lists $\mathsf{tx}[\mathsf{inputs}]$ and $\mathsf{tx}[\mathsf{outputs}]$, (2) intermediate account lists inputs', $\mathsf{outputs}'$, $\{\mathsf{acct}_{\delta,i}\}$ and $\{\mathsf{acct}_{\epsilon,i}\}$, and (3) a NIZK $\Sigma = \Sigma_1 \wedge \Sigma_2 \wedge \Sigma_3$, with Σ_1 proving that a permutation has updated each of the accounts in $\mathsf{tx}[\mathsf{inputs}]$ to the corresponding intermediate account, and Σ_3 similarly proving that $\mathsf{tx}[\mathsf{outputs}]$ is an updated permutation of the set of intermediate accounts. Σ_2 is a combination of multiple NIZKs to prove that a number of conditions on the accounts and their balances are satisfied.

In our UPK construction, an account consists of four elements from $\mathbb{G}$. Using an elliptic curve at the 128-bit security level and with compressed points (i.e., in which points are represented just by the x-coordinate and the sign of the y-coordinate), each group element requires 33 bytes of communication (32 bytes for the x-coordinate and 1 bit for the sign), and each field element is 32 bytes.

The lists of accounts dominate the proof size for large anonymity set sizes. Since (1) and (2) are both lists of accounts of size n, and each account consists of 4 group elements, each transaction contains $24n$ group elements, or $792n$ bytes.

For Σ_1 and Σ_3, the Bayer-Groth shuffle that we use in Section 6 is parameterizable, and we have chosen the options that minimize communication. We thus implement the shuffle with communication complexity that grows proportionally to the square root of the size of the anonymity set. This means that it consists of $11\sqrt{n} + 7$ group elements and $5\sqrt{n} + 12$ field elements. Concretely then, each of these two proofs requires $352\sqrt{n} + 224$ bytes for group elements, and $160\sqrt{n} + 384$ for field elements, for a total of $512\sqrt{n} + 608$ bytes each, giving $1024\sqrt{n} + 1216$ bytes in total.

Bulletproofs can be produced and verified in batches, leading to the resulting proofs growing only logarithmically with the size of the batch, rather than linearly. These proofs are then most efficient when batched in powers of two, and so we have chosen the anonymity set size to be both square and a power of two

[7] `https://github.com/dalek-cryptography/bulletproofs`

$\|A\|$	Gen. (ms)	Verif. (ms)	Proof size	Proof size (bytes)
4	124±4%	25.6±3%	$122\mathbb{G} + 83\mathbb{F}_p$	6528
16	471±4%	71.6±3%	$244\mathbb{G} + 175\mathbb{F}_p$	13,408
64	2110±3%	251 ±4%	$624\mathbb{G} + 503\mathbb{F}_p$	36,064

Table 2: The computation and communication complexity of the NIZKs in Quisquis, reported with various anonymity set sizes, and averaged over 20 seconds of runs.

below. However, anonymity set sizes are not limited to these values. The proof size when using Bulletproofs for range proofs also grows depending on the size of the range, and this also must be a power of two. We have chosen $K = 64$ for $\mathcal{V} = [0, 2^K - 1]$.

Besides the $16n$ group elements used for lists of intermediate accounts, Σ_2 requires $6n + 38 + 2\log_2(t)$ group elements, and $6n + 15$ field elements. The total proof size is then $6n + 22\sqrt{n} + 52 + 2\log_2(t)$ group elements and $6n + 10\sqrt{n} + 39$ field elements.

Concretely, Table 2 shows the time to generate and verify the NIZK arguments in Quisquis with certain anonymity set sizes. These benchmarks were collected on a laptop with an Intel Core i7 2.8GHz CPU and 16GB of RAM, and demonstrate the overall practicality of Quisquis: proofs take 2.1 seconds to generate and comprise 36 kB in the worst case in which the size of the anonymity set is 64. We stress, however, that we do not expect users to end up using anonymity sets of anywhere near this size in a practical deployment of Quisquis, although we leave it as an interesting open problem to understand the effect different set sizes would have on the level of anonymity achieved by users.

8 Related Work and Comparisons

We provide a broad overview of related work, in terms of tumblers designed to provide anonymity, as well as a detailed comparison with the two solutions, Zcash and Monero, that are most related to our own. The results of our comparison are summarized in Table 3. The benchmarks in Table 3 were collected using a server with an Intel Core i7 3.5GHz CPU and 32GB of RAM, due to the Zcash client performing best when used with Linux, and due to the high RAM requirements of its prover. Both the prover and verifier in Quisquis and Monero are CPU rather than RAM bound, and so the additional RAM did not considerably change the proving and verification time, although optimizations may be possible.

There are several approaches that do not fit into the categories below, which we discuss now. First, Mimblewimble [32,15] is a cryptocurrency design that compresses the state of the blockchain via "cut-through" transactions; it thus achieves a goal similar to ours in providing a compact UTXO set. It also achieves a notion of privacy known as *transaction indistinguishability* [15], but it does not provide anonymity in the face of network-level attackers (who can still identify the senders and recipients in individual transactions). In this sense, Mimblewim-

	Security			Efficiency				
	Anon.	Deniability	Theft prev.	UTXO growth	tx size		tx cost (ms)	
					big-$\mathcal{O}$	kB	prover	verifier
Tumblers	yes*	no	yes*	non-monotonic	low - high		slow	
Zcash	yes	no*	yes	monotonic	1	0.29	21,747	8.57
Monero	no	yes	yes	monotonic	$n + \log(v)$	2.71	982	46.3
Quisquis	yes	yes	yes	non-monotonic	$n + \log(v)$	13.4	471	71.6

Table 3: The security properties and efficiency considerations for each privacy solution. For tumblers, the stated properties are for the best solutions, but they vary significantly among solutions. No tumblers satisfy plausible deniability, and all have relatively high transaction cost due to the required latency. Numbers are given for Monero with 2 newly created TXOs and a ring size 10, and Quisquis numbers are given for one sender, 3 receivers and 12 randomly selected accounts (giving a total size of 16). n is the number of participants in the transaction, and v is the bit-length of the largest value allowed in the system.

ble achieves anonymity only against "late-comer" attackers who see the data after it is published in the blockchain, so do not see individual transactions as they move around the network. In Quisquis the attacker is assumed to be able to see all individual transactions, so we can achieve anonymity even against attackers seeing all transactions at the network level. We view this as quite realistic given the high number of full nodes in existing cryptocurrencies. Further, the techniques used in Mimblewimble are in some sense complementary to Quisquis: if individual Quisquis transactions were able to be combined using the same techniques as Mimblewimble (meaning one block would contain a single "super-transaction" combining the inputs and outputs of all the individual transactions), then against the same late-comer adversary you could argue that the anonymity set would be bigger.

Second, after posting our paper online, we were made aware of Appecoin [21], a proposal for an anonymous e-cash system. While there are some similarities in the design of this system compared to ours, including the use of shuffles and updatable public keys, the presentation of Appecoin is very informal, which in turn makes it difficult to identify the extent to which it satisfies our desired security properties.

8.1 Tumblers

Solutions for tumblers are often split into two categories: centralized [8,38,16] and decentralized [23,36,7,24,35]. In terms of the former, the one that achieves the best security is arguably TumbleBit [16], which achieves anonymity and theft prevention assuming RSA and ECDSA are secure. The most naïve centralized solutions do not even achieve theft prevention (as a centralized mix can simply steal your coins rather than permute them), and none achieve plausible deniability. The mixing process is typically quite slow, either because participants

must wait for others to join, or because multiple rounds of interaction with the tumbler are required.

In terms of decentralized solutions, the most common is Coinjoin [23], which has also given rise to the Dash cryptocurrency [1] and the coin mixing protocol ValueShuffle [35]. All of these solutions satisfy theft prevention, but none satisfy plausible deniability. The arguments for anonymity are not typically based on any cryptographic assumptions, and in some cases the protocols are not fully anonymous; e.g., ValueShuffle hides payment values but reveals which transaction outputs are unspent. One exception is Möbius [24], in which security is proven under the DDH assumption (in the random oracle model). Again, latency is often quite high due to the need to wait for others to join the mixing process, and for all participants to exchange messages.

8.2 Zcash

Zcash [6] is based on succinct zero-knowledge proofs (zk-SNARKs), which allow users to prove that a transaction is spending unspent shielded coins (i.e., coins that have already been deposited into a so-called shielded pool), without revealing which shielded coins they are. In terms of security, the anonymity set in Zcash is defined as all other coins that have been deposited into the pool. It also achieves theft prevention due to the soundness of the zero-knowledge proofs, but does not achieve plausible deniability, as all users opt in to the anonymity set by depositing their coins, and their transactions are performed independently of one another.

In terms of efficiency, since it is not known which shielded coins are being spent, no shielded coins can ever be removed from the UTXO set. The protocol mitigates this growth by storing information about shielded coins in a Merkle tree, meaning proofs grow in a logarithmic rather than a linear fashion with respect to the size of the UTXO set, but the growth of the set is still monotonic. It is relatively slow to generate Zcash transactions (`https://speed.z.cash/`), and they also require a large amount of RAM, although these numbers are expected to improve significantly in future releases [2].

Finally, in terms of cryptographic assumptions, despite recent advances [9], Zcash still requires a "trusted setup" to generate the common reference string used for the zk-SNARKs; otherwise, anyone with knowledge of its trapdoor can violate soundness and spend shielded coins that they do not rightfully own. Such structured reference strings are qualitatively different from a *common random string* (such as the one used in Quisquis), which can be generated using a random oracle, and instead require performing relatively cumbersome MPC-based "ceremonies". Additionally, all zk-SNARKs rely on strong (i.e., non-falsifiable and relatively untested) "knowledge-of-exponent"-type assumptions.

8.3 Monero

In Monero [31], senders form transactions by picking other unspent transaction outputs ("mix-ins") and forming a ring signature over them. Pairs of senders

and recipients also strengthen the anonymity of this approach by using freshly generated *stealth addresses* every time they transact, meaning every address is used to receive coins only once. In terms of security, Monero satisfies both theft prevention (due to the unforgeability of the ring signature) and plausible deniability. For anonymity, however, it is known that selecting mix-ins uniformly at random can be used to distinguish the real input from the fake ones [27]. This not only means that a more complex algorithm is needed to generate the anonymity set inside the protocol but also that it is incompatible with our definition of anonymity, in which oracle queries may result in the selection of uniformly random UTXOs. Thus, while we do not rule out the option that Monero could be proved anonymous in a different model, the same anonymity set size does provide more anonymity in Quisquis (in which all keys appear only once) than in Monero (in which keys may be used and re-used in ways that leak information).

To illustrate the main conceptual difference between Monero and Quisquis, consider the following toy example of an intersection-style attack [27,20]. Using a system in which accounts cannot be removed from the UTXO set, such as Monero, acct_1 transfers all its funds to acct_4 and uses acct_2 as its anonymity set. Then acct_2 transfers its funds to acct_5 using acct_1 as its anonymity set, and acct_3 transfers its funds to acct_6 using acct_2 as its anonymity set. As double-spending is not possible, anyone observing the blockchain can see that both acct_1 and acct_2 have already spent their contents at the time the last transaction was performed, so acct_3 must be the actual sender. Using Quisquis instead, all the accounts used as inputs would have been removed from the UTXO set and replaced by new random-looking accounts, meaning it would not be possible to use the same account twice in two different anonymity sets. Thus, such an attack could not be mounted. Furthermore, altruistic users in Quisquis could periodically send themselves money using large anonymity sets in order to "refresh" the UTXO set, in an attempt to ensure that the UTXO set has a relatively uniform distribution in terms of the age of the accounts (i.e., the time at which they were created) and thus evade attacks on Monero that are based on the differences in this distribution [27]. Again, such solutions do not work in Monero, as accounts always stay in the UTXO set.

With respect to efficiency, the UTXO set also grows monotonically, as it does in Zcash. Finally, in terms of assumptions, Monero makes the same ones as Quisquis: it requires DDH to be secure in the random oracle model.

9 Conclusions and Open Problems

In this paper we have identified and solved an open problem in anonymous cryptocurrencies; namely, that of a monotonically increasing UTXO set. We have introduced Quisquis, complete with an updatable public key system and accompanying NIZKs with low communication and computational complexity. Quisquis allows users to achieve strong notions of anonymity and theft prevention, which we have presented with accompanying reductions to the DDH and DL assumptions. As the anonymity properties are achieved by each individual

user's actions, transactions can be made anonymously without increased latency, and without strictly increasing the size of the UTXO set. While our NIZKs are already relatively efficient, we nevertheless leave as an interesting open problem the design of a special-purpose NIZK for improved communication efficiency.

Acknowledgements

Sarah Meiklejohn was supported in part by EPSRC Grant EP/N028104/1. Most of this work was done while the other three authors were working at Aarhus University and were supported by: the Concordium Blockhain Research Center, Aarhus University, Denmark; the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM); the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC); the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC); the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA).

References

1. Dash. `https://www.dash.org/`.
2. What is Jubjub? `https://z.cash/technology/jubjub.html`.
3. E. Androulaki, G. Karame, M. Roeschlin, T. Scherer, and S. Capkun. Evaluating user privacy in Bitcoin. In A.-R. Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 34–51, Okinawa, Japan, Apr. 1–5, 2013. Springer, Heidelberg, Germany.
4. M. Backes, L. Hanzlik, K. Kluczniak, and J. Schneider. Signatures with flexible public key: a unified approach to privacy-preserving signatures. IACR ePrint Archive, Report 2018/191. `https://eprint.iacr.org/2018/191.pdf`.
5. S. Bayer and J. Groth. Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 263–280, Cambridge, UK, Apr. 15–19, 2012. Springer, Heidelberg.
6. E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press.
7. G. Bissias, A. P. Ozisik, B. N. Levine, and M. Liberatore. Sybil-resistant mixing for Bitcoin. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 149–158. ACM, 2014.
8. J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten. Mixcoin: Anonymity for Bitcoin with accountable mixes. In N. Christin and R. Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 486–504, Christ Church, Barbados, Mar. 3–7, 2014. Springer, Heidelberg, Germany.
9. S. Bowe, A. Gabizon, and M. Green. A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK. In *Proceedings of the 5th Workshop on Bitcoin and Blockchain Research*, 2018.

10. B. Bünz, S. Agrawal, M. Zamani, and D. Boneh. Zether: Towards privacy in a smart contract world. `https://crypto.stanford.edu/~buenz/papers/zether.pdf`.
11. B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. *IACR Cryptology ePrint Archive*, 2017:1066, 2017.
12. S. Delgado-Segura, C. Pérez-Solà, G. Navarro-Arribas, and J. Herrera-Joancomartí. Analysis of the Bitcoin UTXO set. In *Proceedings of the 5th Workshop on Bitcoin and Blockchain Research*, 2018.
13. P. Fauzi, S. Meiklejohn, R. Mercer, and C. Orlandi. Quisquis: A New Design for Anonymous Cryptocurrencies. `https://eprint.iacr.org/2018/990`.
14. N. Fleischhacker, J. Krupp, G. Malavolta, J. Schneider, D. Schröder, and M. Simkin. Efficient unlinkable sanitizable signatures from signatures with re-randomizable keys. In C.-M. Cheng, K.-M. Chung, G. Persiano, and B.-Y. Yang, editors, *PKC 2016, Part I*, volume 9614 of *LNCS*, pages 301–330, Taipei, Taiwan, Mar. 6–9, 2016. Springer, Heidelberg, Germany.
15. G. Fuchsbauer, M. Orrù, and Y. Seurin. Aggregate cash systems: A cryptographic investigation of Mimblewimble. In *Proceedings of Eurocrypt 2019*, 2019.
16. E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg. TumbleBit: an untrusted Bitcoin-compatible anonymous payment hub. In *Proceedings of NDSS 2017*, 2017.
17. E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse attacks on Bitcoin's peer-to-peer network. In *Proceedings of the USENIX Security Symposium*, 2017.
18. A. Hinteregger and B. Haslhofer. An Empirical Analysis of Monero Cross-Chain Traceability. *CoRR*, abs/1812.02808, 2018.
19. G. Kappos, H. Yousaf, M. Maller, and S. Meiklejohn. An empirical analysis of anonymity in Zcash. In W. Enck and A. P. Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 463–477. USENIX Association, 2018.
20. A. Kumar, C. Fischer, S. Tople, and P. Saxena. A traceability analysis of Monero's blockchain. In S. N. Foley, D. Gollmann, and E. Snekkenes, editors, *ESORICS 2017, Part II*, volume 10493 of *LNCS*, pages 153–173, Oslo, Norway, Sept. 11–15, 2017. Springer, Heidelberg, Germany.
21. S. D. Lerner. Appecoin: Practical anonymous peer-to-peer e-cash system. `https://bitslog.files.wordpress.com/2014/04/appecoin28.pdf`.
22. G. Malavolta and D. Schröder. Efficient ring signatures in the standard model. In T. Takagi and T. Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 128–157, Hong Kong, China, Dec. 3–7, 2017. Springer, Heidelberg, Germany.
23. G. Maxwell. Coinjoin: Bitcoin privacy for the real world. In *Post on Bitcoin forum*, 2013.
24. S. Meiklejohn and R. Mercer. Möbius: Trustless tumbling for transaction privacy. *Proceedings on Privacy Enhancing Technologies*, 2018.
25. S. Meiklejohn and C. Orlandi. Privacy-Enhancing Overlays in Bitcoin. In M. Brenner, N. Christin, B. Johnson, and K. Rohloff, editors, *FC 2015 Workshops*, volume 8976 of *LNCS*, pages 127–141, San Juan, Puerto Rico, Jan. 30, 2015. Springer, Heidelberg, Germany.
26. S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A Fistful of Bitcoins: Characterizing Payments Among. Men with No Names. In *Proceedings of the 2013 Internet Measurement Conference*, pages 127–140. ACM, 2013.

27. A. Miller, M. Möser, K. Lee, and A. Narayanan. An empirical analysis of linkability in the Monero blockchain. *Proceedings on Privacy Enhancing Technologies*, 2018.
28. P. Moreno-Sanchez, M. B. Zafar, and A. Kate. Listening to whispers of Ripple: Linking wallets and deanonymizing transactions in the Ripple network. *Proceedings on Privacy Enhancing Technologies*, 2016(4):436–453, 2016.
29. M. Möser, R. Böhme, and D. Breuker. An inquiry into money laundering tools in the Bitcoin ecosystem. In *Proceedings of the APWG E-Crime Researchers Summit*, 2013.
30. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. `bitcoin.org/bitcoin.pdf`.
31. S. Noether, A. Mackenzie, et al. Ring confidential transactions. *Ledger*, 1:1–18, 2016.
32. A. Poelstra. Mimblewimble, 2016. `https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf`.
33. F. Reid and M. Harrigan. An analysis of anonymity in the Bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.
34. D. Ron and A. Shamir. Quantitative analysis of the full Bitcoin transaction graph. In A.-R. Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 6–24, Okinawa, Japan, Apr. 1–5, 2013. Springer, Heidelberg, Germany.
35. T. Ruffing and P. Moreno-Sanchez. ValueShuffle: Mixing confidential transactions for comprehensive transaction privacy in Bitcoin. In M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. A. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, editors, *FC 2017 Workshops*, volume 10323 of *LNCS*, pages 133–154, Sliema, Malta, Apr. 7, 2017. Springer, Heidelberg, Germany.
36. T. Ruffing, P. Moreno-Sanchez, and A. Kate. CoinShuffle: Practical decentralized coin mixing for Bitcoin. In M. Kutylowski and J. Vaidya, editors, *ESORICS 2014, Part II*, volume 8713 of *LNCS*, pages 345–364, Wroclaw, Poland, Sept. 7–11, 2014. Springer, Heidelberg, Germany.
37. M. Spagnuolo, F. Maggi, and S. Zanero. BitIodine: Extracting intelligence from the Bitcoin network. In N. Christin and R. Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 457–468, Christ Church, Barbados, Mar. 3–7, 2014. Springer, Heidelberg, Germany.
38. L. Valenta and B. Rowan. Blindcoin: Blinded, accountable mixes for Bitcoin. In M. Brenner, N. Christin, B. Johnson, and K. Rohloff, editors, *FC 2015 Workshops*, volume 8976 of *LNCS*, pages 112–126, San Juan, Puerto Rico, Jan. 30, 2015. Springer, Heidelberg, Germany.
39. B. R. Waters, E. W. Felten, and A. Sahai. Receiver anonymity via incomparable public keys. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM CCS 03*, pages 112–121, Washington D.C., USA, Oct. 27–30, 2003. ACM Press.
40. Z. Yu, M. H. Au, J. Yu, R. Yang, Q. Xu, and W. F. Lau. New Empirical Traceability Analysis of CryptoNote-Style Blockchains.