Zerocash: Decentralized Anonymous Payments from Bitcoin

Eli Ben-Sasson*, Alessandro Chiesa†, Christina Garman‡, Matthew Green‡, Ian Miers‡, Eran Tromer§, Madars Virza†
*Technion, `eli@cs.technion.ac.il`
†MIT, {`alexch, madars`}`@mit.edu`
‡Johns Hopkins University, {`cgarman, imiers, mgreen`}`@cs.jhu.edu`
§Tel Aviv University, `tromer@cs.tau.ac.il`

***Abstract*—Bitcoin is the first digital currency to see widespread adoption. While payments are conducted between pseudonyms, Bitcoin cannot offer strong privacy guarantees: payment transactions are recorded in a public decentralized ledger, from which much information can be deduced. Zerocoin (Miers et al., IEEE S&P 2013) tackles some of these privacy issues by unlinking transactions from the payment's origin. Yet, it still reveals payments' destinations and amounts, and is limited in functionality.**

In this paper, we construct a full-fledged ledger-based digital currency with strong privacy guarantees. Our results leverage recent advances in *zero-knowledge Succinct Non-interactive ARguments of Knowledge* (zk-SNARKs).

First, we formulate and construct *decentralized anonymous payment schemes* (DAP schemes). A DAP scheme enables users to directly pay each other privately: the corresponding transaction hides the payment's origin, destination, and transferred amount. We provide formal definitions and proofs of the construction's security.

Second, we build Zerocash, a practical instantiation of our DAP scheme construction. In Zerocash, transactions are less than 1 kB and take under 6 ms to verify — orders of magnitude more efficient than the less-anonymous Zerocoin and competitive with plain Bitcoin.

Keywords: Bitcoin, decentralized electronic cash, zero knowledge

I. Introduction

Bitcoin is the first digital currency to achieve widespread adoption. The currency owes its rise in part to the fact that, unlike traditional e-cash schemes [1, 2, 3], it requires no trusted parties. Instead of appointing a central bank, Bitcoin leverages a distributed ledger known as the *block chain* to store transactions made between users. Because the block chain is massively replicated by mutually-distrustful peers, the information it contains is public.

While users may employ many identities (or *pseudonyms*) to enhance their privacy, an increasing body of research shows that anyone can *de-anonymize* Bitcoin by using information in the block chain [4, 5, 6], such as the structure of the transaction graph as well as the value and dates of transactions. As a result, Bitcoin fails to offer even a modicum of the privacy provided by traditional payment systems, let alone the robust privacy of anonymous e-cash schemes.

While Bitcoin is not anonymous itself, those with sufficient motivation can obfuscate their transaction history with the help of *mixes* (also known as *laundries* or *tumblers*). A mix allows users to entrust a set of coins to a pool operated by a central party and then, after some interval, retrieve different coins (with the same total value) from the pool. Yet, mixes suffer from three limitations: (i) the delay to reclaim coins must be large to allow enough coins to be mixed in; (ii) the mix can trace coins; and (iii) the mix may steal coins.[1] For users with "something to hide," these risks may be acceptable. But typical legitimate users (1) wish to keep their spending habits private from their peers, (2) are risk-averse and do not wish to expend continual effort in protecting their privacy, and (3) are often not sufficiently aware of their compromised privacy.

To protect their *privacy*, users thus need an instant, risk-free, and, most importantly, automatic guarantee that data revealing their spending habits and account balances is not publicly accessible by their neighbors, co-workers, and merchants. Anonymous transactions also guarantee that the market value of a coin is independent of its history, thus ensuring legitimate users' coins remain *fungible*.[2]

Zerocoin: a decentralized mix. Miers et al. [8] proposed Zerocoin, which extends Bitcoin to provide strong anonymity guarantees. Like many e-cash protocols (e.g., [2]), Zerocoin employs zero-knowledge proofs to prevent transaction graph analyses. Unlike earlier practical e-cash protocols, however, Zerocoin does not rely on digital signatures to validate coins, nor does it require a central bank to prevent double spending. Instead, Zerocoin authenticates coins by proving, in zero-knowledge, that they belong to a public list of valid coins (which can be maintained on the block chain). Yet, rather than a full-fledged anonymous currency, Zerocoin is a *decentralized mix*, where users may periodically "wash" their bitcoins via the Zerocoin protocol. Routine day-to-day transactions must be conducted via Bitcoin, due to reasons that we now review.

The first reason is performance. Redeeming zerocoins requires double-discrete-logarithm proofs of knowledge, which have size that exceeds 45 kB and require 450 ms to verify (at the 128-bit security level).[3] These proofs must be broadcast

[1]CoinJoin [7], an alternative proposal, replaces the central party of a mix with multi-signature transactions that involve many collaborating Bitcoin users. CoinJoin can thus only mix small volumes of coins amongst users who are currently online, is prone to denial-of-service attacks by third parties, and requires effort to find mixing partners.

[2]While the methods we detail in this paper accomplish this, the same techniques open the door for privacy preserving accountability and oversight (see Section X).

[3]These published numbers [8] actually use a mix of parameters at both 128-bit and 80-bit security for different components of the construction. The cost is higher if all parameters are instantiated at the 128-bit security level.

through the network, verified by every node, and permanently stored in the ledger. The entailed costs are higher, by orders of magnitude, than those in Bitcoin and can seriously tax a Bitcoin network operating at normal scale.

The second reason is functionality. While Zerocoin constitutes a basic e-cash scheme, it lacks critical features required of full-fledged anonymous payments. First, Zerocoin uses coins of fixed denomination: it does not support payments of exact values, nor does it provide a means to make change following a transaction (i.e., divide coins). Second, Zerocoin has no mechanism for one user to pay another one directly in "zerocoins." And third, while Zerocoin provides anonymity by unlinking a payment transaction from its origin address, it does not hide the amount or other metadata about transactions occurring on the network.

Our contribution. In this work we address the aforementioned issues via two main contributions.

(1) We introduce the notion of a *decentralized anonymous payment scheme*, which formally captures the functionality and security guarantees of a full-fledged decentralized electronic currency with strong anonymity guarantees. We provide a construction of this primitive and prove its security under specific cryptographic assumptions. The construction leverages recent advances in the area of zero-knowledge proofs. Specifically, it uses *zero-knowledge Succinct Non-interactive ARguments of Knowledge* (zk-SNARKs) [9, 10, 11, 12, 13, 14, 15, 16].

(2) We achieve an implementation of the above primitive, via a system that we call **Zerocash**. Compared to Zerocoin, our system (at 128 bits of security):

- Reduces the size of transactions spending a coin by 97.7%.
- Reduces the spend-transaction verification time by 98.6%.
- Allows for anonymous transactions of variable amounts.
- Hides transaction amounts and the values of coins held by users.
- Allows for payments to be made directly to a user's fixed address (without user interaction).

To validate our system, we measured its performance and established feasibility by conducting experiments in a test network of 1000 nodes (approximately $\frac{1}{16}$ of the unique IPs in the Bitcoin network and $\frac{1}{3}$ of the nodes reachable at any given time [17]). This inspires confidence that Zerocash can be deployed as a fork of Bitcoin and operate at the same scale. Thus, due to its significantly improved functionality and performance, Zerocash makes it possible to entirely replace traditional Bitcoin payments with anonymous alternatives.

Concurrent work. The idea of using zk-SNARKs in the setting of Bitcoin was first presented by one of the authors at Bitcoin 2013 [18]. In concurrent work, Danezis et al. [19] suggest using zk-SNARKs to reduce proof size and verification time in Zerocoin; see Section IX for a comparison.

A. zk-SNARKs

We now sketch in more technical terms the definition of a zk-SNARK; see Section II for more details. A zk-SNARK is a non-interactive zero-knowledge proof of knowledge that is *succinct*, i.e., for which proofs are very short and easy to verify. More precisely, let $\mathcal{L}$ be an NP language, and let C be a nondeterministic decision circuit for $\mathcal{L}$ on a given instance size n. A zk-SNARK can be used to prove and verify membership in $\mathcal{L}$, for instances of size n, as follows. After taking C as input, a trusted party conducts a one-time setup phase that results in two public keys: a proving key pk and a verification key vk. The proving key pk enables any (untrusted) prover to produce a proof π attesting to the fact that $x \in \mathcal{L}$, for an instance x (of size n) of his choice. The non-interactive proof π is *zero knowledge* and a *proof of knowledge*. Anyone can use the verification key vk to verify the proof π; in particular zk-SNARK proofs are publicly verifiable: anyone can verify π, without ever having to interact with the prover that generated π. Succinctness requires that (for a given security level) π has *constant size* and can be verified in time that is linear in $|x|$ (rather than linear in $|C|$).

B. Decentralized anonymous payment schemes

We construct a *decentralized anonymous payment (DAP) scheme*, which is a decentralized e-cash scheme that allows direct anonymous payments of any amount. See Section III for a formal definition. Here, we outline our construction in six incremental steps; the construction details are in Section IV.

Our construction functions on top of any ledger-based base currency, such as Bitcoin. At any given time, a unique valid snapshot of the currency's *ledger* is available to all users. The ledger is a sequence of *transactions* and is append-only. Transactions include both the underlying currency's transactions, as well as new transactions introduced by our construction. For concreteness, we focus the discussion below on Bitcoin (though later definitions and constructions are stated abstractly). We assume familiarity with Bitcoin [20] and Zerocoin [8].

Step 1: user anonymity with fixed-value coins. We first describe a simplified construction, in which all coins have the same value of, e.g., 1 BTC. This construction, similar to the Zerocoin protocol, shows how to hide a payment's origin. In terms of tools, we make use of zk-SNARKs (recalled above) and a commitment scheme. Let COMM denote a statistically-hiding non-interactive commitment scheme (i.e., given randomness r and message m, the commitment is $c := \mathsf{COMM}_r(m)$; subsequently, c is opened by revealing r and m, and one can verify that $\mathsf{COMM}_r(m)$ equals c).

In the simplified construction, a new coin $\mathbf{c}$ is minted as follows: a user u samples a random *serial number* sn and a *trapdoor* r, computes a *coin commitment* $\mathsf{cm} := \mathsf{COMM}_r(\mathsf{sn})$, and sets $\mathbf{c} := (r, \mathsf{sn}, \mathsf{cm})$. A corresponding mint transaction $\mathsf{tx}_{\mathsf{Mint}}$, containing cm (but not sn or r), is sent to the ledger; $\mathsf{tx}_{\mathsf{Mint}}$ is appended to the ledger only if u has paid 1 BTC to a backing escrow pool (e.g., the 1 BTC may be paid via plaintext information encoded in $\mathsf{tx}_{\mathsf{Mint}}$). Mint transactions are thus certificates of deposit, deriving their value from the backing pool.

Subsequently, letting CMList denote the list of all coin commitments on the ledger, u may spend $\mathbf{c}$ by posting a spend transaction $\mathsf{tx}_{\mathsf{Spend}}$ that contains (i) the coin's serial number sn; and (ii) a zk-SNARK proof π of the NP statement *"I know r such that $\mathsf{COMM}_r(\mathsf{sn})$ appears in the list CMList of coin commitments"*. Assuming that sn does not already appear on the ledger (as part of a past spend transaction), u can redeem the deposited amount of 1 BTC, which u can either keep for himself, transfer to someone else, or immediately deposit into a new coin. (If sn does already appear on the ledger, this is considered double spending, and the transaction is discarded.)

User anonymity is achieved because the proof π is zero-knowledge: while sn is revealed, no information about r is, and finding which of the numerous commitments in CMList corresponds to a particular spend transaction $\mathsf{tx}_{\mathsf{Spend}}$ is equivalent to inverting $f(x) := \mathsf{COMM}_x(\mathsf{sn})$, which is assumed to be infeasible. Thus, the origin of the payment is anonymous.

Step 2: compressing the list of coin commitments. In the above NP statement, CMList is specified explicitly as a list of coin commitments. This naive representation severely limits scalability because the time and space complexity of most protocol algorithms (e.g., the proof verification algorithm) grows linearly with CMList. Moreover, coin commitments corresponding to already spent coins cannot be dropped from CMList to reduce costs, since they cannot be identified (due to the same zero-knowledge property that provides anonymity).

As in [3], we rely on a collision-resistant hash function CRH to avoid an explicit representation of CMList. We maintain an efficiently updatable append-only CRH-based Merkle tree $\mathsf{Tree}(\mathsf{CMList})$ over the (growing) list CMList. Letting rt denote the root of $\mathsf{Tree}(\mathsf{CMList})$, it is well-known that updating rt to account for insertion of new leaves can be done with time and space proportional to the tree depth. Hence, the time and space complexity is reduced from linear in the size of CMList to logarithmic. With this in mind, we modify the NP statement to the following one: *"I know r such that $\mathsf{COMM}_r(\mathsf{sn})$ appears as a leaf in a CRH-based Merkle tree whose root is rt"*. Compared with the naive data structure for CMList, this modification increases exponentially the size of CMList which a given zk-SNARK implementation can support (concretely, using trees of depth 64, Zerocash supports 2^{64} coins).

Step 3: extending coins for direct anonymous payments. So far, the coin commitment cm of a coin $\mathbf{c}$ is a commitment to the coin's serial number sn. However, this creates a problem when transferring $\mathbf{c}$ to another user. Indeed, suppose that a user u_A created $\mathbf{c}$, and u_A sends $\mathbf{c}$ to another user u_B. First, since u_A knows sn, the spending of $\mathbf{c}$ by u_B is both not anonymous (since u_A sees when $\mathbf{c}$ is spent, by recognizing sn) and risky (since u_A could still spend $\mathbf{c}$ first). Thus, u_B must immediately spend $\mathbf{c}$ and mint a new coin $\mathbf{c}'$ to protect himself. Second, if u_A in fact wants to transfer to u_B, e.g., 100 BTC, then doing so is both unwieldy (since it requires 100 transfers) and not anonymous (since the amount of the transfer is leaked). And third, transfers in amounts that are not multiples of 1 BTC (the fixed value of a coin) are not supported. Thus, the simplified construction described is inadequate as a payment scheme.

We address this by modifying the derivation of a coin commitment, and using pseudorandom functions to target payments and to derive serial numbers, as follows. We use three pseudorandom functions (derived from a single one). For a seed x these are denoted $\mathsf{PRF}^{\mathrm{addr}}_x(\cdot)$, $\mathsf{PRF}^{\mathrm{sn}}_x(\cdot)$, and $\mathsf{PRF}^{\mathrm{pk}}_x(\cdot)$. We assume that $\mathsf{PRF}^{\mathrm{sn}}$ is moreover collision-resistant.

To provide targets for payments, we use *addresses*: each user u generates an address key pair $(a_{\mathsf{pk}}, a_{\mathsf{sk}})$. The coins of u contain the value a_{pk} and can be spent only with knowledge of a_{sk}. A key pair $(a_{\mathsf{pk}}, a_{\mathsf{sk}})$ is sampled by selecting a random seed a_{sk} and setting $a_{\mathsf{pk}} := \mathsf{PRF}^{\mathrm{addr}}_{a_{\mathsf{sk}}}(0)$. A user can generate and use any number of address key pairs.

Next, we re-design minting to allow for greater functionality. To mint a coin $\mathbf{c}$ of a desired value v, the user u first samples ρ, which is a secret value that determines the coin's serial number as $\mathsf{sn} := \mathsf{PRF}^{\mathrm{sn}}_{a_{\mathsf{sk}}}(\rho)$. Then, u commits to the tuple $(a_{\mathsf{pk}}, v, \rho)$ in two phases: (a) u computes $k := \mathsf{COMM}_r(a_{\mathsf{pk}} \| \rho)$ for a random r; and then (b) u computes $\mathsf{cm} := \mathsf{COMM}_s(v \| k)$ for a random s. The minting results in a coin $\mathbf{c} := (a_{\mathsf{pk}}, v, \rho, r, s, \mathsf{cm})$ and a mint transaction $\mathsf{tx}_{\mathsf{Mint}} := (v, k, s, \mathsf{cm})$. Crucially, due to the nested commitment, anyone can verify that cm in $\mathsf{tx}_{\mathsf{Mint}}$ is a coin commitment of a coin of value v (by checking that $\mathsf{COMM}_s(v \| k)$ equals cm) but cannot discern the owner (by learning the address key a_{pk}) or serial number (derived from ρ) because these are hidden in k. As before, $\mathsf{tx}_{\mathsf{Mint}}$ is accepted by the ledger only if u deposits the correct amount, in this case v BTC.

Coins are spent using the *pour* operation, which takes a set of input coins, to be consumed, and "pours" their value into a set of fresh output coins — such that the total value of output coins equals the total value of the input coins. Suppose that u, with address key pair $(a^{\mathsf{old}}_{\mathsf{pk}}, a^{\mathsf{old}}_{\mathsf{sk}})$, wishes to consume his coin $\mathbf{c}^{\mathsf{old}} = (a^{\mathsf{old}}_{\mathsf{pk}}, v^{\mathsf{old}}, \rho^{\mathsf{old}}, r^{\mathsf{old}}, s^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}})$ and produce two new coins $\mathbf{c}^{\mathsf{new}}_1$ and $\mathbf{c}^{\mathsf{new}}_2$, with total value $v^{\mathsf{new}}_1 + v^{\mathsf{new}}_2 = v^{\mathsf{old}}$, respectively targeted at address public keys $a^{\mathsf{new}}_{\mathsf{pk},1}$ and $a^{\mathsf{new}}_{\mathsf{pk},2}$. (The addresses $a^{\mathsf{new}}_{\mathsf{pk},1}$ and $a^{\mathsf{new}}_{\mathsf{pk},2}$ may belong to u or to some other user.) The user u, for each $i \in \{1, 2\}$, proceeds as follows: (i) u samples serial number randomness ρ^{new}_i; (ii) u computes $k^{\mathsf{new}}_i := \mathsf{COMM}_{r^{\mathsf{new}}_i}(a^{\mathsf{new}}_{\mathsf{pk},i} \| \rho^{\mathsf{new}}_i)$ for a random r^{new}_i; and (iii) u computes $\mathsf{cm}^{\mathsf{new}}_i := \mathsf{COMM}_{s^{\mathsf{new}}_i}(v^{\mathsf{new}}_i \| k^{\mathsf{new}}_i)$ for a random s^{new}_i.

This yields the coins $\mathbf{c}^{\mathsf{new}}_1 := (a^{\mathsf{new}}_{\mathsf{pk},1}, v^{\mathsf{new}}_1, \rho^{\mathsf{new}}_1, r^{\mathsf{new}}_1, s^{\mathsf{new}}_1, \mathsf{cm}^{\mathsf{new}}_1)$ and $\mathbf{c}^{\mathsf{new}}_2 := (a^{\mathsf{new}}_{\mathsf{pk},2}, v^{\mathsf{new}}_2, \rho^{\mathsf{new}}_2, r^{\mathsf{new}}_2, s^{\mathsf{new}}_2, \mathsf{cm}^{\mathsf{new}}_2)$. Next, u produces a zk-SNARK proof $\pi_{\texttt{POUR}}$ for the following NP statement, which we call $\texttt{POUR}$:

"Given the Merkle-tree root rt, serial number $\mathsf{sn}^{\mathsf{old}}$, and coin commitments $\mathsf{cm}^{\mathsf{new}}_1, \mathsf{cm}^{\mathsf{new}}_2$, I know coins $\mathbf{c}^{\mathsf{old}}, \mathbf{c}^{\mathsf{new}}_1, \mathbf{c}^{\mathsf{new}}_2$, and address secret key $a^{\mathsf{old}}_{\mathsf{sk}}$ such that:

- *The coins are well-formed: for $\mathbf{c}^{\mathsf{old}}$ it holds that $k^{\mathsf{old}} = \mathsf{COMM}_{r^{\mathsf{old}}}(a^{\mathsf{old}}_{\mathsf{pk}} \| \rho^{\mathsf{old}})$ and $\mathsf{cm}^{\mathsf{old}} = \mathsf{COMM}_{s^{\mathsf{old}}}(v^{\mathsf{old}} \| k^{\mathsf{old}})$; and similarly for $\mathbf{c}^{\mathsf{new}}_1$ and $\mathbf{c}^{\mathsf{new}}_2$.*
- *The address secret key matches the public key: $a^{\mathsf{old}}_{\mathsf{pk}} = \mathsf{PRF}^{\mathrm{addr}}_{a^{\mathsf{old}}_{\mathsf{sk}}}(0)$.*
- *The serial number is computed correctly: $\mathsf{sn}^{\mathsf{old}} := \mathsf{PRF}^{\mathrm{sn}}_{a^{\mathsf{old}}_{\mathsf{sk}}}(\rho^{\mathsf{old}})$.*
- *The coin commitment $\mathsf{cm}^{\mathsf{old}}$ appears as a leaf of a Merkle-*

tree with root rt.

- *The values add up:* $v_1^{\mathsf{new}} + v_2^{\mathsf{new}} = v^{\mathsf{old}}$.*"*

A resulting pour transaction $\mathsf{tx}_{\mathsf{Pour}} := (\mathsf{rt}, \mathsf{sn}^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, \pi_{\mathtt{POUR}})$ is appended to the ledger. (As before, the transaction is rejected if the serial number sn appears in a previous transaction.)

Now suppose that u does not know, say, the address secret key $a_{\mathsf{sk},1}^{\mathsf{new}}$ that is associated with the public key $a_{\mathsf{pk},1}^{\mathsf{new}}$. Then, u cannot spend $\mathbf{c}_1^{\mathsf{new}}$ because he cannot provide $a_{\mathsf{sk},1}^{\mathsf{new}}$ as part of the witness of a subsequent pour operation. Furthermore, when a user that knows $a_{\mathsf{sk},1}^{\mathsf{new}}$ does spend $\mathbf{c}_1^{\mathsf{new}}$, the user u cannot track it, because he knows no information about its revealed serial number, which is $\mathsf{sn}_1^{\mathsf{new}} := \mathsf{PRF}^{\mathrm{sn}}_{a_{\mathsf{sk},1}^{\mathsf{new}}}(\rho_1^{\mathsf{new}})$.

Also observe that $\mathsf{tx}_{\mathsf{Pour}}$ reveals no information about how the value of the consumed coin was divided among the two new fresh coins, nor which coin commitment corresponds to the consumed coin, nor the address public keys to which the two new fresh coins are targeted. The payment was conducted in full anonymity.

More generally, a user may pour $N^{\mathsf{old}} \geq 0$ coins into $N^{\mathsf{new}} \geq 0$ coins. For simplicity we consider the case $N^{\mathsf{old}} = N^{\mathsf{new}} = 2$, without loss of generality. Indeed, for $N^{\mathsf{old}} < 2$, the user can mint a coin with value 0 and then provide it as a "null" input, and for $N^{\mathsf{new}} < 2$, the user can create (and discard) a new coin with value 0. For $N^{\mathsf{old}} > 2$ or $N^{\mathsf{new}} > 2$, the user can compose $\log N^{\mathsf{old}} + \log N^{\mathsf{new}}$ of the 2-input/2-output pours.

Step 4: sending coins. Suppose that $a_{\mathsf{pk},1}^{\mathsf{new}}$ is the address public key of u_1. In order to allow u_1 to actually spend the new coin $\mathbf{c}_1^{\mathsf{new}}$ produced above, u must somehow send the secret values in $\mathbf{c}_1^{\mathsf{new}}$ to u_1. One way is for u to send u_1 a private message, but the requisite private communication channel necessitates additional infrastructure or assumptions. We avoid this "out-of-band" channel and instead build this capability directly into our construction by leveraging the ledger as follows.

We modify the structure of an address key pair. Each user now has a key pair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$, where $\mathsf{addr}_{\mathsf{pk}} = (a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}})$ and $\mathsf{addr}_{\mathsf{sk}} = (a_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}})$. The values $(a_{\mathsf{pk}}, a_{\mathsf{sk}})$ are generated as before. In addition, $(\mathsf{pk}_{\mathsf{enc}}, \mathsf{sk}_{\mathsf{enc}})$ is a key pair for a *key-private encryption scheme* [21].

Then, u computes the ciphertext $\mathbf{C}_1$ that is the encryption of the plaintext $(v_1^{\mathsf{new}}, \rho_1^{\mathsf{new}}, r_1^{\mathsf{new}}, s_1^{\mathsf{new}})$, under $\mathsf{pk}_{\mathsf{enc},1}^{\mathsf{new}}$ (which is part of u_1's address public key $\mathsf{addr}_{\mathsf{sk},1}^{\mathsf{new}}$), and includes $\mathbf{C}_1$ in the pour transaction $\mathsf{tx}_{\mathsf{Pour}}$. The user u_1 can then find and decrypt this message (using his $\mathsf{sk}_{\mathsf{enc},1}^{\mathsf{new}}$) by scanning the pour transactions on the public ledger. Again, note that adding $\mathbf{C}_1$ to $\mathsf{tx}_{\mathsf{Pour}}$ leaks neither paid amounts, nor target addresses due to the key-private property of the encryption scheme. (The user u does the same with $\mathbf{c}_2^{\mathsf{new}}$ and includes a corresponding ciphertext $\mathbf{C}_2$ in $\mathsf{tx}_{\mathsf{Pour}}$.)

Step 5: public outputs. The construction so far allows users to mint, merge, and split coins. But how can a user redeem one of his coins, i.e., convert it back to the base currency (Bitcoin)? For this, we modify the pour operation to include a *public output*. When spending a coin, the user u also specifies a nonnegative v_{pub} and an arbitrary string info. The balance equation in the NP statement $\mathtt{POUR}$ is changed accordingly: "$v_1^{\mathsf{new}} + v_2^{\mathsf{new}} + v_{\mathsf{pub}} = v^{\mathsf{old}}$". Thus, of the input value v^{old}, a part v_{pub} is publicly declared, and its target is specified, somehow, by the string info. The string info can be used to specify the destination of these redeemed funds (e.g., a Bitcoin wallet public key).[4] Both v_{pub} and info are now included in the resulting pour transaction $\mathsf{tx}_{\mathsf{Pour}}$. (The public output is optional, as the user u can set $v_{\mathsf{pub}} = 0$.)

Step 6: non-malleability. To prevent malleability attacks on a pour transaction $\mathsf{tx}_{\mathsf{Pour}}$ (e.g., embezzlement by re-targeting the public output of the pour by modifying info), we further modify the NP statement $\mathtt{POUR}$ and use digital signatures. Specifically, during the pour operation, the user u (i) samples a key pair $(\mathsf{pk}_{\mathsf{sig}}, \mathsf{sk}_{\mathsf{sig}})$ for a one-time signature scheme; (ii) computes $h_{\mathsf{Sig}} := \mathsf{CRH}(\mathsf{pk}_{\mathsf{sig}})$; (iii) computes the two values $h_1 := \mathsf{PRF}^{\mathrm{pk}}_{a_{\mathsf{sk},1}^{\mathsf{old}}}(h_{\mathsf{Sig}})$ and $h_2 := \mathsf{PRF}^{\mathrm{pk}}_{a_{\mathsf{sk},2}^{\mathsf{old}}}(h_{\mathsf{Sig}})$, which act as MACs to "tie" h_{Sig} to both address secret keys; (iv) modifies $\mathtt{POUR}$ to include the three values $h_{\mathsf{Sig}}, h_1, h_2$ and prove that the latter two are computed correctly; and (v) uses $\mathsf{sk}_{\mathsf{sig}}$ to sign every value associated with the pour operation, thus obtaining a signature σ, which is included, along with $\mathsf{pk}_{\mathsf{sig}}$, in $\mathsf{tx}_{\mathsf{Pour}}$. Since the $a_{\mathsf{sk},i}^{\mathsf{old}}$ are secret, and with high probability h_{Sig} changes for each pour transaction, the values h_1, h_2 are unpredictable. Moreover, the signature on the NP statement (and other values) binds all of these together.

This ends the outline of the construction, which is summarized in part in Figure 1. We conclude by noting that, due to the zk-SNARK, our construction requires a one-time trusted setup of public parameters. The trust affects soundness of the proofs, though anonymity continues to hold even if the setup is corrupted by a malicious party.

C. Zerocash

We outline Zerocash, a concrete implementation, at 128 bits of security, of our DAP scheme construction; see Section V for details. Zerocash entails carefully instantiating the cryptographic ingredients of the construction to ensure that the zk-SNARK, the "heaviest" component, is efficient enough in practice. In the construction, the zk-SNARK is used to prove/verify a specific NP statement: $\mathtt{POUR}$. While zk-SNARKs are asymptotically efficient, their concrete efficiency depends on the arithmetic circuit C that is used to decide the NP statement. Thus, we seek instantiations for which we can design a relatively-small arithmetic circuit $C_{\mathtt{POUR}}$ for verifying the NP statement $\mathtt{POUR}$.

Our approach is to instantiate all of the necessary cryptographic ingredients (commitment schemes, pseudorandom functions, and collision-resistant hashing) based on SHA256. We first design a hand-optimized circuit for verifying SHA256 computations (or, more precisely, its compression function,

[4]These public outputs can be considered as an "input" to a Bitcoin-style transaction, where the info string contains the Bitcoin output scripts. This mechanism also allows us to support Bitcoin's public transaction fees.

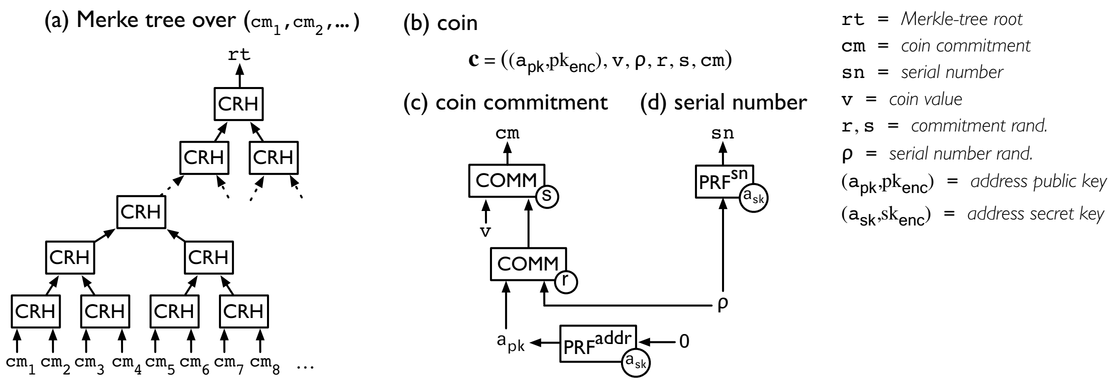


Fig. 1: **(a)** Illustration of the CRH-based Merkle tree over the list CMList of coin commitments. **(b)** A coin **c**. **(c)** Illustration of the structure of a coin commitment cm. **(d)** Illustration of the structure of a coin serial number sn.

which suffices for our purposes).[5] Then, we use this circuit in constructing $C_{\texttt{POUR}}$, which verifies all the necessary checks for satisfying the NP statement $C_{\texttt{POUR}}$.

This, along with judicious parameter choices, and a state-of-the-art implementation of a zk-SNARK for arithmetic circuits [16] (see Section II-C), results in a zk-SNARK prover running time of few minutes and zk-SNARK verifier running time of few milliseconds. This allows the DAP scheme implementation to be practical for deployment, as our experiments show.

Zerocash can be integrated into Bitcoin or forks of it (commonly referred to as "altcoins"); we later describe how this is done.

D. Paper organization

The remainder of this paper is organized as follows. Section II provides background on zk-SNARKs. We define DAP schemes in Section III, and our construction thereof in Section IV. Section V discusses the concrete instantiation in Zerocash. Section VI describes the integration of Zerocash into existing ledger-based currencies. Section VII provides microbenchmarks for our prototype implementation, as well as results based on full-network simulations. Section VIII describes optimizations. We discuss concurrent work in Section IX and summarize our contributions and future directions in Section X.

II. Background on zk-SNARKs

The main cryptographic primitive used in this paper is a special kind of *Succinct Non-interactive ARgument of Knowledge* (SNARK). Concretely, we use a *publicly-verifiable preprocessing zero-knowledge* SNARK, or zk-SNARK for short. In this section we provide basic background on zk-SNARKs, provide an informal definition, and recall known constructions and implementations.

[5]Alternatively, we could have opted to rely on the circuit generators [13, 14, 16], which support various classes of C programs, by writing C code expressing the POUR checks. However, as discussed later, these generic approaches are more expensive than our hand-optimized construction.

A. Informal definition

We informally define zk-SNARKs for arithmetic circuit satisfiability. We refer the reader to, e.g., [11] for a formal definition.

For a field $\mathbb{F}$, an $\mathbb{F}$-*arithmetic circuit* takes inputs that are elements in $\mathbb{F}$, and its gates output elements in $\mathbb{F}$. We naturally associate a circuit with the function it computes. To model nondeterminism we consider circuits that have an *input* $x \in \mathbb{F}^n$ and an auxiliary input $a \in \mathbb{F}^h$, called a *witness*. The circuits we consider only have *bilinear gates*.[6] Arithmetic circuit satisfiability is defined analogously to the boolean case, as follows.

Definition II.1. The *arithmetic circuit satisfiability problem* of an $\mathbb{F}$-arithmetic circuit $C\colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$ is captured by the relation $\mathcal{R}_C = \{(x,a) \in \mathbb{F}^n \times \mathbb{F}^h \colon C(x,a) = 0^l\}$; its language is $\mathcal{L}_C = \{x \in \mathbb{F}^n : \exists\, a \in \mathbb{F}^h \text{ s.t. } C(x,a) = 0^l\}$.

Given a field $\mathbb{F}$, a (publicly-verifiable preprocessing) **zk-SNARK** for $\mathbb{F}$-arithmetic circuit satisfiability is a triple of polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$:

- $\mathsf{KeyGen}(1^\lambda, C) \to (\mathsf{pk}, \mathsf{vk})$. On input a security parameter λ (presented in unary) and an $\mathbb{F}$-arithmetic circuit C, the *key generator* KeyGen probabilistically samples a *proving key* pk and a *verification key* vk. Both keys are published as public parameters and can be used, any number of times, to prove/verify membership in $\mathcal{L}_C$.
- $\mathsf{Prove}(\mathsf{pk}, x, a) \to \pi$. On input a proving key pk and any $(x,a) \in \mathcal{R}_C$, the *prover* Prove outputs a non-interactive proof π for the statement $x \in \mathcal{L}_C$.
- $\mathsf{Verify}(\mathsf{vk}, x, \pi) \to b$. On input a verification key vk, an input x, and a proof π, the *verifier* Verify outputs $b = 1$ if he is convinced that $x \in \mathcal{L}_C$.

A zk-SNARK satisfies the following properties.

Completeness. For every security parameter λ, any $\mathbb{F}$-arithmetic circuit C, and any $(x,a) \in \mathcal{R}_C$, the honest prover

[6]A gate with inputs $y_1, \ldots, y_m \in \mathbb{F}$ is *bilinear* if the output is $\langle \vec{a}, (1, y_1, \ldots, y_m)\rangle \cdot \langle \vec{b}, (1, y_1, \ldots, y_m)\rangle$ for some $\vec{a}, \vec{b} \in \mathbb{F}^{m+1}$. These include addition, multiplication, negation, and constant gates.

can convince the verifier. Namely, $b = 1$ with probability $1 - \mathrm{negl}(\lambda)$ in the following experiment: $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(1^\lambda, C)$; $\pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, a)$; $b \leftarrow \mathsf{Verify}(\mathsf{vk}, x, \pi)$.

Succinctness. An honestly-generated proof π has $O_\lambda(1)$ bits and $\mathsf{Verify}(\mathsf{vk}, x, \pi)$ runs in time $O_\lambda(|x|)$. (Here, O_λ hides a fixed polynomial factor in λ.)

Proof of knowledge (and soundness). If the verifier accepts a proof output by a bounded prover, then the prover "knows" a witness for the given instance. (In particular, soundness holds against bounded provers.) Namely, for every $\mathrm{poly}(\lambda)$-size adversary $\mathcal{A}$, there is a $\mathrm{poly}(\lambda)$-size extractor $\mathcal{E}$ such that $\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1$ and $(x, a) \notin \mathcal{R}_C$ with probability $\mathrm{negl}(\lambda)$ in the following experiment: $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(1^\lambda, C)$; $(x, \pi) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{vk})$; $a \leftarrow \mathcal{E}(\mathsf{pk}, \mathsf{vk})$.

Perfect zero knowledge. An honestly-generated proof is perfect zero knowledge.[7] Namely, there is a $\mathrm{poly}(\lambda)$-size simulator Sim such that for all stateful $\mathrm{poly}(\lambda)$-size distinguishers $\mathcal{D}$ the following two probabilities are equal:

- The probability that $\mathcal{D}(\pi) = 1$ on an honest proof.

$$\Pr\left[\begin{array}{c} (x,a) \in \mathcal{R}_C \\ \mathcal{D}(\pi) = 1 \end{array} \middle| \begin{array}{r} (\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{KeyGen}(C) \\ (x,a) \leftarrow \mathcal{D}(\mathsf{pk},\mathsf{vk}) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk},x,a) \end{array}\right]$$

- The probability that $\mathcal{D}(\pi) = 1$ on a simulated proof.

$$\Pr\left[\begin{array}{c} (x,a) \in \mathcal{R}_C \\ \mathcal{D}(\pi) = 1 \end{array} \middle| \begin{array}{r} (\mathsf{pk},\mathsf{vk},\mathsf{trap}) \leftarrow \mathsf{Sim}(C) \\ (x,a) \leftarrow \mathcal{D}(\mathsf{pk},\mathsf{vk}) \\ \pi \leftarrow \mathsf{Sim}(\mathsf{pk},x,\mathsf{trap}) \end{array}\right]$$

B. Known constructions and security

There are many zk-SNARK constructions in the literature [9, 10, 11, 12, 13, 14, 15, 16]. We are interested in zk-SNARKs for arithmetic circuit satisfiability, and the most efficient ones for this language are based on *quadratic arithmetic programs* [12, 11, 13, 14, 16]; such constructions provide a linear-time KeyGen, quasilinear-time Prove, and linear-time Verify.

Security of zk-SNARKs is based on knowledge-of-exponent assumptions and variants of Diffie–Hellman assumptions in bilinear groups [9, 22, 23]. While knowledge-of-exponent assumptions are fairly strong, there is evidence that such assumptions may be inherent for constructing zk-SNARKs [24, 25].

C. zk-SNARK implementations

There are three published implementations of zk-SNARKs: (i) Parno et al. [13] present an implementation of zk-SNARKs for programs having no data dependencies;[8] (ii) Ben-Sasson et al. [14] present an implementation of zk-SNARKs for arbitrary programs (with data dependencies); and (iii) Ben-Sasson et al. [16] present an implementation of zk-SNARKs that supports programs that modify their own code (e.g., for runtime code generation); their implementation also reduces costs for programs of larger size and allows for universal key pairs.

Each of the works above also achieves zk-SNARKs for arithmetic circuit satisfiability as a stepping stone towards their respective higher-level efforts. In this paper we are only interested in a zk-SNARK for arithmetic circuit satisfiability, and we rely on the implementation of [16] for such a zk-SNARK.[9] The implementation in [16] is itself based on the protocol of Parno et al. [13]. We thus refer the interested reader to [13] for details of the protocol, its intuition, and its proof of security; and to [16] for the implementation and its performance. In terms of concrete parameters, the implementation of [16] provides 128 bits of security, and the field $\mathbb{F}$ is of a 256-bit prime order p.

III. Definition of a decentralized anonymous payment scheme

We introduce the notion of a *decentralized anonymous payment scheme* (DAP scheme), extending the notion of *decentralized e-cash* [8]. Later, in Section IV, we provide a construction.

A. Data structures

We begin by describing, and giving intuition about, the data structures used by a DAP scheme. The algorithms that use and produce these data structures are introduced in Section III-B.

Basecoin ledger. Our protocol is applied on top of a ledger-based base currency such as Bitcoin; for generality we refer to this base currency as *Basecoin*. At any given time T, all users have access to L_T, the *ledger* at time T, which is a sequence of *transactions*. The ledger is append-only (i.e., $T < T'$ implies that L_T is a prefix of $L_{T'}$).[10] The transactions in the ledger include both Basecoin transactions as well as two new transaction types described below.

Public parameters. A list of *public parameters* pp is available to all users in the system. These are generated by a trusted party at the "start of time" and are used by the system's algorithms.

Addresses. Each user generates at least one *address key pair* $(\mathsf{addr_{pk}}, \mathsf{addr_{sk}})$. The public key $\mathsf{addr_{pk}}$ is published and enables others to direct payments to the user. The secret key $\mathsf{addr_{sk}}$ is used to receive payments sent to $\mathsf{addr_{pk}}$. A user may generate any number of address key pairs.

Coins. A *coin* is a data object $\mathbf{c}$, to which we associate the following:

- A *coin commitment*, denoted $\mathsf{cm}(\mathbf{c})$: a string that appears on the ledger once $\mathbf{c}$ is *minted*.

[7] While most zk-SNARK descriptions in the literature only mention statistical zero knowledge, all zk-SNARK constructions can be made perfect zero knowledge by allowing for a negligible error probability in completeness.

[8] They only support programs where array indices are restricted to be known compile-time constants; similarly, loop iteration counts (or at least upper bounds to these) must be known at compile time.

[9] In [16], one optimization to the verifier's runtime requires preprocessing the verification key vk; for simplicity, we do not use this optimization.

[10] In reality, the Basecoin ledger (such as the one of Bitcoin) is not perfect and may incur temporary inconsistencies. In this respect our construction is as good as the underlying ledger. We discuss the effects of this on anonymity and mitigations in Section VI-C.

- A *coin value*, denoted $v(\mathbf{c})$: the denomination of $\mathbf{c}$, as measured in basecoins, as an integer between 0 and a maximum value v_{max} (which is a system parameter).
- A *coin serial number*, denoted $\mathsf{sn}(\mathbf{c})$: a unique string associated with the $\mathbf{c}$, used to prevent double spending.
- A *coin address*, denoted $\mathsf{addr}_{\mathsf{pk}}(\mathbf{c})$: an address public key, representing who owns $\mathbf{c}$.

Any other quantities associated with a coin $\mathbf{c}$ (e.g., various trapdoors) are implementation details.

New transactions. Besides Basecoin transactions, there are two new types of transactions.

- *Mint transactions.* A mint transaction $\mathsf{tx}_{\mathsf{Mint}}$ is a tuple $(\mathsf{cm}, v, *)$, where cm is a coin commitment, v is a coin value, and $*$ denotes other (implementation-dependent) information. The transaction $\mathsf{tx}_{\mathsf{Mint}}$ records that a coin $\mathbf{c}$ with coin commitment cm and value v has been minted.
- *Pour transactions.* A pour transaction $\mathsf{tx}_{\mathsf{Pour}}$ is a tuple $(\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, \mathsf{info}, *)$, where rt is a root of a Merkle tree, $\mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}$ are two coin serial numbers, $\mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}$ are two coin commitments, v_{pub} is a coin value, info is an arbitrary string, and $*$ denotes other (implementation-dependent) information. The transaction $\mathsf{tx}_{\mathsf{Pour}}$ records the pouring of two input (and now consumed) coins $\mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}$, with respective serial numbers $\mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}$, into two new output coins $\mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$, with respective coin commitments $\mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}$, as well as a public output v_{pub} (which may be zero). Furthermore, $\mathsf{tx}_{\mathsf{Pour}}$ also records an information string info (perhaps containing information on who is the recipient of v_{pub} basecoins) and that, when this transaction was made, the root of the Merkle tree over coin commitments was rt (see below).

Commitments of minted coins and serial numbers of spent coins. For any given time T,

- CMList_T denotes the list of all coin commitments appearing in mint and pour transactions in L_T;
- SNList_T denotes the list of all serial numbers appearing in pour transactions in L_T.

While both of these lists can be deduced from L_T, it will be convenient to think about them as separate (as, in practice, these may be separately maintained due to efficiency reasons).

Merkle tree over commitments. For any given time T, Tree_T denotes a Merkle tree over CMList_T and rt_T its root. Moreover, the function $\mathsf{Path}_T(\mathsf{cm})$ gives the authentication path from a coin commitment cm appearing in CMList_T to the root of Tree_T.[11] For convenience, we assume that L_T also stores $\mathsf{rt}_{T'}$ for all $T' \leq T$ (i.e., it stores all past Merkle tree roots).

B. Algorithms

A DAP scheme Π is a tuple of polynomial-time algorithms

$$(\mathsf{Setup}, \mathsf{CreateAddress}, \mathsf{Mint}, \mathsf{Pour}, \mathsf{VerifyTransaction}, \mathsf{Receive})$$

with the following syntax and semantics.

System setup. The algorithm Setup generates a list of public parameters:

Setup
- INPUTS: security parameter λ
- OUTPUTS: public parameters pp

The algorithm Setup is executed by a trusted party. The resulting public parameters pp are published and made available to all parties (e.g., by embedding them into the protocol's implementation). The setup is done *only once*; afterwards, no trusted party is needed, and no global secrets or trapdoors are kept.

Creating payment addresses. The algorithm $\mathsf{CreateAddress}$ generates a new address key pair:

CreateAddress
- INPUTS: public parameters pp
- OUTPUTS: address key pair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$

Each user generates at least one address key pair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ in order to receive coins. The public key $\mathsf{addr}_{\mathsf{pk}}$ is published, while the secret key $\mathsf{addr}_{\mathsf{sk}}$ is used to redeem coins sent to $\mathsf{addr}_{\mathsf{pk}}$. A user may generate any number of address key pairs; doing so does not require any interaction.

Minting coins. The algorithm Mint generates a coin (of a given value) and a mint transaction:

Mint
- INPUTS:
  - public parameters pp
  - coin value $v \in \{0, 1, \ldots, v_{\mathsf{max}}\}$
  - destination address public key $\mathsf{addr}_{\mathsf{pk}}$
- OUTPUTS: coin $\mathbf{c}$ and mint transaction $\mathsf{tx}_{\mathsf{Mint}}$

A system parameter, v_{max}, caps the value of any single coin. The output coin $\mathbf{c}$ has value v and coin address $\mathsf{addr}_{\mathsf{pk}}$; the output mint transaction $\mathsf{tx}_{\mathsf{Mint}}$ equals $(\mathsf{cm}, v, *)$, where cm is the coin commitment of $\mathbf{c}$.

Pouring coins. The Pour algorithm transfers value from input coins into new output coins, marking the input coins as consumed. Moreover, a fraction of the input value may be publicly revealed. Pouring allows users to subdivide coins into smaller denominations, merge coins, and transfer ownership of anonymous coins, or make public payments.[12]

Pour
- INPUTS:
  - public parameters pp
  - the Merkle root rt
  - old coins $\mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}$
  - old addresses secret keys $\mathsf{addr}_{\mathsf{sk},1}^{\mathsf{old}}, \mathsf{addr}_{\mathsf{sk},2}^{\mathsf{old}}$
  - authentication path path_1 from commitment $\mathsf{cm}(\mathbf{c}_1^{\mathsf{old}})$ to root rt,

[11] While we refer to Mekle trees for simplicity, it is straightforward to extend the definition to allow other data structures representing sets with fast insertion and short proofs of membership.

[12] We consider pours with 2 inputs and 2 outputs, for simplicity and (as discussed in Section I-B) without loss of generality.

  authentication path path_2 from commitment $\mathsf{cm}(\mathbf{c}_2^{\mathsf{old}})$ to root rt
  - new values $v_1^{\mathsf{new}}, v_2^{\mathsf{new}}$
  - new addresses public keys $\mathsf{addr}_{\mathsf{pk},1}^{\mathsf{new}}, \mathsf{addr}_{\mathsf{pk},2}^{\mathsf{new}}$
  - public value v_{pub}
  - transaction string info
- OUTPUTS: new coins $\mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$ and pour transaction $\mathsf{tx}_{\mathsf{Pour}}$

Thus, the Pour algorithm takes as input two distinct input coins $\mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}$, along with corresponding address secret keys $\mathsf{addr}_{\mathsf{sk},1}^{\mathsf{old}}, \mathsf{addr}_{\mathsf{sk},2}^{\mathsf{old}}$ (required to redeem the two input coins). To ensure that $\mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}$ have been previously minted, the Pour algorithm also takes as input the Merkle root rt (allegedly, equal to the root of Merkle tree over all coin commitments so far), along with two authentication paths $\mathsf{path}_1, \mathsf{path}_2$ for the two coin commitments $\mathsf{cm}(\mathbf{c}_1^{\mathsf{old}}), \mathsf{cm}(\mathbf{c}_2^{\mathsf{old}})$. Two input values $v_1^{\mathsf{new}}, v_2^{\mathsf{new}}$ specify the values of two new anonymous coins $\mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$ to be generated, and two input address public keys $\mathsf{addr}_{\mathsf{pk},1}^{\mathsf{new}}, \mathsf{addr}_{\mathsf{pk},2}^{\mathsf{new}}$ specify the recipients of $\mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$. A third value, v_{pub}, specifies the amount to be publicly spent (e.g., to redeem coins or pay transaction fees). The sum of output values $v_1 + v_2 + v_{\mathsf{pub}}$ must be equal to the sum of the values of the input coins (and cannot exceed v_{max}). Finally, the Pour algorithm also receives an arbitrary string info, which is bound into the output pour transaction $\mathsf{tx}_{\mathsf{Pour}}$.

The Pour algorithm outputs two new coins $\mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$ and a pour transaction $\mathsf{tx}_{\mathsf{Pour}}$. The transaction $\mathsf{tx}_{\mathsf{Pour}}$ equals $(\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, \mathsf{info}, *)$, where $\mathsf{cm}_1^{\mathsf{new}}$, $\mathsf{cm}_2^{\mathsf{new}}$ are the two coin commitments of the two output coins, and $*$ denotes other (implementation-dependent) information. Crucially, $\mathsf{tx}_{\mathsf{Pour}}$ reveals only one currency value, the public value v_{pub} (which may be zero); it does not reveal the payment addresses or values of the old or new coins.

Verifying transactions. The algorithm VerifyTransaction checks the validity of a transaction:

VerifyTransaction
- INPUTS:
  - public parameters pp
  - a (mint or pour) transaction tx
  - the current ledger L
- OUTPUTS: bit b, equals 1 iff the transaction is valid

Both mint and pour transactions must be verified before being considered well-formed. In practice, transactions can be verified by the nodes in the distributed system maintaining the ledger, as well as by users who rely on these transactions.

Receiving coins. The algorithm Receive scans the ledger and retrieves unspent coins paid to a particular user address:

Receive
- INPUTS:
  - recipient address key pair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$
  - the current ledger L
- OUTPUTS: set of (unspent) received coins

When a user with address key pair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ wishes to receive payments sent to $\mathsf{addr}_{\mathsf{pk}}$, he uses the Receive algorithm to scan the ledger. For each payment to $\mathsf{addr}_{\mathsf{pk}}$ appearing in the ledger, Receive outputs the corresponding coins whose serial numbers do not appear on the ledger L. Coins received in this way may be spent, just like minted coins, using the Pour algorithm. (We only require Receive to detect coins paid to $\mathsf{addr}_{\mathsf{pk}}$ via the Pour algorithm and not also detect coins minted by the user himself.)

Next, we describe completeness (Section III-C) and security (Section III-D).

C. Completeness

Completeness of a DAP scheme requires that unspent coins can be spent. More precisely, consider a *ledger sampler* $\mathcal{S}$ outputting a ledger L. If $\mathbf{c}_1$ and $\mathbf{c}_2$ are two coins whose coin commitments appear in (valid) transactions on L, but their serial numbers do not appear in L, then $\mathbf{c}_1$ and $\mathbf{c}_2$ can be spent using Pour. Namely, running Pour results in a pour transaction $\mathsf{tx}_{\mathsf{Pour}}$ that VerifyTransaction accepts, and the new coins can be received by the intended recipients (by using Receive); moreover, $\mathsf{tx}_{\mathsf{Pour}}$ correctly records the intended v_{pub} and transaction string info. This property is formalized via an *incompleteness experiment* INCOMP.

Definition III.1. A DAP scheme $\Pi = (\mathsf{Setup}, \mathsf{CreateAddress}, \mathsf{Mint}, \mathsf{Pour}, \mathsf{VerifyTransaction}, \mathsf{Receive})$ is **complete** if no polynomial-size ledger sampler $\mathcal{S}$ wins INCOMP with more than negligible probability.

D. Security

Security of a DAP scheme is characterized by three properties, which we call *ledger indistinguishability*, *transaction non-malleability*, and *balance*.

Definition III.2. A DAP scheme $\Pi = (\mathsf{Setup}, \mathsf{CreateAddress}, \mathsf{Mint}, \mathsf{Pour}, \mathsf{VerifyTransaction}, \mathsf{Receive})$ is **secure** if it satisfies ledger indistinguishability, transaction non-malleability, and balance.

Below, we provide an informal overview of each property, and defer formal definitions to the extended version of this paper [26].

Each property is formalized as a game between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. In each game, the behavior of honest parties is realized via a DAP scheme oracle $\mathcal{O}^{\mathsf{DAP}}$, which maintains a ledger L and provides an interface for executing CreateAddress, Mint, Pour and Receive algorithms for honest parties. To elicit behavior from honest parties, $\mathcal{A}$ passes a query to $\mathcal{C}$, which (after sanity checks) proxies the query to $\mathcal{O}^{\mathsf{DAP}}$. For each query that requests an honest party to perform an action, $\mathcal{A}$ specifies identities of previous transactions and the input values, and learns the resulting transaction, but not any of the secrets or trapdoors involved in producing that transaction. The oracle $\mathcal{O}^{\mathsf{DAP}}$ also provides an **Insert** query that allows $\mathcal{A}$ to directly add aribtrary transactions to the ledger L.

Ledger indistinguishability. This property captures the requirement that the ledger reveals no new information to the adversary beyond the publicly-revealed information (values of minted coins, public values, information strings, total number of transactions, etc.), even when the adversary can adaptively induce honest parties to perform DAP operations of his choice. That is, no bounded adversary $\mathcal{A}$ can distinguish between two ledgers L_0 and L_1, constructed by $\mathcal{A}$ using queries to two DAP scheme oracles, when the queries to the two oracles are *publicly consistent*: they have matching type and are identical in terms of publicly-revealed information and the information related to addresses controlled by $\mathcal{A}$.

Ledger indistinguishability is formalized by an experiment $\mathsf{L\text{-}IND}$ that proceeds as follows. First, a challenger samples a random bit b and initializes two DAP scheme oracles $\mathcal{O}_0^{\mathsf{DAP}}$ and $\mathcal{O}_1^{\mathsf{DAP}}$, maintaining ledgers L_0 and L_1. Throughout, the challenger allows $\mathcal{A}$ to issue queries to $\mathcal{O}_0^{\mathsf{DAP}}$ and $\mathcal{O}_1^{\mathsf{DAP}}$, thus controlling the behavior of honest parties on L_0 and L_1. The challenger provides the adversary with the view of both ledgers, but in randomized order: $L_{\mathsf{Left}} := L_b$ and $L_{\mathsf{Right}} := L_{1-b}$. The adversary's goal is to distinguish whether the view he sees corresponds to $(L_{\mathsf{Left}}, L_{\mathsf{Right}}) = (L_0, L_1)$, i.e. $b = 0$, or to $(L_{\mathsf{Left}}, L_{\mathsf{Right}}) = (L_1, L_0)$, i.e. $b = 1$.

At each round of the experiment, the adversary issues queries in pairs Q, Q' of matching query type. If the query type is **CreateAddress**, then the same address is generated at both oracles. If it is to **Mint**, **Pour** or **Receive**, then Q is forwarded to L_0 and Q' to L_1; for **Insert** queries, query Q is forwarded to L_{Left} and Q' is forwarded to L_{Right}. The adversary's queries are restricted in the sense that they must maintain the *public consistency* of the two ledgers. For example, the public values for **Pour** queries must be the same, as well as minted amounts for **Mint** queries.

At the conclusion of the experiment, $\mathcal{A}$ outputs a guess b', and wins when $b = b'$. Ledger indistinguishability requires that $\mathcal{A}$ wins $\mathsf{L\text{-}IND}$ with probability at most negligibly greater than $1/2$.

Transaction non-malleability. This property requires that no bounded adversary $\mathcal{A}$ can alter any of the data stored within a (valid) pour transaction $\mathsf{tx}_{\mathsf{Pour}}$. This *transaction non-malleability* prevents malicious attackers from modifying others' transactions before they are added to the ledger (e.g., by re-targeting the Basecoin public output of a pour transaction).

Transaction non-malleability is formalized by an experiment $\mathsf{TR\text{-}NM}$, in which $\mathcal{A}$ adaptively interacts with a DAP scheme oracle $\mathcal{O}^{\mathsf{DAP}}$ and then outputs a pour transaction tx^*. Letting $\mathcal{T}$ denote the set of pour transactions returned by $\mathcal{O}^{\mathsf{DAP}}$, and L denote the final ledger, $\mathcal{A}$ wins the game if there exists $\mathsf{tx} \in \mathcal{T}$, such that (i) $\mathsf{tx}^* \neq \mathsf{tx}$; (ii) tx^* reveals a serial number contained in tx; and (iii) both tx and tx^* are valid with respect to the ledger L' containing all transactions preceding tx on L. In other words, $\mathcal{A}$ wins the game if tx^* manages to modify some previous pour transaction to spend the same coin in a different way.

Transaction non-malleability requires that $\mathcal{A}$ wins $\mathsf{TR\text{-}NM}$ with only negligible probability. (Note that $\mathcal{A}$ can of course produce valid pour transactions that are unrelated to those in $\mathcal{T}$; the condition that tx^* reveals a serial number of a previously-spent coin captures non-malleability.)

Balance. This property requires that no bounded adversary $\mathcal{A}$ can own more money than what he minted or received via payments from others.

Balance is formalized by an experiment BAL, in which $\mathcal{A}$ adaptively interacts with a DAP scheme oracle $\mathcal{O}^{\mathsf{DAP}}$ and then outputs a set of coins S_{coin}. Letting S_{addr} be set of addresses returned by **CreateAddress** queries (i.e., addresses of "honest" users), $\mathcal{A}$ wins the game if the total value he can spend or has spent (either as coins or $Basecoin$ public outputs) is greater than the value he has received or mined. That is, $\mathcal{A}$ wins if $v_{\mathsf{Unspent}} + v_{\mathsf{Basecoin}} + v_{\mathcal{A}\to\mathsf{ADDR}} > v_{\mathsf{Mint}} + v_{\mathsf{ADDR}\to\mathcal{A}}$ where: (i) v_{Unspent} is the total value of unspent coins in S_{coin}; (ii) v_{Basecoin} is the total value of public outputs placed by $\mathcal{A}$ on the ledger; (iii) v_{Mint} is the total value of $\mathcal{A}$'s mint transactions; (iv) $v_{\mathsf{ADDR}\to\mathcal{A}}$ is the total value of payments received by $\mathcal{A}$ from addresses in S_{addr}; (v) $v_{\mathcal{A}\to\mathsf{ADDR}}$ is the total value of payments sent by $\mathcal{A}$ to addresses in S_{addr}.

Balance requires that $\mathcal{A}$ wins BAL with only negligible probability.

IV. Construction of a decentralized anonymous payment scheme

We show how to construct a DAP scheme (introduced in Section III) using zk-SNARKs and other building blocks. Later, in Section V, we give a concrete instantiation of this construction.

A. Cryptographic building blocks

We first introduce notation for the standard cryptographic building blocks that we use. We assume familiarity with the definitions of these building blocks; for more details, see, e.g., [27]. Throughout, λ denotes the security parameter.

Collision-resistant hashing. We use a collision-resistant hash function $\mathsf{CRH}\colon \{0,1\}^* \to \{0,1\}^{O(\lambda)}$.

Pseudorandom functions. We use a pseudorandom function family $\mathsf{PRF} = \{\mathsf{PRF}_x\colon \{0,1\}^* \to \{0,1\}^{O(\lambda)}\}_x$ where x denotes the seed. From PRF_x, we derive three "non-overlapping" pseudorandom functions, chosen arbitrarily as $\mathsf{PRF}_x^{\mathrm{addr}}(z) := \mathsf{PRF}_x(00\|z)$, $\mathsf{PRF}_x^{\mathrm{sn}}(z) := \mathsf{PRF}_x(01\|z)$, $\mathsf{PRF}_x^{\mathrm{pk}}(z) := \mathsf{PRF}_x(10\|z)$. Furthermore, we assume that $\mathsf{PRF}^{\mathrm{sn}}$ is also collision resistant, in the sense that it is infeasible to find $(x,z) \neq (x',z')$ such that $\mathsf{PRF}_x^{\mathrm{sn}}(z) = \mathsf{PRF}_{x'}^{\mathrm{sn}}(z')$.

Statistically-hiding commitments. We use a commitment scheme COMM where the binding property holds computationally, while the hiding property holds statistically. It is denoted $\{\mathsf{COMM}_x\colon \{0,1\}^* \to \{0,1\}^{O(\lambda)}\}_x$ where x denotes the commitment trapdoor. Namely, to reveal a commitment cm to a value z, it suffices to provide z and the trapdoor x; then one can check that $\mathsf{cm} = \mathsf{COMM}_x(z)$.

One-time strongly-unforgeable digital signatures. We use a digital signature scheme $\mathsf{Sig} = (\mathcal{G}_{\mathsf{sig}}, \mathcal{K}_{\mathsf{sig}}, \mathcal{S}_{\mathsf{sig}}, \mathcal{V}_{\mathsf{sig}})$ that works as follows.

- $\mathcal{G}_{\mathsf{sig}}(1^\lambda) \to \mathsf{pp}_{\mathsf{sig}}$. Given a security parameter λ (presented in unary), $\mathcal{G}_{\mathsf{sig}}$ samples public parameters $\mathsf{pp}_{\mathsf{enc}}$ for the encryption scheme.
- $\mathcal{K}_{\mathsf{sig}}(\mathsf{pp}_{\mathsf{sig}}) \to (\mathsf{pk}_{\mathsf{sig}}, \mathsf{sk}_{\mathsf{sig}})$. Given public parameters $\mathsf{pp}_{\mathsf{sig}}$, $\mathcal{K}_{\mathsf{sig}}$ samples a public key and a secret key for a single user.
- $\mathcal{S}_{\mathsf{sig}}(\mathsf{sk}_{\mathsf{sig}}, m) \to \sigma$. Given a secret key $\mathsf{sk}_{\mathsf{sig}}$ and a message m, $\mathcal{S}_{\mathsf{sig}}$ signs m to obtain a signature σ.
- $\mathcal{V}_{\mathsf{sig}}(\mathsf{pk}_{\mathsf{sig}}, m, \sigma) \to b$. Given a public key $\mathsf{pk}_{\mathsf{sig}}$, message m, and signature σ, $\mathcal{V}_{\mathsf{sig}}$ outputs $b = 1$ if the signature σ is valid for message m; else it outputs $b = 0$.

The signature scheme Sig satisfies the security property of *one-time strong unforgeability against chosen-message attacks* ($\mathsf{SUF\text{-}1CMA}$ security).

Key-private public-key encryption. We use a public-key encryption scheme $\mathsf{Enc} = (\mathcal{G}_{\mathsf{enc}}, \mathcal{K}_{\mathsf{enc}}, \mathcal{E}_{\mathsf{enc}}, \mathcal{D}_{\mathsf{enc}})$ that works as follows.

- $\mathcal{G}_{\mathsf{enc}}(1^\lambda) \to \mathsf{pp}_{\mathsf{enc}}$. Given a security parameter λ (presented in unary), $\mathcal{G}_{\mathsf{enc}}$ samples public parameters $\mathsf{pp}_{\mathsf{enc}}$ for the encryption scheme.
- $\mathcal{K}_{\mathsf{enc}}(\mathsf{pp}_{\mathsf{enc}}) \to (\mathsf{pk}_{\mathsf{enc}}, \mathsf{sk}_{\mathsf{enc}})$. Given public parameters $\mathsf{pp}_{\mathsf{enc}}$, $\mathcal{K}_{\mathsf{enc}}$ samples a public key and a secret key for a single user.
- $\mathcal{E}_{\mathsf{enc}}(\mathsf{pk}_{\mathsf{enc}}, m) \to c$. Given a public key $\mathsf{pk}_{\mathsf{enc}}$ and a message m, $\mathcal{E}_{\mathsf{enc}}$ encrypts m to obtain a ciphertext c.
- $\mathcal{D}_{\mathsf{enc}}(\mathsf{sk}_{\mathsf{enc}}, c) \to m$. Given a secret key $\mathsf{sk}_{\mathsf{enc}}$ and a ciphertext c, $\mathcal{D}_{\mathsf{enc}}$ decrypts c to produce a message m (or $\perp$ if decryption fails).

The encryption scheme Enc satisfies two security properties: (i) *ciphertext indistinguishability under chosen-ciphertext attack* ($\mathsf{IND\text{-}CCA}$ security); and (ii) *key indistinguishability under chosen-ciphertext attack* ($\mathsf{IK\text{-}CCA}$ security). While the first property is standard, the second is less known; informally, $\mathsf{IK\text{-}CCA}$ requires that ciphertexts cannot be linked to the public key used to encrypt them, or to other ciphertexts encrypted with the same public key. For definitions, we refer the reader to [21].

B. zk-SNARKs for pouring coins

As outlined in Section I-B, our construction invokes a zk-SNARK for a specific NP statement, POUR, which we now define. We first recall the context motivating POUR. When a user u *pours* "old" coins $\mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}$ into new coins $\mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$, a corresponding pour transaction

$$\mathsf{tx}_{\mathsf{Pour}} = (\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, \mathsf{info}, *)$$

is generated. In our construction, we need to provide evidence in "$*$" that various conditions were respected by the pour operation. Concretely, $\mathsf{tx}_{\mathsf{Pour}}$ should demonstrate that (i) u owns $\mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}$; (ii) coin commitments for $\mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}$ appear somewhere on the ledger; (iii) the revealed serial numbers $\mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}$ are of $\mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}$; (iv) the revealed coin commitments $\mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}$ are of $\mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$; (v) balance is preserved. Our construction achieves this by including a zk-SNARK proof $\pi_{\mathtt{POUR}}$ for the statement POUR which checks the above invariants (as well as others needed for non-malleability).

The statement POUR. Concretely, the NP statement POUR is defined as follows.

- Instances are of the form $\vec{x} = (\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, h_{\mathsf{Sig}}, h_1, h_2)$. Thus, an instance $\vec{x}$ specifies a root rt for a CRH-based Merkle tree (over the list of commitments so far), the two serial numbers of the consumed coins, two coin commitments for the two new coins, a public value, and fields $h_{\mathsf{Sig}}, h_1, h_2$ used for non-malleability.
- Witnesses are of the form $\vec{a} = (\mathsf{path}_1, \mathsf{path}_2, \mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}, \mathsf{addr}_{\mathsf{sk},1}^{\mathsf{old}}, \mathsf{addr}_{\mathsf{sk},2}^{\mathsf{old}}, \mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}})$ where, for each $i \in \{1, 2\}$:

$$\begin{aligned} \mathbf{c}_i^{\mathsf{old}} &= (\mathsf{addr}_{\mathsf{pk},i}^{\mathsf{old}}, v_i^{\mathsf{old}}, \rho_i^{\mathsf{old}}, r_i^{\mathsf{old}}, s_i^{\mathsf{old}}, \mathsf{cm}_i^{\mathsf{old}}) \ , \\ \mathbf{c}_i^{\mathsf{new}} &= (\mathsf{addr}_{\mathsf{pk},i}^{\mathsf{new}}, v_i^{\mathsf{new}}, \rho_i^{\mathsf{new}}, r_i^{\mathsf{new}}, s_i^{\mathsf{new}}, \mathsf{cm}_i^{\mathsf{new}}) \\ &\text{for the same } \mathsf{cm}_i^{\mathsf{new}} \text{ as in } \vec{x}, \end{aligned}$$

$$\begin{aligned} \mathsf{addr}_{\mathsf{pk},i}^{\mathsf{old}} &= (a_{\mathsf{pk},i}^{\mathsf{old}}, \mathsf{pk}_{\mathsf{enc},i}^{\mathsf{old}}) \ , \\ \mathsf{addr}_{\mathsf{pk},i}^{\mathsf{new}} &= (a_{\mathsf{pk},i}^{\mathsf{new}}, \mathsf{pk}_{\mathsf{enc},i}^{\mathsf{new}}) \ , \\ \mathsf{addr}_{\mathsf{sk},i}^{\mathsf{old}} &= (a_{\mathsf{sk},i}^{\mathsf{old}}, \mathsf{sk}_{\mathsf{enc},i}^{\mathsf{old}}) \ . \end{aligned}$$

  Thus, a witness $\vec{a}$ specifies authentication paths for the two new coin commitments, the entirety of coin information about both the old and new coins, and address secret keys for the old coins.

Given a POUR instance $\vec{x}$, a witness $\vec{a}$ is valid for $\vec{x}$ if the following holds:

1) For each $i \in \{1, 2\}$:
   a) The coin commitment $\mathsf{cm}_i^{\mathsf{old}}$ of $\mathbf{c}_i^{\mathsf{old}}$ appears on the ledger, i.e., path_i is a valid authentication path for leaf $\mathsf{cm}_i^{\mathsf{old}}$ with respect to root rt, in a CRH-based Merkle tree.
   b) The address secret key $a_{\mathsf{sk},i}^{\mathsf{old}}$ matches the address public key of $\mathbf{c}_i^{\mathsf{old}}$, i.e., $a_{\mathsf{pk},i}^{\mathsf{old}} = \mathsf{PRF}^{\mathrm{addr}}_{a_{\mathsf{sk},i}^{\mathsf{old}}}(0)$.
   c) The serial number $\mathsf{sn}_i^{\mathsf{old}}$ of $\mathbf{c}_i^{\mathsf{old}}$ is computed correctly, i.e., $\mathsf{sn}_i^{\mathsf{old}} = \mathsf{PRF}^{\mathrm{sn}}_{a_{\mathsf{sk},i}^{\mathsf{old}}}(\rho_i^{\mathsf{old}})$.
   d) The coin $\mathbf{c}_i^{\mathsf{old}}$ is well-formed, i.e., $\mathsf{cm}_i^{\mathsf{old}} = \mathsf{COMM}_{s_i^{\mathsf{old}}}(\mathsf{COMM}_{r_i^{\mathsf{old}}}(a_{\mathsf{pk},i}^{\mathsf{old}} \| \rho_i^{\mathsf{old}}) \| v_i^{\mathsf{old}})$.
   e) The coin $\mathbf{c}_i^{\mathsf{new}}$ is well-formed, i.e., $\mathsf{cm}_i^{\mathsf{new}} = \mathsf{COMM}_{s_i^{\mathsf{new}}}(\mathsf{COMM}_{r_i^{\mathsf{new}}}(a_{\mathsf{pk},i}^{\mathsf{new}} \| \rho_i^{\mathsf{new}}) \| v_i^{\mathsf{new}})$.
   f) The address secret key $a_{\mathsf{sk},i}^{\mathsf{old}}$ ties h_{Sig} to h_i, i.e., $h_i = \mathsf{PRF}^{\mathrm{pk}}_{a_{\mathsf{sk},i}^{\mathsf{old}}}(h_{\mathsf{Sig}})$.
2) Balance is preserved: $v_1^{\mathsf{new}} + v_2^{\mathsf{new}} + v_{\mathsf{pub}} = v_1^{\mathsf{old}} + v_2^{\mathsf{old}}$ (with $v_1^{\mathsf{old}}, v_2^{\mathsf{old}} \geq 0$ and $v_1^{\mathsf{old}} + v_2^{\mathsf{old}} \leq v_{\mathsf{max}}$).

Recall that in this paper zk-SNARKs are relative to the language of arithmetic circuit satisfiability (see Section II); thus, we express the checks in POUR via an arithmetic circuit, denoted $C_{\mathtt{POUR}}$. In particular, the depth d_{tree} of the Merkle tree needs to be hardcoded in $C_{\mathtt{POUR}}$, and we thus make it a parameter of our construction (see below); the maximum number of supported coins is then $2^{d_{\mathsf{tree}}}$.

C. Algorithm constructions

We proceed to describe the construction of the DAP scheme $\Pi = (\mathsf{Setup}, \mathsf{CreateAddress}, \mathsf{Mint}, \mathsf{Pour}, \mathsf{VerifyTransaction}, \mathsf{Receive})$ whose intuition was given in Section I-B. Figure 2 gives the pseudocode for each one of the six algorithms in Π, in terms of the building blocks introduced in Section IV-A and Section IV-B. In the construction, we hardcode two quantities:

```
Setup
• INPUTS: security parameter $\lambda$
• OUTPUTS: public parameters $\mathsf{pp}$
1) Construct $C_{\mathtt{POUR}}$ for POUR at security $\lambda$.
2) Compute $(\mathsf{pk}_{\mathtt{POUR}}, \mathsf{vk}_{\mathtt{POUR}}) := \mathsf{KeyGen}(1^\lambda, C_{\mathtt{POUR}})$.
3) Compute $\mathsf{pp}_{\mathsf{enc}} := \mathcal{G}_{\mathsf{enc}}(1^\lambda)$.
4) Compute $\mathsf{pp}_{\mathsf{sig}} := \mathcal{G}_{\mathsf{sig}}(1^\lambda)$.
5) Set $\mathsf{pp} := (\mathsf{pk}_{\mathtt{POUR}}, \mathsf{vk}_{\mathtt{POUR}}, \mathsf{pp}_{\mathsf{enc}}, \mathsf{pp}_{\mathsf{sig}})$.
6) Output $\mathsf{pp}$.

CreateAddress
• INPUTS: public parameters $\mathsf{pp}$
• OUTPUTS: address key pair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$
1) Compute $(\mathsf{pk}_{\mathsf{enc}}, \mathsf{sk}_{\mathsf{enc}}) := \mathcal{K}_{\mathsf{enc}}(\mathsf{pp}_{\mathsf{enc}})$.
2) Randomly sample a $\mathsf{PRF}^{\mathrm{addr}}$ seed $a_{\mathsf{sk}}$.
3) Compute $a_{\mathsf{pk}} = \mathsf{PRF}^{\mathrm{addr}}_{a_{\mathsf{sk}}}(0)$.
4) Set $\mathsf{addr}_{\mathsf{pk}} := (a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}})$.
5) Set $\mathsf{addr}_{\mathsf{sk}} := (a_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}})$.
6) Output $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$.

Mint
• INPUTS:
  – public parameters $\mathsf{pp}$
  – coin value $v \in \{0, 1, \dots, v_{\mathsf{max}}\}$
  – destination address public key $\mathsf{addr}_{\mathsf{pk}}$
• OUTPUTS: coin $\mathbf{c}$ and mint transaction $\mathsf{tx}_{\mathsf{Mint}}$
1) Parse $\mathsf{addr}_{\mathsf{pk}}$ as $(a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}})$.
2) Randomly sample a $\mathsf{PRF}^{\mathrm{sn}}$ seed $\rho$.
3) Randomly sample two $\mathsf{COMM}$ trapdoors $r, s$.
4) Compute $k := \mathsf{COMM}_r(a_{\mathsf{pk}} \| \rho)$.
5) Compute $\mathsf{cm} := \mathsf{COMM}_s(v \| k)$.
6) Set $\mathbf{c} := (\mathsf{addr}_{\mathsf{pk}}, v, \rho, r, s, \mathsf{cm})$.
7) Set $\mathsf{tx}_{\mathsf{Mint}} := (\mathsf{cm}, v, *)$, where $* := (k, s)$.
8) Output $\mathbf{c}$ and $\mathsf{tx}_{\mathsf{Mint}}$.

VerifyTransaction
• INPUTS:
  – public parameters $\mathsf{pp}$
  – a (mint or pour) transaction $\mathsf{tx}$
  – the current ledger $L$
• OUTPUTS: bit $b$, equals 1 iff the transaction is valid
1) If given a mint transaction $\mathsf{tx} = \mathsf{tx}_{\mathsf{Mint}}$:
   a) Parse $\mathsf{tx}_{\mathsf{Mint}}$ as $(\mathsf{cm}, v, *)$, and $*$ as $(k, s)$.
   b) Set $\mathsf{cm}' := \mathsf{COMM}_s(v \| k)$.
   c) Output $b := 1$ if $\mathsf{cm} = \mathsf{cm}'$, else output $b := 0$.
2) If given a pour transaction $\mathsf{tx} = \mathsf{tx}_{\mathsf{Pour}}$:
   a) Parse $\mathsf{tx}_{\mathsf{Pour}}$ as $(\mathsf{rt}, \mathsf{sn}^{\mathsf{old}}_1, \mathsf{sn}^{\mathsf{old}}_2, \mathsf{cm}^{\mathsf{new}}_1, \mathsf{cm}^{\mathsf{new}}_2, v_{\mathsf{pub}}, \mathsf{info}, *)$, and $*$ as
      $(\mathsf{pk}_{\mathsf{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, \mathbf{C}_1, \mathbf{C}_2, \sigma)$.
   b) If $\mathsf{sn}^{\mathsf{old}}_1$ or $\mathsf{sn}^{\mathsf{old}}_2$ appears on $L$ (or $\mathsf{sn}^{\mathsf{old}}_1 = \mathsf{sn}^{\mathsf{old}}_2$), output $b := 0$.
   c) If the Merkle root $\mathsf{rt}$ does not appear on $L$, output $b := 0$.
   d) Compute $h_{\mathsf{Sig}} := \mathsf{CRH}(\mathsf{pk}_{\mathsf{sig}})$.
   e) Set $\vec{x} := (\mathsf{rt}, \mathsf{sn}^{\mathsf{old}}_1, \mathsf{sn}^{\mathsf{old}}_2, \mathsf{cm}^{\mathsf{new}}_1, \mathsf{cm}^{\mathsf{new}}_2, v_{\mathsf{pub}}, h_{\mathsf{Sig}}, h_1, h_2)$.
   f) Set $m := (\vec{x}, \pi_{\mathtt{POUR}}, \mathsf{info}, \mathbf{C}_1, \mathbf{C}_2)$
   g) Compute $b := \mathcal{V}_{\mathsf{sig}}(\mathsf{pk}_{\mathsf{sig}}, m, \sigma)$.
   h) Compute $b' := \mathsf{Verify}(\mathsf{vk}_{\mathtt{POUR}}, \vec{x}, \pi_{\mathtt{POUR}})$, and output $b \wedge b'$.
```

```
Pour
• INPUTS:
  – public parameters $\mathsf{pp}$
  – the Merkle root $\mathsf{rt}$
  – old coins $\mathbf{c}^{\mathsf{old}}_1, \mathbf{c}^{\mathsf{old}}_2$
  – old addresses secret keys $\mathsf{addr}^{\mathsf{old}}_{\mathsf{sk},1}, \mathsf{addr}^{\mathsf{old}}_{\mathsf{sk},2}$
  – path $\mathsf{path}_1$ from commitment $\mathsf{cm}(\mathbf{c}^{\mathsf{old}}_1)$ to root $\mathsf{rt}$,
    path $\mathsf{path}_2$ from commitment $\mathsf{cm}(\mathbf{c}^{\mathsf{old}}_2)$ to root $\mathsf{rt}$
  – new values $v^{\mathsf{new}}_1, v^{\mathsf{new}}_2$
  – new addresses public keys $\mathsf{addr}^{\mathsf{new}}_{\mathsf{pk},1}, \mathsf{addr}^{\mathsf{new}}_{\mathsf{pk},2}$
  – public value $v_{\mathsf{pub}}$
  – transaction string $\mathsf{info}$
• OUTPUTS: new coins $\mathbf{c}^{\mathsf{new}}_1, \mathbf{c}^{\mathsf{new}}_2$ and pour transaction $\mathsf{tx}_{\mathsf{Pour}}$
1) For each $i \in \{1, 2\}$:
   a) Parse $\mathbf{c}^{\mathsf{old}}_i$ as $(\mathsf{addr}^{\mathsf{old}}_{\mathsf{pk},i}, v^{\mathsf{old}}_i, \rho^{\mathsf{old}}_i, r^{\mathsf{old}}_i, s^{\mathsf{old}}_i, \mathsf{cm}^{\mathsf{old}}_i)$.
   b) Parse $\mathsf{addr}^{\mathsf{old}}_{\mathsf{sk},i}$ as $(a^{\mathsf{old}}_{\mathsf{sk},i}, \mathsf{sk}^{\mathsf{old}}_{\mathsf{enc},i})$.
   c) Compute $\mathsf{sn}^{\mathsf{old}}_i := \mathsf{PRF}^{\mathrm{sn}}_{a^{\mathsf{old}}_{\mathsf{sk},i}}(\rho^{\mathsf{old}}_i)$.
   d) Parse $\mathsf{addr}^{\mathsf{new}}_{\mathsf{pk},i}$ as $(a^{\mathsf{new}}_{\mathsf{pk},i}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i})$.
   e) Randomly sample a $\mathsf{PRF}^{\mathrm{sn}}$ seed $\rho^{\mathsf{new}}_i$.
   f) Randomly sample two $\mathsf{COMM}$ trapdoors $r^{\mathsf{new}}_i, s^{\mathsf{new}}_i$.
   g) Compute $k^{\mathsf{new}}_i := \mathsf{COMM}_{r^{\mathsf{new}}_i}(a^{\mathsf{new}}_{\mathsf{pk},i} \| \rho^{\mathsf{new}}_i)$.
   h) Compute $\mathsf{cm}^{\mathsf{new}}_i := \mathsf{COMM}_{s^{\mathsf{new}}_i}(v^{\mathsf{new}}_i \| k^{\mathsf{new}}_i)$.
   i) Set $\mathbf{c}^{\mathsf{new}}_i := (\mathsf{addr}^{\mathsf{new}}_{\mathsf{pk},i}, v^{\mathsf{new}}_i, \rho^{\mathsf{new}}_i, r^{\mathsf{new}}_i, s^{\mathsf{new}}_i, \mathsf{cm}^{\mathsf{new}}_i)$.
   j) Set $\mathbf{C}_i := \mathcal{E}_{\mathsf{enc}}(\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}, (v^{\mathsf{new}}_i, \rho^{\mathsf{new}}_i, r^{\mathsf{new}}_i, s^{\mathsf{new}}_i))$.
2) Generate $(\mathsf{pk}_{\mathsf{sig}}, \mathsf{sk}_{\mathsf{sig}}) := \mathcal{K}_{\mathsf{sig}}(\mathsf{pp}_{\mathsf{sig}})$.
3) Compute $h_{\mathsf{Sig}} := \mathsf{CRH}(\mathsf{pk}_{\mathsf{sig}})$.
4) Compute $h_1 := \mathsf{PRF}^{\mathrm{pk}}_{a^{\mathsf{old}}_{\mathsf{sk},1}}(h_{\mathsf{Sig}})$ and $h_2 := \mathsf{PRF}^{\mathrm{pk}}_{a^{\mathsf{old}}_{\mathsf{sk},2}}(h_{\mathsf{Sig}})$.
5) Set $\vec{x} := (\mathsf{rt}, \mathsf{sn}^{\mathsf{old}}_1, \mathsf{sn}^{\mathsf{old}}_2, \mathsf{cm}^{\mathsf{new}}_1, \mathsf{cm}^{\mathsf{new}}_2, v_{\mathsf{pub}}, h_{\mathsf{Sig}}, h_1, h_2)$.
6) Set $\vec{a} := (\mathsf{path}_1, \mathsf{path}_2, \mathbf{c}^{\mathsf{old}}_1, \mathbf{c}^{\mathsf{old}}_2, \mathsf{addr}^{\mathsf{old}}_{\mathsf{sk},1}, \mathsf{addr}^{\mathsf{old}}_{\mathsf{sk},2}, \mathbf{c}^{\mathsf{new}}_1, \mathbf{c}^{\mathsf{new}}_2)$.
7) Compute $\pi_{\mathtt{POUR}} := \mathsf{Prove}(\mathsf{pk}_{\mathtt{POUR}}, \vec{x}, \vec{a})$.
8) Set $m := (\vec{x}, \pi_{\mathtt{POUR}}, \mathsf{info}, \mathbf{C}_1, \mathbf{C}_2)$.
9) Compute $\sigma := \mathcal{S}_{\mathsf{sig}}(\mathsf{sk}_{\mathsf{sig}}, m)$.
10) Set $\mathsf{tx}_{\mathsf{Pour}} := (\mathsf{rt}, \mathsf{sn}^{\mathsf{old}}_1, \mathsf{sn}^{\mathsf{old}}_2, \mathsf{cm}^{\mathsf{new}}_1, \mathsf{cm}^{\mathsf{new}}_2, v_{\mathsf{pub}}, \mathsf{info}, *)$, where
    $* := (\mathsf{pk}_{\mathsf{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, \mathbf{C}_1, \mathbf{C}_2, \sigma)$.
11) Output $\mathbf{c}^{\mathsf{new}}_1, \mathbf{c}^{\mathsf{new}}_2$ and $\mathsf{tx}_{\mathsf{Pour}}$.

Receive
• INPUTS:
  – public parameters $\mathsf{pp}$
  – recipient address key pair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$
  – the current ledger $L$
• OUTPUTS: set of received coins
1) Parse $\mathsf{addr}_{\mathsf{pk}}$ as $(a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}})$.
2) Parse $\mathsf{addr}_{\mathsf{sk}}$ as $(a_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}})$.
3) For each Pour transaction $\mathsf{tx}_{\mathsf{Pour}}$ on the ledger:
   a) Parse $\mathsf{tx}_{\mathsf{Pour}}$ as $(\mathsf{rt}, \mathsf{sn}^{\mathsf{old}}_1, \mathsf{sn}^{\mathsf{old}}_2, \mathsf{cm}^{\mathsf{new}}_1, \mathsf{cm}^{\mathsf{new}}_2, v_{\mathsf{pub}}, \mathsf{info}, *)$,
      and $*$ as $(\mathsf{pk}_{\mathsf{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, \mathbf{C}_1, \mathbf{C}_2, \sigma)$.
   b) For each $i \in \{1, 2\}$:
      i) Compute $(v_i, \rho_i, r_i, s_i) := \mathcal{D}_{\mathsf{enc}}(\mathsf{sk}_{\mathsf{enc}}, \mathbf{C}_i)$.
      ii) If $\mathcal{D}_{\mathsf{enc}}$'s output is not $\perp$, verify that:
          • $\mathsf{cm}^{\mathsf{new}}_i$ equals $\mathsf{COMM}_{s_i}(v_i \| \mathsf{COMM}_{r_i}(a_{\mathsf{pk}} \| \rho_i))$;
          • $\mathsf{sn}_i := \mathsf{PRF}^{\mathrm{sn}}_{a_{\mathsf{sk}}}(\rho_i)$ does not appear on $L$.
      iii) If both checks succeed, output
           $\mathbf{c}_i := (\mathsf{addr}_{\mathsf{pk}}, v_i, \rho_i, r_i, s_i, \mathsf{cm}^{\mathsf{new}}_i)$.
```

Fig. 2: Construction of a DAP scheme using zk-SNARKs and other ingredients.

the maximum value of a coin, v_{max}, and the depth of the Merkle tree, d_{tree}.

D. Completeness and security

Our main theorem states that the above construction is indeed a DAP scheme.

Theorem IV.1. The tuple $\Pi = (\mathsf{Setup}, \mathsf{CreateAddress}, \mathsf{Mint}, \mathsf{Pour}, \mathsf{VerifyTransaction}, \mathsf{Receive})$, as defined in Section IV-C, is a complete (cf. Definition III.1) and secure (cf. Definition III.2) DAP scheme.

We provide a proof of Theorem IV.1 in the extended version of this paper [26]. We note that our construction can be modified to yield statistical (i.e., everlasting) anonymity; see the discussion in the extension section of the full version of this paper.

Remark (trusted setup)**.** Security of Π relies on a trusted party running Setup to generate the public parameters (once and for all). This trust is needed for the transaction non-malleability and balance properties but not for ledger indistinguishability. Thus, even if a powerful espionage agency were to corrupt the setup, anonymity will *still* be maintained. Moreover, if one wishes to mitigate the trust requirements of this step, one can conduct the computation of Setup using secure multiparty computation techniques; we leave this to future work.

V. Zerocash

We describe a concrete instantiation of a DAP scheme; this instantiation forms the basis of Zerocash. Later, in Section VI, we discuss how Zerocash can be integrated with existing ledger-based currencies.

A. Instantiation of building blocks

We instantiate the DAP scheme construction from Section IV (see Figure 2), aiming at a level of security of 128 bits. Doing so requires concrete choices, described next.

CRH, PRF, COMM from SHA256. Let $\mathcal{H}$ be the SHA256 compression function, which maps a 512-bit input to a 256-bit output. We mostly rely on $\mathcal{H}$, rather than the "full" hash, since this suffices for our fixed-size single-block inputs, and it simplifies the construction of $C_{\texttt{POUR}}$. We instantiate $\mathsf{CRH}, \mathsf{PRF}, \mathsf{COMM}$ via $\mathcal{H}$ (under suitable assumptions on $\mathcal{H}$).

First, we instantiate the collision-resistant hash function CRH as $\mathcal{H}(z)$ for $z \in \{0,1\}^{512}$; this function compresses "two-to-one", so it can be used to construct binary Merkle trees.[13]

Next, we instantiate the pseudorandom function $\mathsf{PRF}_x(z)$ as $\mathcal{H}(x\|z)$, with $x \in \{0,1\}^{256}$ as the seed, and $z \in \{0,1\}^{256}$ as the input.[14] Thus, the derived functions are $\mathsf{PRF}^{\mathrm{addr}}_x(z) := \mathcal{H}(x\|00\|z)$, $\mathsf{PRF}^{\mathrm{sn}}_x(z) := \mathcal{H}(x\|01\|z)$ and $\mathsf{PRF}^{\mathrm{pk}}_x(z) := \mathcal{H}(x\|10\|z)$, with $x \in \{0,1\}^{256}$ and $z \in \{0,1\}^{254}$.

As for the commitment scheme COMM, we only use it in the following pattern:

$$k := \mathsf{COMM}_r(a_{\mathsf{pk}}\|\rho)$$
$$\mathsf{cm} := \mathsf{COMM}_s(v\|k)$$

Due to our instantiation of PRF, a_{pk} is 256 bits. So we can set ρ also to 256 bits and r to $256+128=384$ bits; then we can compute $k := \mathsf{COMM}_r(a_{\mathsf{pk}}\|\rho)$ as $\mathcal{H}(r\|[\mathcal{H}(a_{\mathsf{pk}}\|\rho)]_{128})$. Above, $[\cdot]_{128}$ denotes that we are truncating the 256-bit string to 128 bits (say, by dropping least-significant bits, as in our implementation). Heuristically, for any string $x \in \{0,1\}^{128}$, the distribution induced by $\mathcal{H}(r\|x)$ is 2^{-128}-close to uniform, and this forms the basis of the statistically-hiding property. For computing cm, we set coin values to be 64-bit integers (so that, in particular, $v_{\mathsf{max}} = 2^{64}-1$ in our implementation), and then compute $\mathsf{cm} := \mathsf{COMM}_s(v\|k)$ as $\mathcal{H}(k\|0^{192}\|v)$. Noticeably, above we are *ignoring* the commitment randomness s. The reason is that we already know that k, being the output of a statistically-hiding commitment, can serve as randomness for the next commitment scheme.

Instantiating the NP statement POUR. The above choices imply a concrete instantiation of the NP statement $\texttt{POUR}$ (see Section IV-B). Specifically, in our implementation, $\texttt{POUR}$ checks that the following holds, for each $i \in \{1,2\}$:

- path_i is an authentication path for leaf $\mathsf{cm}^{\mathsf{old}}_i$ with respect to root rt, in a CRH-based Merkle tree;
- $a^{\mathsf{old}}_{\mathsf{pk},i} = \mathcal{H}(a^{\mathsf{old}}_{\mathsf{sk},i}\|0^{256})$;
- $\mathsf{sn}^{\mathsf{old}}_i = \mathcal{H}(a^{\mathsf{old}}_{\mathsf{sk},i}\|01\|[\rho^{\mathsf{old}}_i]_{254})$;
- $\mathsf{cm}^{\mathsf{old}}_i = \mathcal{H}(\mathcal{H}(r^{\mathsf{old}}_i\|[\mathcal{H}(a^{\mathsf{old}}_{\mathsf{pk},i}\|\rho^{\mathsf{old}}_i)]_{128})\|0^{192}\|v^{\mathsf{old}}_i)$;
- $\mathsf{cm}^{\mathsf{new}}_i = \mathcal{H}(\mathcal{H}(r^{\mathsf{new}}_i\|[\mathcal{H}(a^{\mathsf{new}}_{\mathsf{pk},i}\|\rho^{\mathsf{new}}_i)]_{128})\|0^{192}\|v^{\mathsf{new}}_i)$; and
- $h_i = \mathcal{H}(a^{\mathsf{old}}_{\mathsf{sk},i}\|10\|[h_{\mathsf{Sig}}]_{254})$.

Moreover, $\texttt{POUR}$ checks that $v^{\mathsf{new}}_1 + v^{\mathsf{new}}_2 + v_{\mathsf{pub}} = v^{\mathsf{old}}_1 + v^{\mathsf{old}}_2$, with $v^{\mathsf{old}}_1, v^{\mathsf{old}}_2 \geq 0$ and $v^{\mathsf{old}}_1 + v^{\mathsf{old}}_2 < 2^{64}$.

Finally, as mentioned, in order for $C_{\texttt{POUR}}$ to be well-defined, we need to fix a Merkle tree depth d_{tree}. In our implementation, we fix $d_{\mathsf{tree}} = 64$, and thus support up to 2^{64} coins.

Instantiating Sig. For the signature scheme Sig, we use ECDSA to retain consistency and compatibility with the existing `bitcoind` source code. However, standard ECDSA is malleable: both (r,s) and $(r,-s)$ verify as valid signatures. We use a non-malleable variant, where s is restricted to the "lower half" of field elements. While we are not aware of a formal $\mathsf{SUF\text{-}CMA}$ proof for this variant, its use is consistent with proposals to resolve Bitcoin transaction malleability [29].[15]

Instantiating Enc. For the encryption scheme Enc, we use the key-private Elliptic-Curve Integrated Encryption Scheme (ECIES) [30, 31]; it is one of the few standardized key-private encryption schemes with available implementations.

For further details about efficiently realizing these in the arithmetic circuit for $\texttt{POUR}$, see the full version of this paper.

VI. Integration with existing ledger-based currencies

Zerocash can be deployed atop any ledger (even one maintained by a central bank.) Here, we briefly detail integration with the Bitcoin protocol. Unless explicitly stated otherwise, in the following section when referring to *Bitcoin*, and its unit of account *bitcoin* (plural bitcoins), we mean the underlying protocol and software, not the currency system. (The discussion holds, with little or no modification, for many forks of Bitcoin, a.k.a. "altcoins", such as Litecoin.)

By introducing new transaction types and payment semantics, Zerocash breaks compatibility with the Bitcoin network. While Zerocash could be integrated into Bitcoin (the actual currency and its supporting software) via a "flag day" where a super-majority of Bitcoin miners simultaneously adopt the new software, we neither expect nor advise such integration in the near future and suggest using Zerocash in a separate altcoin.

[13]A single exception: we still compute h_{Sig} according to the full hash SHA256, rather than its compression function, because there is no need for this computation to be verified by $C_{\texttt{POUR}}$.

[14]This assumption is reminiscent of previous works analyzing the security of hash-based constructions (e.g., [28]). However in this work we assume that a portion of the compression function is the *seed* for the pseudorandom function, rather than using the chaining variable as in [28].

[15]In practice, one might replace this ECDSA variant with an EC-Schnorr signature satisfying $\mathsf{SUF\text{-}CMA}$ security with proper encoding of EC group elements; the performance would be similar.

Integrating Zerocash into Bitcoin consists of adding a new transaction type, Zerocash transactions, and modifying the protocol and software to invoke Zerocash's DAP interface to create and verify these transactions. Two approaches to doing so are described next, followed by a discussion of anonymizing the network layer.

A. Integration by replacing the base currency

One approach is to alter the underlying system so that all monetary transactions are done using Zerocash, i.e., by invoking the DAP interface and writing/reading the associated transactions in the distributed ledger.

As seen in Section III, this suffices to offer the core functionality of payments, minting, merging, splitting, etc., while assuring users that all transactions using this currency are anonymous. However, this has several drawbacks: all transactions incur the cost of generating a zk-SNARK proof; the scripting feature of Bitcoin is lost; and Bitcoin's ability to spend unconfirmed transactions is lost.

B. Integration by hybrid currency

A different approach is to extend Bitcoin with a parallel, anonymized currency of "zerocoins," existing alongside bitcoins, using the same ledger, and with the ability to convert freely between the two. The behavior and functionality of regular bitcoins is unaltered; in particular, they may support functionality such as scripting.

In this approach, the Bitcoin ledger consists of Bitcoin-style transactions, containing inputs and outputs [20]. Each input is either a pointer to an output of a previous transaction (as in plain Bitcoin), or a Zerocash pour transaction (which contributes its *public* value, v_{pub}, of bitcoins to this transaction). Outputs are either an amount and destination public address/script (as in plain Bitcoin), or a Zerocash mint transaction (which consumes the input bitcoins to produce zerocoins). The usual invariant over bitcoins is maintained and checked in plain view: the sum of bitcoin inputs (including pours' v_{pub}) must be at least the sum of bitcoin outputs (including mints' v), and any difference is offered as a transaction fee. However, the accounting for zerocoins consumed and produced is done separately and implicitly by the DAP scheme.

C. Additional anonymity considerations

Zerocash only anonymizes the transaction ledger. Network traffic used to announce transactions, retrieve blocks, and contact merchants will still leak identifying information (e.g., IP addresses). Thus users need some anonymity network to safely use Zerocash. The most obvious way to do this is via Tor [32]. Given that Zerocash transactions are not low latency themselves, Mixnets (e.g., Mixminion [33]) are also a viable way to add anonymity (and one that is not as vulnerable to traffic analysis as Tor). Using mixnets that provide email-like functionality has the added benefit of providing an out-of-band notification mechanism as a replacement to Receive.

Additionally, although in theory all users have a single view of the block chain, a powerful attacker could potentially fabricate an additional block *solely* for a targeted user. Spending any coins with respect to the updated Merkle tree in this "poison-pill" block will uniquely identify the targeted user. To mitigate such attacks, users should check with trusted peers their view of the block chain and, for sensitive transactions, only spend coins relative to blocks further back in the ledger (since creating the illusion for multiple blocks is far harder).

VII. Experiments

To measure the performance of Zerocash, we ran several experiments. First, we benchmarked the performance of the zk-SNARK for the NP statement POUR (Section VII-A) and of the six DAP scheme algorithms (Section VII-B). Second, we studied the impact of a higher block verification time via a simulation of a Bitcoin network (Section VII-C).

A. Performance of zk-SNARKs for pouring coins

Our zk-SNARK for the NP statement POUR is obtained by constructing an arithmetic circuit $C_{\texttt{POUR}}$ for verifying POUR, and then invoking the generic implementation of zk-SNARK for arithmetic circuit satisfiability of [16] (see Section II-C). The arithmetic circuit $C_{\texttt{POUR}}$ is built from scratch and hand-optimized to exploit nondeterministic verification and the large field characteristic.

Figure 3 reports performance characteristics of the resulting zk-SNARK for POUR. This includes three settings: single-thread performance on a laptop machine; and single-thread and multi-thread performance on a desktop machine. (The time measurements are the average of 10 runs, with standard deviation under 2.5%.)

B. Performance of Zerocash algorithms

In Figure 4 we report performance characteristics for each of the six DAP scheme algorithms in our implementation. Note that these numbers do not include the costs of maintaining the Merkle tree because doing so is not the responsibility of these algorithms. Moreover, for VerifyTransaction, we separately report the cost of verifying mint and pour transactions and, in the latter case, we exclude the cost of scanning L (as this cost depends on L). Finally, for the case of Receive, we report the cost to process a given pour transaction in L.

C. Large-scale network simulation

Because Bitcoin mining typically takes place on dedicated GPUs or ASICs, the CPU resources to execute the DAP scheme algorithms are often of minimal consequence to network performance. There is one potential exception to this rule: the VerifyTransaction algorithm must be run by all of the network nodes in the course of routine transaction validation. The time it takes to perform this verification can have significant impact on network performance.

In the Zerocash implementation (as in Bitcoin), every Zerocash transaction is verified at each hop as it is forwarded though the network and, potentially, again when blocks containing the transaction are verified. Verifying a block consists of checking the proof of work and validating the contained transactions.

		Intel Core i7-2620M @ 2.70GHz 12GB of RAM	Intel Core i7-4770 @ 3.40GHz 16GB of RAM	
		1 thread	1 thread	8 threads
KeyGen	Time	7 min 48 s	5 min 17 s	4 min 11 s
	Proving key	896 MiB		
	Verification key	749 B		
Prove	Time	2 min 55 s	2 min 2 s	1 min 3 s
	Proof	288 B		
Verify	Time	8.5 ms	5.4 ms	

Fig. 3: Performance of our zk-SNARK for the NP statement POUR. ($N = 10$, $\sigma \leq 2.5\%$)

Intel Core i7-4770 @ 3.40GHz with 16GB of RAM (1 thread)		
Setup	Time	5 min 17 s
	pp	896 MiB
CreateAddress	Time	326.0 ms
	$\mathsf{addr_{pk}}$	343 B
	$\mathsf{addr_{sk}}$	319 B
Mint	Time	23 µs
	Coin $\mathbf{c}$	463 B
	$\mathsf{tx_{Mint}}$	72 B
Pour	Time	2 min 2.01 s
	$\mathsf{tx_{Pour}}$	855 B[16]
VerifyTransaction	mint	8.3 µs
	pour (excludes L scan)	5.7 ms
Receive	Time (per pour tx)	1.6 ms

Fig. 4: Performance of Zerocash algorithms. ($N = 10$, $\sigma \leq 2.5\%$[17])

Thus Zerocash transactions may take longer to spread though the network and blocks containing Zerocash transactions may take longer to verify. While we are concerned with the first issue, the potential impact of the second issue is cause for greater concern. This is because Zerocash transactions cannot be spent until they make it onto the ledger.

Because blocks are also verified at each hop before they are forwarded through the network, delays in block verification slow down the propagation of new blocks through the network. This causes nodes to waste CPU-cycles mining on out-of-date blocks, reducing the computational power of the network and making it easier to mount a "51% attack" (dishonest majority of miners) on the distributed ledger.

It is a priori unclear whether this potential issue is a real concern. Bitcoin caches transaction verifications, so a transaction that was already verified when it propagated through the network need not be verified again when it is seen in a block. The unknown is what percentage of transactions in a block are actually in any given node's cache. We thus conduct a simulation of the Bitcoin network to investigate both the time it takes Zerocash transactions to make it onto the ledger and establish the effects of Zerocash transactions on block verification and propagation. We find that Zerocash transactions can be spent reasonably quickly and that the effects of increased block validation time are minimal.

Simulation design. Because Zerocash requires breaking changes to the Bitcoin protocol, we cannot test our protocol in the live Bitcoin network or even in the dedicated testnet. We must run our own private testnet. For efficiency and cost reasons, we would like to run as many Bitcoin nodes as possible on the least amount of hardware. This raises two issues. First, reducing the proof of work to practical levels while still preserving a realistic rate of new blocks is difficult (especially on virtualized hardware with variable performance). Second, the overhead of zk-SNARK verification prevents us from running many Bitcoin nodes on one virtualized server.

The frequency of new blocks can be modeled as a Poisson process with a mean of Λ_{block} seconds. To generate blocks stochastically, we modify `bitcoind` to fix its block difficulty at a trivial level and run a Poisson process, on the simulation control server, which trivially mines a block on a randomly selected node. This preserves the distribution of blocks, without the computational overhead of a real proof of work. Another Poisson process triggering mechanism, with a different mean Λ_{tx}, introduces new transactions at random network nodes.

To differentiate which transactions represent normal Bitcoin expenditures vs. which contain Zerocash pour transactions, simulated Zerocash transactions pay a unique amount of bitcoins (we set this value arbitrarily at 7 BTC). If a transaction's output matches this preset value, and it is not in verification cache, then our modified Bitcoin client inserts a 10 ms delay simulating the runtime of VerifyTransaction.[18] Otherwise transactions are processed as specified by the Bitcoin protocol. We vary the amount of simulated Zerocash traffic by varying the number of transactions with this particular output amount. This minimizes code changes and estimates only the generic impact of verification delays and not of any specific implementation choice.

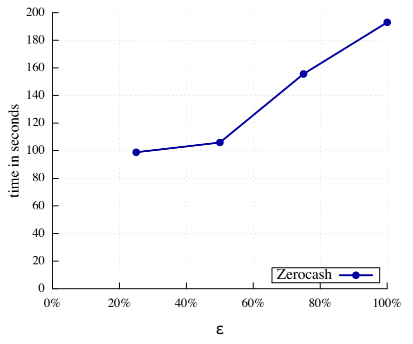
Methodology. Recent research [17] suggests that the Bitcoin network contains 16,000 distinct nodes though most are likely no longer participating: approximately 3,500 are reachable at any given time. Each node has an average of 32 open connections to randomly selected peers. As of November 2013, the peak observed transaction rate for Bitcoin is slightly under one transaction per second [34].

In our simulation, we use a 1000-node network in which each node has an average of 32 peers, transactions are generated with a mean of $\Lambda_{tx} = 1$ s, a duration of 1 hour, and a variable percentage ϵ of Zerocash traffic. To allow for faster experiments, instead of generating a block every 10 minutes as in Bitcoin, we create blocks at an average of every $\Lambda_{block} = 150$ s (as in Litecoin, a popular altcoin).

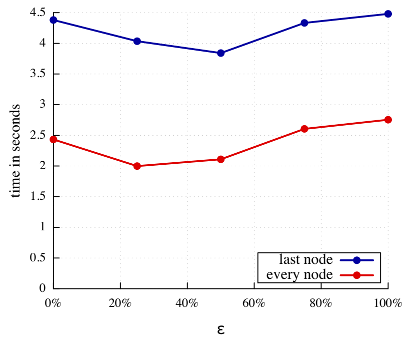
[16]346 B of this are due to the ciphertexts $\mathbf{C}_1, \mathbf{C}_2$. Future implementations may significantly reduce this overhead or discard these (cf. Section VI-C).

[17]We note that σ for both Mint and VerifyTransaction (mint) is higher than 2.5% due to the variability at such short timescales. Respectively, it is 3.3 µs and 1.9 µs.

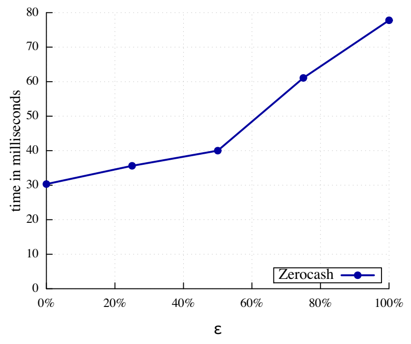
[18]Subsequent optimizations lowered the cost of VerifyTransaction below this, after our experiments.



(a) Transaction latency



(b) Block propagation time



(c) Block verification time

Fig. 5: The average values of the three metrics we study, as a function of ϵ, the percentage of transactions that are Zerocash transactions. Note that, in (a), latency is undefined when $\epsilon = 0$ and hence omitted.

We run our simulation for different traffic mixes, where ϵ indicates the percentage of Zerocash transactions and $\epsilon \in \{0\%, 25\%, 50\%, 75\%, 100\%\}$. Each simulation is run on 200 Amazon EC2 general-purpose `m1.medium` instances, in one region on a `10.10./16` private network. On each instance, we deploy 5 instances of `bitcoind`.

Results. Transactions are triggered by a blocking function call on the simulation control node that must connect to a random node and wait for it to complete sending a transaction. Because the Poisson process modeling transactions generates delays between such calls and not between the exact points when the node actuals sends the transactions, the actual transaction rate is skewed. In our experiments the real transaction rate shifts away from our target of one per second to an average of one every 1.4 seconds.

In Figure 5 we plot three metrics for $\epsilon \in \{0\%, 25\%, 50\%, 75\%, 100\%\}$. Each is the average defined over the data from the entire run of the simulation for a given ϵ (i.e., they include multiple transactions and blocks).[19] *Transaction latency* is the interval between a transaction's creation and its inclusion in a block. *Block propagation time* comes in two flavors: 1) the average time for a new block to reach a node computed over the times for all nodes, and 2) the same average computed over only the last node to see the block.

Block verification time is the average time, over all nodes, required to verify a block. If verification caching was not effective, we would expect to see a marked increase in both block verification time and propagation time. Since blocks occur on average every 150 s, and we expect approximately one transaction each second, we should see $150 \times 10\,\text{ms} = 1500\,\text{ms}$ of delay if all transactions were non-cached Zerocash transactions. Instead, we see worst case 80 ms and conclude caching is effective. This results in a negligible effect on block propagation (likely because network operations dominate).

The time needed for a transaction to be confirmed, and hence spendable, is roughly 190 s. For slower block generation rates (e.g., Bitcoin's block every 10 minutes) this should mean users must wait only one block before spending received transactions.

[19]Because our simulated Bitcoin nodes ran on shared EC2 instances, they were subject to variable external load, limiting the benchmark precision. Still, it clearly demonstrates that the mild additional delay does not cause catastrophic network effects.

VIII. Optimizations and extensions

See the extended version of this paper [26] for extensions on everlasting anonymity, batched Merkle tree updates, faster block propagation, and scaling to 2^{64} serial numbers.

IX. Concurrent work

Danezis et al. [19] suggest using zk-SNARKs to reduce proof size and verification time in Zerocoin. Our work differs from [19] in both supported functionality and scalability.

First, [19]'s protocol, like Zerocoin, only supports fixed-value coins, and is best viewed as a decentralized mix. Instead, we define, construct, and implement a full-fledged decentralized electronic currency, which provides anonymous payments of any amount.

Second, in [19], the complexity of the zk-SNARK generator, prover, and verifier all scale superlinearly in the number of coins, because their arithmetic circuit computes, *explicitly*, a product over all coins. In particular, the number of coins "mixed together" for anonymity cannot be large. Instead, in our construction, the respective complexities are polylogarithmic, polylogarithmic, and constant in the number of coins; our approach supports a practically-unbounded number of coins.

X. Conclusion

Decentralized currencies should ensure a user's privacy from his peers when conducting legitimate financial transactions. Zerocash provides such privacy protection, by hiding user identities, transaction amounts, and account balances from public view. This, however, may be criticized for hampering accountability, regulation, and oversight. Yet, Zerocash need not be limited to enforcing the basic monetary invariants of a currency system. The underlying zk-SNARK cryptographic proof machinery is flexible enough to support a wide range of policies. It can, for example, let a user prove that he paid his due taxes on all transactions *without* revealing those transactions, their amounts, or even the amount of taxes paid. As long as the policy can be specified by efficient nondeterministic computation using NP statements, it can (in principle) be enforced using zk-SNARKs, and added to Zerocash. This can enable privacy-preserving verification and enforcement of a wide range of compliance and regulatory policies that would otherwise be invasive to check directly or might be bypassed by corrupt authorities. This raises research, policy, and engineering questions over what policies are desirable and practically realizable.

Another research question is what new functionality can be realized by augmenting the capabilities already present in Bitcoin's scripting language with zk-SNARKs that allow fast verification of expressive statements.

Acknowledgments

We thank Amazon for their assistance and kind donation of EC2 resources, and Gregory Maxwell for his advice regarding the Bitcoin codebase. We thank Iddo Ben-Tov and the SCIPR Lab members — Daniel Genkin, Lior Greenblat, Shaul Kfir, Gil Timnat and Michael Riabzev — for inspiring discussions.

This work was supported by: Amazon.com through an AWS in Education research grant; the Broadcom Foundation and Tel Aviv University Authentication Initiative; the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370; the Check Point Institute for Information Security; the U.S. Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211; the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 240258; the Israeli Centers of Research Excellence I-CORE program (center 4/11); the Israeli Ministry of Science and Technology; the Office of Naval Research under contract N00014-11-1-0470; the Simons Foundation, with a Simons Award for Graduate Students in Theoretical Computer Science; and the Skolkovo Foundation under grant agreement 6926059.

The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] D. Chaum, "Blind signatures for untraceable payments," in *CRYPTO '82*.

[2] J. Camenisch, S. Hohenberger, and A. Lysyanskaya, "Compact e-cash," in *EUROCRYPT '05*.

[3] T. Sander and A. Ta-Shma, "Auditable, anonymous electronic cash," in *CRYPTO '99*.

[4] F. Reid and H. Martin, "An analysis of anonymity in the Bitcoin system," in *SocialCom/PASSAT '11*.

[5] S. Barber, X. Boyen, E. Shi, and E. Uzun, "Bitter to better - how to make Bitcoin a better currency," in *FC '12*.

[6] D. Ron and A. Shamir, "Quantitative analysis of the full Bitcoin transaction graph," ePrint 2012/584, 2012.

[7] G. Maxwell, "CoinJoin: Bitcoin privacy for the real world," August 2013, bitcoin Forum. [Online]. Available: https://bitcointalk.org/index.php?topic=279249.0

[8] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *SP '13*.

[9] J. Groth, "Short pairing-based non-interactive zero-knowledge arguments," in *ASIACRYPT '10*.

[10] H. Lipmaa, "Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments," in *TCC '12*.

[11] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth, "Succinct non-interactive arguments via linear interactive proofs," in *TCC '13*.

[12] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct NIZKs without PCPs," in *EUROCRYPT '13*.

[13] B. Parno, C. Gentry, J. Howell, and M. Raykova, "Pinocchio: nearly practical verifiable computation," in *Oakland '13*.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: verifying program executions succinctly and in zero knowledge," in *CRYPTO '13*.

[15] H. Lipmaa, "Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes," in *ASIACRYPT '13*.

[16] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive arguments for a von Neumann architecture," ePrint 2013/879.

[17] C. Decker and R. Wattenhofer, "Information propagation in the Bitcoin network," in *P2P '13*.

[18] E. Ben-Sasson, "Universal and affordable computational integrity," May 2013, bitcoin 2013: The Future of Payments. [Online]. Available: http://www.youtube.com/watch?v=YRcPReUpkcU

[19] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno, "Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system," in *PETShop '13*. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/G.Danezis/papers/DanezisFournetKohlweissParno13.pdf

[20] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[21] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval, "Key-privacy in public-key encryption," in *ASIACRYPT '01*.

[22] D. Boneh and X. Boyen, "Secure identity based encryption without random oracles," in *CRYPTO '04*.

[23] R. Gennaro, "Multi-trapdoor commitments and their applications to proofs of knowledge secure under concurrent man-in-the-middle attacks," in *CRYPTO '04*.

[24] C. Gentry and D. Wichs, "Separating succinct non-interactive arguments from all falsifiable assumptions," in *STOC '11*.

[25] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in *ITCS '12*.

[26] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from Bitcoin (extended version)," Cryptology ePrint Archive, 2014.

[27] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007.

[28] M. Bellare, "New proofs for NMAC and HMAC: security without collision-resistance," in *CRYPTO '06*.

[29] P. Wuille, "Proposed BIP for dealing with malleability," Available at https://gist.github.com/sipa/8907691, 2014.

[30] V. Shoup, "A proposal for an ISO standard for public key encryption (version 2.1)," *IACR E-Print Archive*, 2001.

[31] Certicom Research, "SEC 1: Elliptic curve cryptography," 2000. [Online]. Available: http://www.secg.org/collateral/sec1_final.pdf

[32] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: the second-generation onion router," in *Security '04*.

[33] G. Danezis, R. Dingledine, and N. Mathewson, "Mixminion: design of a type III anonymous remailer protocol," in *SP '03*.

[34] T. B. Lee, "Bitcoin needs to scale by a factor of 1000 to compete with Visa. here's how to do it." The Washington Post (http://www.washingtonpost.com), November 2013.