Accountable Privacy for Decentralized Anonymous Payments

Christina Garman* Matthew Green* Ian Miers*

June 26, 2016

Abstract

Decentralized ledger-based currencies such as Bitcoin provide a means to construct payment systems without requiring a trusted bank. Removing this trust assumption comes at the significant cost of transaction privacy. A number of academic works have sought to improve the privacy offered by ledger-based currencies using anonymous electronic cash (e-cash) techniques. Unfortunately, this strong degree of privacy creates new regulatory concerns, since the new private transactions cannot be subject to the same controls used to prevent individuals from conducting illegal transactions such as money laundering. We propose an initial approach to addressing this issue by adding *privacy preserving* policy-enforcement mechanisms that guarantee regulatory compliance, allow selective user tracing, and admit tracing of tainted coins (e.g., ransom payments). To accomplish this new functionality we also provide improved definitions for Zerocash and, of independent interest, an efficient construction for simulation sound zk-SNARKs.

*Johns Hopkins University, {cgarman,mgreen,imiers}@cs.jhu.edu

Contents

1 Introduction **3**

2 Decentralized Anonymous Payments **4**
2.1 Existing definitions and limitations . . . 5
2.2 Simulation-based definition for DAP schemes . . . 5

3 Modifying and extending Zerocash **7**

4 Simulation sound Succinct non-interactive arguments of knowledge **7**

5 Policies **8**
5.1 Building blocks . . . 8
5.2 Regulatory closure . . . 10
5.3 Spending limits . . . 11
5.4 Tax . . . 11
5.5 Identity escrow . . . 13

6 Coin tracing **13**
6.1 Construction . . . 13
6.2 Security . . . 15

7 Accountable user tracing **16**

8 Related work **17**
8.1 Anti-money laundering for centralized e-cash . . . 17
8.2 Hawk . . . 17

9 Acknowledgements **18**

A Simulation sound Succinct non-interactive arguments of knowledge **20**
A.1 Notation . . . 20
A.2 Definitions . . . 20
A.3 Construction of simulation-sound SNARKs . . . 22

B Zerocash Modifications **25**

C Proof Sketch for Theorem 2.2 **26**

1 Introduction

The success of decentralized currencies like Bitcoin has led to renewed interest in anonymous electronic cash both in academia [3, 10, 23] and in practice (including Coinjoin, CryptoNote, and DarkWallet). It has also highlighted new problems related to trust, privacy and regulatory compliance. In modern electronic payment systems, users must trust that their bank is not tampering with the system (e.g., by "forging" extra currency), that no party is abusing the privacy of users' transactions, and simultaneously, that other users are not using the system to engage in money laundering or extortion. Unfortunately, these goals seem fundamentally at odds.

Decentralized payment systems such as Bitcoin address the first issue by replacing the central bank with a distributed ledger and consensus system. Unfortunately, this benefit comes at a significant cost to privacy [2, 6, 26], since any user can now view the transaction graph and potentially trace payments made by another user. Proposals such as Zerocoin and Zerocash [3, 23] attempt to resolve the privacy problem by employing sophisticated zero knowledge proofs of transaction correctness. While such constructions address privacy concerns, they do not offer any additional protections against money laundering and other activity. From an investigative standpoint, Zerocash is no different than cash.

The problem of preventing money laundering in decentralized currencies is not theoretical. In May 2015 the decentralized payment network Ripple was ordered to pay a $700,000 fine by the U.S. Financial Crimes Enforcement Network (FINCEN) due to inadequate monitoring of transactions on their network. This raises questions for the deployment of privacy-preserving payment networks, where individual network operators and currency exchanges may be held criminally liable for facilitating laundering. In general, the difficulty of preventing abuse may prove a barrier to the deployment of private currency systems. In particular, it seems likely that the application of existing reporting requirements will force exchanges/online wallets, even when dealing with anonymous currencies, to hold vast amounts of information about their users, their transactions, and the amounts. If, as in Bitcoin, most consumers use online wallets, then they gain little privacy.

In this paper we aim for a middle ground. Specifically, we design new Decentralized Anonymous Payment (DAP) systems [3] that are capable of enforcing compliance with specific transaction policies, while protecting the privacy of network participants. Our approach builds on the techniques of Ben-Sasson et al.'s Zerocash system [3], using efficient zero knowledge Arguments of Knowledge (zk-SNARKs) [4, 5, 7, 12, 16, 21, 22, 24] to both prove the validity of transactions and to enforce transaction *policies*. These policies allow network participants to verifiably prove compliance with global transaction policies, such as tax payments, deposit limits, and verifiable coin tracing, all while protecting the privacy of each transaction. Interestingly, a side effect of our mechanisms is that even centralized banks or exchanges participating in our scheme are prevented from evading anti-money laundering controls, as these controls are enforced by cryptographically sound proofs.

Improving security definitions for DAP schemes. A necessary starting point for our exploration is to find a formal security definition for DAP schemes that can be readily extended to incorporate our policies. Previous security definitions for DAP schemes involve a series of distinct games, each of which addresses one aspect of the security requirements for the DAP scheme. Not only are these games complex, they fail to simultaneously ensure both *correctness* and privacy of the DAP scheme. In particular, we note that the Zerocash scheme [3] admits a practical attack due to a limitation of the correctness definition. In this paper we explore a more promising route: we define an ideal-world functionality to describe the security properties of a DAP scheme, which captures both the correctness and privacy of the scheme in a single intuitive definition. Moreover, as we show in later sections, this simulation-based definition can easily be augmented to incorporate policies.

Augmenting existing constructions with simulation-sound zkSNARKs. Given an improved simulation-based definition for DAP schemes, we encounter a technical problem that prevents us from proving security of the scheme: we require zero-knowledge proofs which are sound and extractable in the presence of simulated proofs, i.e., we need simulation sound proofs. However, the zk-SNARK used in Zerocash provides no such guarantees. To address this we build the first instantiation of simulation sound zk-SNARKs.

While this seems like a small step and uses folklore techniques, it is necessary and must be done carefully to ensure security and efficiency. Our approach preserves succinctness and through clever choice of primitives, has almost no effect on Zerocash performance. See Section 4.

Anti-money laundering policies for DAP schemes. Given a stronger formal foundation, we can now explore and reason about a wide range of anti-money laundering policies that still preserve user privacy. In fact we are able to go substantially beyond this and provide policies that simultaneously allow the authorities to both trace coins as they go from individual to individual and retrieve all of a particular user's transactions and provide an accountable record of when and why those powers were used. We provide examples of several concrete compliance policies that leverage our new techniques. These include policies to enforce regulatory closure, enforced tax payments, spending limits, coin tracing, and user tracing. We also provide the first schemes in the decentralized e-cash setting that allow for coin and user tracing while maintaining an accountable record to limit abuse of those powers.

Paper outline. The remainder of this paper proceeds as follows. In Section 2 we provide definitions for secure DAP schemes and propose our modified security definition. In Section 3 we introduce the Zerocash construction and propose modifications to achieve our new definitions. In Section 5 we show how to modify a DAP scheme to enforce various policies. In Sections 6 and 7 we detail specific policies for coin tracing and accountable user tracing.

2 Decentralized Anonymous Payments

Decentralized Anonymous Payment (DAP) systems were introduced by Ben-Sasson et al. [3]. A complete formal definition can be found in [3]. A DAP system consists of an append-only ledger and a tuple of polynomial time algorithms ($\mathsf{Setup}, \mathsf{CreateAddress}, \mathsf{Mint}, \mathsf{Pour}, \mathsf{VerifyTransaction}, \mathsf{Receive}$) with the following (informal) semantics:

$\mathsf{Setup}(1^\lambda) \to params$ generates global parameters for the system.
$\mathsf{CreateAddress}(params) \to (\mathsf{addr_{pk}}, \mathsf{addr_{sk}})$ outputs an individual address (key pair) for each user.
$\mathsf{Mint}(params, \mathsf{addr_{pk}}, v) \to (\mathsf{cm}, \pi, \mathsf{trap})$ creates a new transaction embedding a coin of value v, a proof π, and outputs a trapdoor trap for spending the coin.
$\mathsf{Pour}(params, \mathsf{cm}_1, \mathsf{cm}_2, \mathsf{trap}_1, \mathsf{trap}_2, \mathsf{addr}_{\mathsf{pk},1}, \mathsf{addr}_{\mathsf{pk},2}, v_1, v_2, v_{\mathsf{pub}}) \to (\mathsf{cm}_1, \mathsf{cm}_2, \pi)$ takes as input two coin commitments and the corresponding trapdoors, as well as a public value output and two new coin values and destination addresses, and outputs a new transaction embedding the new coins.
$\mathsf{VerifyTransaction}$ validates any of the above transactions.
$\mathsf{Receive}$ scans the transactions on the ledger, using the user's corresponding secret key to identify transactions addressed to the user.

Informally, a DAP scheme uses these transactions along with a trusted append-only ledger to transact funds between different users. By issuing "coins" in exchange for work, or for non-anonymous currencies, users can split, merge and combine coins of arbitrary value. Additionally,

senders can transmit funds to another user without revealing the contents of the transfer to any third party.

2.1 Existing definitions and limitations

The DAP security definitions provided by Ben-Sasson et al. [3] are game based. As a result, these definitions are complex, spanning three games: one for anonymity, one for non-malleability (needed to ensure correct integration into Bitcoin), and one for balance. While non-malleability is simple, both the game for anonymity ("ledger indistinguishability") and the one preventing coin forgery ("balance") are quite complex. More importantly, these definitions are also incomplete: they do not fully enforce correctness. Unfortunately, malicious users can exploit these weaknesses to break other properties in cryptographic systems.

Specifically, in Zerocash, each Pour transaction contains one or more serial numbers related to the coins spent. These are designed to prevent double spending, and are formed as a combination of the coin recipient's private key and randomness chosen by the sender. As pointed out by Wilcox [27], since the private key is fixed, if the sender uses the same randomness in two different transactions, then the serial numbers will be identical. As a result, the recipient will be unable to spend both transactions, since the duplicate serial numbers will appear to indicate a double spend. This would allow an attacker to send two payments of e.g. \$500 and \$1,000, and then ask for a refund on the "accidental" \$1,000 dollar payment. While this attack costs the attacker \$500, since both payments go through and she only gets one back, it also costs the victim \$500, since the "accidental" payment can never be spent. This attack is easily fixed by having the recipient check for duplicate serial number randomness over all previous transactions they received.

Nonetheless, this attack shows that the "balance" definition does not fully capture what we assume about a currency system. Clearly, some part of the security definitions in a DAP scheme should prevent this, yet the definitions for Zerocash do not.

2.2 Simulation-based definition for DAP schemes

An alternative to using game-based definitions like those of [3] is to use a simulation-based definition. In this approach, we define our system in terms of an ideal functionality implemented by a trusted party T_P that plays the role that our cryptographic constructions play in the real system [14]. In the ideal-world experiment, a collection of parties interact with the trusted party according to a specific interface. In the real-world experiment, the parties interact with each other using cryptography. We now define the experiments:

Ideal-world execution $\texttt{IDEAL}_{T_P,\mathcal{S},\Sigma}$. The ideal world attacker, $\mathcal{S}$, selects some subset of the $P_1, \ldots, P_n$ parties to corrupt and informs the T_P. The attacker controlled parties behave arbitrarily, the honest ones follow a set of strategies Σ. All parties then interact via messages passed to and from the T_P implementing an ideal functionality outlined in Figure 1.

Real world functionality $\texttt{REAL}_{DAP,\mathcal{A},\Sigma}$. The real world attacker $\mathcal{A}$ controls a subset of the parties $P_1, \ldots, P_n$ interacting with the real DAP scheme. The honest parties execute the commands output by the strategy Σ using the DAP scheme while the attacker controlled parties can behave arbitrarily. We assume that all parties can interact with a trusted append only ledger.

For generality we present our ideal functionality definitions, as well as subsequent transaction policies, as operations on n coins in and n coins out. As we do not wish to re-present all of Zerocash, however, for brevity we deal concretely with the existing Zerocash construction which was defined for the 2-in-2-out case.

We initialize two tables, one for addresses, and one for coins.

- $\mathsf{RegisterID}(\mathsf{addr_{pk}})$. When called by party U, registers a new address $\mathsf{addr_{pk}}$
  - The input is an address $\mathsf{addr_{pk}}$.
  - If $\mathsf{addr_{pk}}$ is not already in the address table, the trusted party T_P stores $(U, \mathsf{addr_{pk}})$ in the address table. Else, reject.
- $\mathsf{c} \leftarrow \mathsf{Mint}(v, \mathsf{addr_{pk}})$. Creates a coin for party U
  - The input is a value v and a destination address $\mathsf{addr_{pk}}$.
  - The trusted party T_P checks to make sure that $\mathsf{addr_{pk}}$ has been registered to user U. If not, reject.
  - Else, T_P generates a unique random transaction id c, stores $(\mathsf{c}, v, \mathsf{addr_{pk}})$ in the coin table, and notifies all parties of $Mint(v)$.
  - T_P returns c to the user.
- $[\mathsf{c}_1^{new}, \ldots, \mathsf{c}_n^{new}] \leftarrow \mathsf{Pour}([\mathsf{c}_1, \ldots, \mathsf{c}_m], [(\mathsf{addr_{pk,1}}, v_1), \ldots, (\mathsf{addr_{pk,n}}, v_n)], v_{\mathsf{pub}})$. Generates n new coins. Party U pours coins to new addresses
  - The input is a list of previous transaction id's c_i, tuples of (destination address $\mathsf{addr_{pk,i}}$, transaction value v_i), and a public value v_{pub}.
  - For each input c_i, T_P retrieves each transaction, its associated value v_i^{old}, and address from the coin table and checks the address belongs to U in the address table. If not, reject.
  - T_P checks $\sum_{i=1}^{m} v_i^{old} = \sum_{i=1}^{n} v_i + v_{\mathsf{pub}}$. If not, reject.
  - The trusted party then creates n new random transaction id's c_i^{new} each with value v_i and for each stores $(\mathsf{c}_i^{new}, v_i, \mathsf{addr_{pk,i}})$ in the coin table.
  - T_P deletes the input transaction(s) from the coin table.
  - For each $\mathsf{addr_{pk,i}}$ that is not in the address table, T_P notifies the corrupted parties of $Pour(\mathsf{c}_i^{new}, v_i, \mathsf{addr_{pk,i}})$.
  - For all $\mathsf{addr_{pk,i}}$ in address table, T_P notifies the party associated with $\mathsf{addr_{pk,i}}$ of $Pour(\mathsf{c}_i^{new}, v_i, \mathsf{addr_{pk,i}})$.
  - The trusted party T_P notifies all parties of $Pour(v_{\mathsf{pub}})$.
  - T_P returns $[\mathsf{c}_1^{new}, \ldots, \mathsf{c}_n^{new}]$ to the user.

Figure 1: Ideal Functionality. Security of a basic DAP scheme.

Definition 2.1 *We say that DAP securely emulates the ideal functionality provided by T_P if for all probabilistic polynomial-time real-world adversaries $\mathcal{A}$ and all honest party strategies Σ, there exists a simulator $\mathcal{S}$ such that for any ppt distinguishers d:*

$$P[d(\texttt{IDEAL}_{T_P,\mathcal{S},\Sigma}(\lambda)) = 1] - P[d(\texttt{REAL}_{DAP,\mathcal{A},\Sigma})(\lambda) = 1] \leq negl(\lambda)$$

Theorem 2.2 *The basic DAP scheme described in [3] and this paper satisfies Definition 2.1 given the existence of simulation-sound zkSNARKs, collision resistant hash functions, PRFs, statistically hiding and computationally binding commitments, one-time strongly-unforgeable digital signatures, and key-private public key encryption.*

See Appendix C for a proof of Theorem 2.2.

3 Modifying and extending Zerocash

We use the same notation for addresses, Merkle tree, ledger, and other components as Zerocash [3]. However, to meet the new definitions we must lightly modify the existing construction. In particular, we need a mechanism to allow us to extract who a Mint transaction was authored by, and we need to extend the definitions to allow for policies to be applied that validate transactions.

We modify the Zerocash construction in two ways. First, by adding a proof to the Mint algorithm that $\mathsf{cm} := \mathsf{COMM}_s(v \| \mathsf{COMM}_r(a_{\mathsf{pk}} \| \rho))$ and a verification of this proof to $\mathsf{VerifyTransaction}$. This is to support our new definition. Second, we modify $\mathsf{Receive}$ to fix the attack described in Section 2 and check if the serial number randomness is distinct from any previously received coins, not just the ones already spent on the ledger. We refer the curious reader to Appendix B and [3] for the complete details.

4 Simulation sound Succinct non-interactive arguments of knowledge

One technical detail of the proof for Theorem 2.2 is that the proof requires a simulation-sound zk-SNARK. However, the SNARK constructions used by Zerocash [3] are based on a construction [25] that does not provide this property. Thus to complete the proof, we require proof systems that are sound and extractable even when there are simulated proofs. A formal treatment of this can be found in Appendix A.

Informally, we accomplish this in two ways. First, we provide a mechanism for producing a simulated proof without directly forging the zero-knowledge proof itself. To do this we apply a standard technique due to Groth [15] in which the simulation sound proof of a statement is modified to be "statement OR signature under key in the CRS," applied to the zk-SNARK. For efficiency, we do this without using a signature scheme in the proof statement, instead using symmetric primitives and a statement of the form "statement OR MAC under key committed to in CRS".

Second, we must ensure the proof is still a proof of knowledge, i.e., we can still extract a witness even from a forged proof. This is a seemingly innocuous detail, but SNARKs are unlike most proof systems in that they depend on the ability to inspect the state of the attacker. When the attacker has access to a simulator, we must ensure that he can't hide critical information from the extractor by using the simulator.

Informally zk-SNARKs assume there exists a knowledge of exponent extractor which, given white-box access to an attacker $\mathcal{A}$ who forges a proof and their random coins, there exists some

extractor $\mathcal{X}_{\mathcal{A}}$ who can recover a series of exponents from which a witness is computed. For us, we must ensure the knowledge of exponent extractor still functions even when $\mathcal{A}$ has oracle access to a proof simulator. We ensure this through a hybrid argument where in the end the extractor $\mathcal{X}_{\mathcal{A}}$ is given the CRS signature/MAC key needed to simulate proofs and emulates the simulator to $\mathcal{A}$.

The resulting construction Π_{SS} uses an existing SNARK, a **PRF**, a MAC, and a strong one time signature scheme to create an efficient simulation sound zk-SNARK consisting of a proof π and a one time signature.

Theorem 4.1 *If* Π *is a witness-indistinguishable SNARK,* **PRF** *is a secure pseudo-random permutation, MAC is an EU-CMA message authentication code, and sig is a strongly-unforgeable one-time signature scheme, then* Π_{SS} *is computationally simulation knowledge sound and zero-knowledge proof system.*

A proof of Theorem 4.1 can be found in Appendix A.

5 Policies

Given an extensible security model for Zerocash-like systems we can now consider more complex policies governing the transfer of funds than those originally embedded in Zerocash. Zerocash only enforced one restriction on funds transfer: the sum of the values in the list of output commitments must not exceed the sum of the input ones. Although that is the most basic policy governing any monetary system, far more complex ones govern our modern banking system. These include policies limiting money laundering, enforcing tax payments, and facilitating international funds transfers, among other examples. In this section, we explore several analogous policies for decentralized e-cash that still mostly preserve user privacy.

Formally, we define a policy as an algorithm that is executed each time a coin is spent ("poured"). The algorithm is parameterized by some constants (e.g., all transaction tax policies are the same except for the tax rate parameter) and takes public inputs, private inputs, and returns true or false. Policies are realized by a zero-knowledge proof, so we are essentially giving a procedural description of the standard efficiently decidable binary relation used in zero-knowledge proofs. We note that no fresh computation takes place in a policy, only validation of precomputed data.

5.1 Building blocks

To construct our policies, we use use a few basic techniques detailed here.

5.1.1 Adding information to coins

In order to accomplish various policies, we need to store more than a numeric value inside a coin (indeed, as shown later, we will need to use coins for other things).

We do this by replacing the 192 bit zero padding in the commitment with a special *tag* used to mark what kind of counter it is. To do this we can alter the creation of a coin from

$$\mathsf{cm} := \mathsf{COMM}_s(v \| k) \quad \text{as} \quad \mathcal{H}(k \| 0^{192} \| v)$$

to

$$\mathsf{cm} := \mathsf{COMM}_s(v \| type \| data \| k) \quad \text{as} \quad \mathcal{H}(k \| type \| data \| v).$$

Policies will use the type information to ensure that an adversary cannot cause them to operate over the wrong data. Careful care must be taken so that one policy does not output data of a type

used by a different policy for a different purpose. This also means that each new type requires a special Mint.

5.1.2 Counters

A recurring problem we will encounter in these policies is the problem of how to count, e.g., how do you count how much money a user has sent, how much they need to pay taxes on, etc? Many policies depend on counting funds or transactions and as such, we treat it as a basic building block.

The approach we take is to replace the content of a coin with a counter. Upon each transaction the *policy* governing it requires the user to input their current counter state and output the new counter with the appropriate increment. Counters are a new type of "coin" and we define MintCTR analogously to Mint but have it reveal the *tag* in addition to v which is now called ctr. In order to maintain anonymity, these counters need to be input via the same mechanism that coin commitments are (i.e., by proving they are in a Merkle tree). In the policies below, we denote them as special inputs solely for clarity.

The particular mechanism for restricted mint depends on deployment. For certain scenarios, it may be enough to ensure that there is only one counter per address. For most scenarios where counters are tightly coupled with real world identity, we assume such mints must tied to some trusted party, probably via signature. There is thus an implicit policy on which MintCTRs will be accepted.

We also define a utility function for verifying a counter in Algorithm 1.

```
input  : CTROld, CTRNew, delta, tag
output : True if counter is valid

if not tag = CTROld.tag = CTRNew.tag then
  | return False;
end
return CTRNew = CTROld.value + delta;
```

Algorithm 1: VerifyCounter

5.1.3 Identity

An important aspect of any regulatory system is user *identity*. Legal concepts of identity are decidedly centralized. Since most policies are a result of legal and regulatory requirements, most of them require some notion of legal identity. In the case of counters bound to an identity, a trusted third party can issue the counters. More practically, given any trusted party who issues identity certificates, we can require a signature under its certificate chain.

Another possibility is to simply bind counters or other objects to Bitcoin/Zerocash addresses and enforce real world identity enforcement at exchanges and online wallets. In this case then, a user would be associated with a set of addresses and would later have to tie these addresses to a real-world identity. This is analogous to the current approach used in Bitcoin, where exchanges and online wallets track user identities. We believe this is the most promising avenue for real deployment.

5.1.4 Signatures and encryption

The final building block some of our policies use are public key encryption and signature schemes. Because Zerocash uses zk-SNARKs which are proofs over arithmetic circuits, the naive approach to

instantiating these schemes would be extremely costly: first build a circuit for modular exponentiation operations in some group and then build the encryption scheme on top of that. However, it is possible instead to directly calculate the signature verification operations in an extension field [10] of the arithmetic circuit, vastly increasing efficiency.

5.2 Regulatory closure

This policy allows multiple regulatory environments to coexist in the same system but the units of currency under each scheme do not mix. In practice, this policy ensures that transactions do not bridge between regulatory schemes, except through an outside mechanism such as an exchange (where legal requirements can be enforced). To implement this policy, coins, instead of just including a numeric value v, now include a regulatory type. These are created on Mint. The policy ensures that either:

1. the regulatory type of all input coins and all output coins is the same
2. or the type of all output coins is marked as $\perp$.

The point of this policy is to ensure continuity of some other regulatory scheme which may or may not be enforced by zero-knowledge. For example, such a policy could ensure that coins only came from exchanges or hosted wallets in a particular jurisdiction. In the algorithm described in Algorithm 2, we allow coins to have an arbitrary regulator marking if the output coins are signed by a valid authority. This could be a bank, an exchange, or the government itself. This allows the transfer of coins from one scheme to the other.

```
PrivateInput : CoinsIn, CoinsOut, CA, sig
Constants    : inRegType,CA
output       : True if transaction is valid

if sig ≠ ⊥ then                    // if tx signed by authorized party, skip policy
  | return VerifySig(CA, CoinsOut, sig);
end
outRegType ← inRegType;
for e ∈ CoinsIn do
  | if e.regType ≠ inRegType then
  |   | outRegType ← ⊥; // if any regType is wrong, outRegType is ⊥
  | end
end
for e ∈ CoinsOut do                // check each new coin has the correct regtype
  | if e.regType ≠ outRegType then
  |   | return False;
  | end
end
return True;
```

Algorithm 2: Regulatory closure

5.3 Spending limits

In the US one common form of anti-money laundering control is a Suspicious Activity Report. This report is generated by a bank when any single transaction over $10,000 USD is reported. This can be easily implemented.

```
PrivateInput : CoinsIn,CoinsOut
PublicInput  : sig
Constants    : limit
output       : True if transaction is valid

sent ← 0;
if sig ≠ ⊥ then                // if tx signed by authorized party, skip policy
  | return VerifySig(CA, CoinsOut, sig);
end
for e ∈ CoinsOut do
  | // if we are paying to an input address, it is ''change'' back to the
  |    user and not counted against the limit
  | if e.addr ∉ {c.addr|c ∈ CoinsIn} then
  |   | sent ← sent + e.value;
  | end
end
return sent ≤ limit;
```

Algorithm 3: Transaction limit

The mechanism is simple: no transaction over the limit is valid unless signed by an authority. This allows the authority to force reporting for transactions over the limit. Note, we explicitly require the signature to be public. The reason is to prevent fraud on the part of the authority. Were the signature private, a bank enforcing this could undetectably allow anyone to circumvent the reporting mechanism by issuing more signatures. On the other hand, when signatures are public, we can require that the authority log (privately) all transaction details along with the transaction itself (and perhaps a proof the details are valid). See Algorithm 3. Since the signature is used once and only known to the authority, revealing it does not impact privacy.

A more sophisticated policy would place spending limit over all transactions instead of a limit on a single transaction. For this we require counters. See Algorithm 4.

Modified ideal functionality and proof sketch. We now detail, informally, the modifications to the ideal functionality necessary to capture counter based spending limits and the changes necessary to the proof. The ideal functionality changes are small: we add a per user counter maintained by the ideal functionality and modify *pour* to increment the counter as appropriate and reject the transaction if the counter is over a given limit. The resulting proof change is similarly tiny: the simulator aborts if the attacker submits a real world transaction violating their counting limit or if they try and issue a new counter. The probability of the former event is negligible if the proof system is simulation sound and the second event is not allowed in either the real or ideal protocol since the attacker cannot make new counters.

5.4 Tax

Our policies also allow us to enforce taxes, such as income taxes, sales taxes, and Value Added Tax (VAT). We accomplish this by requiring a percentage payment to a tax account with each Zerocash

```
PrivateInput : CoinsIn, CoinsOut, CTRcm^old, CTRcm^new
PublicInput  : epoch,sig
Constants    : limit
output       : True if transaction is valid

spent ← 0;
if sig ≠ ⊥ then                // if tx signed by authorized party, skip policy
  | return VerifySig(CA, CoinsOut, sig);
end
if CTRcm^new.value ≥ limit then                // if counter over limit, reject
  | return False;
end
for e ∈ CoinsOut do
  | // if we are paying to an input address, it is ''change'' back to the
  |    user and not counted against the limit
  | if e.addr ∉ {c.addr|c ∈ CoinsIn} then
  |   | spent ← spent + e.value;
  | end
end
return VerifyCounter(CTRcm^old,CTRcm^new,
                spent,"limit" );
```

Algorithm 4: Spending limit

```
PrivateInput : CoinsIn, CoinsOut, CTRcm^old, CTRcm^new
Constants    : taxrate, tag
output       : True if transaction is valid

spent ← 0;
for e ∈ CoinsOut do
  | if e.addr ∉ CoinsIn then
  |   | spent ← spent + e.value;
  | end
end
return VerifyCounter(CTRcm^old,CTRcm^new,
                ⌊spent · taxrate⌋,tag);
```

Algorithm 5: Tax

transaction. Because Zerocash is payment based, in practice we charge taxes on the sending side of the transaction. Obviously, this cost can be passed on to the recipient.

The tax algorithm is straight forward. All outputs being sent to other parties are summed and a percentage of that amount is added to a user's tax counter. We reveal the tag to support multiple tax authorities.[1]

Actually forcing users pay taxes is a problem left to the authority. Since they know who they issued tax counters to, they can force those people to report their income. An alternate policy would be to directly pay taxes off each transaction. This would, however, leak transaction amounts to the authority, rather than just total income.

5.5 Identity escrow

PrivateInput : CoinsIn.CoinsOut, r
PublicInput : escrowct
Constants : CA
output : True if transaction is valid

return escrowct $= Pkenc(\mathsf{CA}, \mathsf{CoinsIn} \| \mathsf{CoinsOut}; r)$;

Algorithm 6: Identity escrow

The identity escrow policy allows governments to selectively revoke the anonymity of individual transactions, using a master key held securely by some tracing authority. The implementation of this function is simple: a copy of the transaction details is encrypted under the public key of some authority. The policy validates that the encryption is correct. We note it may be possible to forgo public key encryption if every user has a pre-shared key with the tracing authority (since it is likely this would be a bank or government which users already had to register with, this is possible) and we can authenticate the key with a signature scheme if this proves more efficient.

6 Coin tracing

We now detail how to trace individual coins.

6.1 Construction

In a coin tracing scheme, individual coins can be marked for tracing. All subsequent coins resulting from transactions on those initial coins will themselves be traceable. To implement this in Algorithm 7, each coin commitment contains a fresh key for a public key encryption scheme. The idea is that all of the information needed to trace the output coins (including the new private keys) is encrypted under the existing key for the input coin commitments. Thus, if the inputs are traced, then the outputs are traceable. If we did only this, however, the sender would be able to trace the coins because he generated the fresh keys for the outputs. Thus, we encrypt all of this output under the authority's public key. For space efficiency, instead of encrypting everything under each input key, we encrypt the same symmetric key under each public key and encrypt everything under that key.

[1] We note that this tax policy is ill defined if users collude to generate a transaction using MPC where the input identities are different. This can be fixed by ensuring that there is only one identity used for all input coins. However, the same MPC mechanism could be used to share one identity and never pay taxes.

```
PrivateInput : CoinsIn, CoinsOut, r, pkCTs, rEnc, k, ct, rKeyGen, pk, sk
PublicInput  : escrowct
Constants    : 𝒢
Constants    : CA
output       : True if transaction is valid

if sig ≠ ⊥ then                          // tx is from e.g a bank, skip tracing
  | return VerifySig(CA, CoinsOut, sig)
end
i, j ← 0;
for e ∈ CoinsOut do                  // check new coins' tracing keys are correct
  | if (pk[i], sk[i]) ≠ PKgen(𝒢; rKeyGen[i]) or e.key ≠ pk[i] then
  |   | return False;                          // key was not generated honestly
  | end
  | i ← i + 1;
end
for e ∈ CoinsIn do              // ensure k is encrypted under each tracing key
  | if pkCTs[j] ≠ PKEnc(e.key, k; rEnc[j]) then
  |   | return False ; // symmetric key k is not encrypted under input coin j's
  |   | tracing key
  | end
  | j ← j + 1;
end
if ct ≠ enc(k, CoinsIn‖CoinsOut‖sk; rEnc) then
  | return False;            // new tracing private keys not encrypted under k
end
// ensure ct is encrypted under authority key
return escrowct = PkEnc(CA, ct‖pkCTs );
```

Algorithm 7: Coin tracing

Our tracing scheme has two major limitations: first unless someone removes the tracing key for a coin and replaces it with a dummy key, eventually all coins in the network will be traced. This is an inherent limitation of a tracing mechanism that marks all coins involved in a transaction as tainted. Of course, we could have more limited versions where we map the taint bits from input coins to output coins directly (e.g., the largest input's taint bit maps to the largest output) but these lead to circumvention issues.

The second issue is that the user and the authority can collude to trace coins even if the input coins were not marked as traced. This is not an inherent limitation but rather arises from the fact that the authority always knows the decryption keys for the outer layer of encryption and the user always knows the inner layer keys.

Because of these limitations we anticipate that in a real system, coins will frequently return to exchanges and online wallets and at these locations tracing can be removed or added. To facilitate this we allow exchanges who have an authorized signing key to either add or remove tracing by changing the key associated with a coin. We note that when an exchange removes tracing this can be done in a verifiable manner by using a known dummy key as the coin key. Looking forward, we can use similar techniques to accountable user tracing in the next section to allow the exchange

to accountably add or remove tracing by providing a randomized tracing key that is either the authority's key or a dummy key.

Given the above policy, the algorithm to actually do tracing is detailed in Algorithm 8.

```
input  : The list of trace ciphertexts TraceCTs, the authority's decryption key sk, the initial
         coin tracing private key ck for the coin we want to trace
output : The trace for a particular coin Trace

for e ∈ TraceCTs do                // decrypt the outer layer of all transactions
  | TraceRecords.append(PKDec(sk,TraceCTs));
end
KeysToTrace.append(ck);
for ck ∈ KeysToTrace do
  | // For every public key ciphertext in every pour, we attempt to decrypt
  |    with the target key.  On success, we get back the symmetric ciphertext
  |    associated with the pour, and the decrypted symmetric key.
  | encrec, k ← FindTxByTrialDecryption(TraceRecords, ck);
  | if encrec ≠ ⊥ then
  |   | rec ← SymDec(k, encrec);                          // now decrypt the record
  |   | Trace.append(rec);                  // add the decrypted record to the trace
  |   | for i ∈ {0, 1, ..., rec.CoinsIn.Size() − 1} do
  |   |   | KeysToTrace.append(rec.sk[i]);              // enqueue the new tracing keys
  |   | end
  | end
end
return Trace;
```

Algorithm 8: TraceCoin

The algorithm to trace coins takes an initial target coin commitment, the initial public key for that coin commitment, and the key to decrypt all tracing records. It simply searches for all coin commitments output by that transaction, maps the new keys to those coin, decrypts the record with the existing keys, and adds those coin commitments to the list of ones to search for.

6.2 Security

6.2.1 Modified ideal functionality

To allow tracing, we need to modify the Ideal Functionality to (1) have a special party designated the tracing authority, (2) allow coins to be marked as traced by that authority, (3) mark all output coins as traced if any of the inputs are traced and (4) allow a report on all traced coins to be made to that authority. Of course, since we allow static corruption, the tracing authority may be the adversary. We detail the modifications in Figure 2.

6.2.2 Proof sketch

If the tracing authority is not corrupted, then the simulator can always use random values for the escrow ciphertext and the probability of abort is bounded by the security of the outer encryption scheme.

- $[\mathsf{c}_1^{new}, \ldots, \mathsf{c}_n^{new}] \leftarrow \mathsf{Pour}([\mathsf{c}_1, \ldots, \mathsf{c}_m], [(\mathsf{addr}_{\mathsf{pk},1}, v_1), \ldots, (\mathsf{addr}_{\mathsf{pk},n}, v_n)], v_{\mathsf{pub}}, aux)$.
  - Perform steps one through eight of Pour in Figure 1, adding an additional stored field (initially set to false) to denote whether a coin is being traced or not.
  - If any coin c_i is marked as traced, mark all output coins as traced.
  - T_P returns $[\mathsf{c}_1^{new}, \ldots, \mathsf{c}_n^{new}]$ to the user.
- $\mathsf{Trace}(\mathsf{c}, U)$
  - The input is a coin c and a requesting party U.
  - If the requesting party U is the tracing authority, update the stored value for c and mark it as traced. Else, reject.
- $[\mathsf{c}_0, \ldots, \mathsf{c}_m] \leftarrow \mathsf{Report}(U)$
  - The T_P returns all entries $[\mathsf{c}_0, \ldots, \mathsf{c}_m]$ for coins marked as traced if the requesting party U is the tracing authority, otherwise return $\perp$.

Figure 2: Modified ideal functionality for coin tracing.

We now deal with the more complex case: a corrupted tracing authority who the simulator must generate notifications for. There are two subcases: traced coins and non-traced coins. For traced coins, the probability that the simulator fails in decrypting both layers of the tracing ciphertext (and hence cannot mark the coins appropriately in the ideal functionality) is bounded by the soundness of the proof system. For non-traced coins in the ideal model, the simulator can use a random ciphertext for the inner symmetric ciphertext ct and the associated encryptions of its key. For untraced coins, since the real world tracing authority does not have the keys for non-traced coins, the probability of abort is bounded by the security of the public key encryption scheme.

7 Accountable user tracing

```
PrivateInput : CoinsIn, CoinsOut, pk, sig, r
PublicInput  : usertracect
Constants    : CA
output       : True if transaction is valid

if VerifySig(CA, pk, sig) then
  | return usertracect = Pkenc(pk, CoinsIn||CoinsOut; r);
else
  | return False;
end
```

Algorithm 9: Accountable user tracing

The problem with the original escrow above is that it provides no privacy from the tracing authority: all users' data is encrypted under a single key the authority holds. The only privacy guarantees users have are based on trust. And even those are limited: in order to find the coins used by a specified user, the authority needs to decrypt all coins. Ideally, we would have a system that

not only allows the authority to only open coins of traced users, it creates an accountable record that they did so.

To do so, we borrow an idea from Accountable Tracing Signatures [17]: each user encrypts their data under a unique key they are given by the authority. If they are being traced, this key is a randomized version of the authority's public key. If, as is the common case, users are not being traced, the key is a randomized version of a null key (e.g. a random group element). Crucially, without knowledge of the randomness, users cannot distinguish which key they are getting at the time, but if the authority later reveals the randomness they can provably tell they were or were not traced. Thus we term this system accountable user tracing.

This requires one small change to the standard identity escrow policy: we have to allow an arbitrary key and ensure that it is signed by the authority and for the current epoch.

The larger question is key management. If we simply require the authority to reveal who they were monitoring at some later point, this policy becomes simple to implement: each user receives a key at each time period (e.g. daily) and by the disclosure deadline for searches, the authority reveals the random coins used to make the key. Unfortunately, this approach is unlikely to be compatible with real-world requirements. Given the existence of National Security Letters and similar warrants containing "gag orders", it is likely there are cases where the authority will prefer not to reveal which users were monitored. The standard solution in practice is to produce a transparency report that details which fraction of users were searched, and/or offers a loose bound on the number of searches.

However, building a cryptographically binding transparency report over millions of users efficiently is challenging given that this would involve processing millions of records. There are various ways this might be accomplished (e.g. techniques for batched proofs). In this work we propose a simple one: use a separate instance of Zerocash to create tokens that allow search. Each user gets their key along with a proof that their key is either randomized from the null key or randomized from the authority's key and a zerocoin is spent. At the end of each time period, the authority can spend all unused coins, thus revealing what fraction of people they searched.

8 Related work

8.1 Anti-money laundering for centralized e-cash

Many, if not all of the policies in this paper have been implemented in centralized e-cash schemes, be it spending limits [8] or user and coin tracing. Somewhat surprisingly accountable user and coin tracing exists for centralized schemes [19, 20]. However none of these have been done in the decentralized setting. Moreover, the known techniques for coin and user tracing [19, 20] require coins to be withdrawn from the authority and then deposited back and have only one transfer, whereas our techniques allow continual transactions with the need to interact with the authority only dictated by the allowable delay in starting a trace.

8.2 Hawk

In concurrent work Kosba et al. [18] present Hawk, a system for privacy preserving smart contracts with an ideal functionality based definition. Hawk's security is set in the UC model [9], which is a stronger setting than we explore here. Achieving UC security requires all proofs to include an encrypted copy of their witness and a proof of its validity. This entails two costs: first, the increase in proving cost due to circuits handling encryption and second, the loss of succinctness. In the case of Zerocash, this results in a dramatic increase in the size of the proof due to the need to encrypt

all witnesses.[2] This trade off between security and performance does not appear to affect our actual constructions here: if built using a UC security proof scheme, they can achieve UC security.

9 Acknowledgements

This work was supported by: The National Science Foundation under awards EFRI-1441209 and CNS-1414023; Google ATAP; The Mozilla Foundation; and the Office of Naval Research under contract N00014-14-1-0333.

References

[1] Abe, M., Fehr, S.: Perfect NIZK with adaptive soundness. In: TCC. Lecture Notes in Computer Science, vol. 4392, pp. 118–136 (2007)

[2] Barber, S., Boyen, X., Shi, E., Uzun, E.: Bitter to better - how to make Bitcoin a better currency. In: Proceedings of the 16th International Conference on Financial Cryptography and Data Security. pp. 399–414. FC '12 (2012)

[3] Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: Security and Privacy (SP), 2014 IEEE Symposium on. pp. 459–474. IEEE (2014)

[4] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Proceedings of the 33rd Annual International Cryptology Conference. pp. 90–108. CRYPTO '13 (2013)

[5] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von Neumann architecture. In: Proceedings of the 23rd USENIX Security Symposium. Security '14 (2014), available at `http://eprint.iacr.org/2013/879`

[6] Biryukov, A., Khovratovich, D., Pustogarov, I.: Deanonymisation of clients in bitcoin p2p network. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 15–29. ACM (2014)

[7] Bitansky, N., Chiesa, A., Ishai, Y., Ostrovsky, R., Paneth, O.: Succinct non-interactive arguments via linear interactive proofs. In: Proceedings of the 10th Theory of Cryptography Conference. pp. 315–333. TCC '13 (2013)

[8] Camenisch, J., Hohenberger, S., Lysyanskaya, A.: Balancing accountability and privacy using e-cash. In: Security and Cryptography for Networks, pp. 141–155. Springer (2006)

[9] Canetti, R.: Universally Composable Security: A new paradigm for cryptographic protocols. In: FOCS '01. p. 136. IEEE Computer Society (2001), `http://eprint.iacr.org/2000/067`

[10] Danezis, G., Fournet, C., Kohlweiss, M., Parno, B.: Pinocchio coin: building zerocoin from a succinct pairing-based proof system. In: Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies. pp. 27–30. ACM (2013)

[2] It may be possible to reduce the cost of these proofs by introducing zero-knowledge proofs that are only partially extractable, although this technique is not described by the Hawk authors. For example, the exact Merkle tree path need not be extracted from the zkSNARK proof.

[11] Dent, A.W.: Adapting the weaknesses of the random oracle model to the generic group model. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 2501, pp. 100–109 (2002)

[12] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques. pp. 626–645. EUROCRYPT '13 (2013)

[13] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcps. In: EUROCRYPT. vol. 7881, pp. 626–645. Springer (2013)

[14] Goldreich, O.: Foundations of cryptography: volume 2, basic applications. Cambridge university press (2004)

[15] Groth, J.: Simulation-sound NIZK proofs for a practical language and constant size group signatures. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 4248, pp. 444–459 (2006)

[16] Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security. pp. 321–340. ASIACRYPT '10 (2010)

[17] Kohlweiss, M., Miers, I.: Accountable tracing signatures. Cryptology ePrint Archive, Report 2014/824 (2014), `http://eprint.iacr.org/`

[18] Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts (2015)

[19] Kügler, D., Vogt, H.: Auditable tracing with unconditional anonymity. In: International Workshop on Information Security Application – WISA 2001. pp. 151–163 (2001)

[20] Kügler, D., Vogt, H.: Offline payments with auditable tracing. In: Financial cryptography. pp. 269–281. Springer (2003)

[21] Lipmaa, H.: Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In: Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography. pp. 169–189. TCC '12 (2012)

[22] Lipmaa, H.: Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In: Proceedings of the 19th International Conference on the Theory and Application of Cryptology and Information Security. pp. 41–60. ASIACRYPT '13 (2013)

[23] Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: Anonymous distributed e-cash from bitcoin. In: Security and Privacy (SP), 2013 IEEE Symposium on. pp. 397–411. IEEE (2013)

[24] Parno, B., Gentry, C., Howell, J., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: Proceedings of the 34th IEEE Symposium on Security and Privacy. pp. 238–252. Oakland '13 (2013)

[25] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: IEEE Symposium on Security and Privacy. pp. 238–252. IEEE Computer Society (2013)

[26] Reid, F., Martin, H.: An analysis of anonymity in the Bitcoin system. In: Proceedings of the 3rd IEEE International Conference on Privacy, Security, Risk and Trust and on Social Computing. pp. 1318–1326. SocialCom/PASSAT '11 (2011)

[27] Wilcox, Z.: Personal communication (May, 2015)

A Simulation sound Succinct non-interactive arguments of knowledge

A.1 Notation

Given two functions $f, g : \mathbb{N} \to [0,1]$ we write $f(\lambda) \approx g(\lambda)$ when $|f(\lambda) - g(\lambda)| = \lambda^{-\omega(1)}$. We say that f is *negligible* when $f(\lambda) \approx 0$ and that f is *overwhelming* when $f(\lambda) \approx 1$.

We write $y = A(x; r)$ when the algorithm A on input x and randomness r, outputs y. We write $y \leftarrow A(x)$ for the process of picking randomness r at random and setting $y = A(x; r)$. We also write $y \leftarrow S$ for sampling y uniformly at random from the set S. We will assume it is possible to sample uniformly at random from sets such as $\mathbb{Z}_p$.

Following Abe and Fehr [1] we write $(y; z) \leftarrow (\mathcal{A} \parallel \mathcal{X}_{\mathcal{A}})(x)$ when $\mathcal{A}$ on input x outputs y and $\mathcal{X}_{\mathcal{A}}$ on the same input (including random coins) outputs z. I.e., as a shorthand for $r \leftarrow \{0,1\}^*; y \leftarrow \mathcal{A}(x; r); z \leftarrow \mathcal{X}_{\mathcal{A}}(x, r)$.

When dealing with adversaries that have access to oracles $\mathcal{O}$, e.g. a simulator, it is necessary to augment the notation both so $\mathcal{A}$ gets access to the simulator $\mathcal{O}$, and that the extractor $\mathcal{X}_{\mathcal{A}}$ has the ability to emulate the oracles, including access to their random coins.

We write $(y; z) \leftarrow (\mathcal{A}^{\mathcal{O}(x_2,\cdot)} \parallel \mathcal{X}_{\mathcal{A}}^{(x_2)})(x_1)$ as a shorthand for $r_1, r_2 \leftarrow \{0,1\}^*; y \leftarrow \mathcal{A}^{\mathcal{O}(x_2;r_2)}(x_1; r_1)$; $z \leftarrow \mathcal{X}_{\mathcal{A}}(x_1, x_2, r_1, r_2)$.

A.2 Definitions

Let $\{\mathbb{R}_\lambda\}_{\lambda \in \mathbb{N}}$ be a sequence of families of efficiently decidable binary relations R. For pairs $(u, w) \in R$ we call u the statement and w the witness. A non-interactive argument for $\{\mathbb{R}_\lambda\}_{\lambda \in \mathbb{N}}$ is a quadruple of efficient algorithms $(\mathsf{Setup}, \mathrm{Prove}, \mathrm{VfyProof}, \mathrm{Sim})$.

$(\sigma, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R)$**:** The setup takes as input a security parameter λ and a relation $R \in \mathcal{R}_\lambda$ and returns a common reference string σ and a simulation trapdoor τ for the relation R.

$\pi \leftarrow \mathrm{Prove}(\sigma, u, w)$**:** The prover algorithm takes as input a common reference string σ for a relation R and $(u, w) \in R$ and returns an argument π.

$0/1 \leftarrow \mathrm{VfyProof}(\sigma, u, \pi)$**:** The verification algorithm takes as input a common reference string, a statement u and an argument π and returns 0 (reject) or 1 (accept).

$\pi \leftarrow \mathrm{Sim}(\tau, u)$**:** The simulator takes as input a simulation trapdoor and statement u and returns an argument π.

Definition A.1 *We say* $(\mathsf{Setup}, \mathrm{Prove}, \mathrm{VfyProof}, \mathrm{Sim})$ *is a perfect non-interactive zero-knowledge argument of knowledge for* $\{\mathcal{R}_\lambda\}_{\lambda \in \mathbb{N}}$ *if it has perfect completeness, perfect zero-knowledge and computation simulation extractable as defined below.*

Perfect completeness. Completeness says that, given any true statement, an honest prover should be able to convince an honest verifier. For all $\lambda \in \mathbb{N}$, $R \in \mathcal{R}_\lambda$, $(u, w) \in R$

$$Pr\Big[(\sigma, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R); \pi \leftarrow \mathrm{Prove}(\sigma, u, w) : \mathrm{VfyProof}(\sigma, u, \pi) = 1\Big] = 1$$

Perfect zero-knowledge. An argument is zero-knowledge if it does not leak any information besides the truth of the statement. We say $(\mathsf{Setup}, \mathrm{Prove}, \mathrm{VfyProof}, \mathrm{Sim})$ is perfect zero-knowledge if for all $\lambda \in \mathbb{N}, R \in \mathcal{R}_\lambda, (u, w) \in R$ and all adversaries $\mathcal{A}$, we have

$$\begin{aligned} & \Pr\Big[(\sigma, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R); \pi \leftarrow \mathrm{Prove}(\sigma, u, w) : \mathcal{A}(\sigma, \tau, \pi) = 1\Big] \\ = \; & \Pr\Big[(\sigma, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R); \pi \leftarrow \mathrm{Sim}(\tau, u) : \mathcal{A}(\sigma, \tau, \pi) = 1\Big]. \end{aligned}$$

Perfect witness-indistinguishability. An argument is witness indistinguishable if it does not leak any information about which witness was used to prove a statement. We say $(\mathsf{Setup}, \mathrm{Prove}, \mathrm{VfyProof}, \mathrm{Sim})$ is perfect witness-indistinguishable if for all $\lambda \in \mathbb{N}, R \in \mathcal{R}_\lambda, (u, w_0) \in R, (u, w_1) \in R$ and all adversaries $\mathcal{A}$, we have

$$\begin{aligned} & \Pr\Big[(\sigma, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R); \pi \leftarrow \mathrm{Prove}(\sigma, u, w_0) : \mathcal{A}(\sigma, \tau, \pi) = 1\Big] \\ = \; & \Pr\Big[(\sigma, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R); \pi \leftarrow \mathrm{Prove}(\sigma, u, w_1) : \mathcal{A}(\sigma, \tau, \pi) = 1\Big]. \end{aligned}$$

Computation knowledge soundness. We call $(\mathsf{Setup}, \mathrm{Prove}, \mathrm{VfyProof}, \mathrm{Sim})$ an argument of knowledge if there is an extractor that can compute a witness whenever the adversary produces a valid argument. The extractor gets full access to the adversary's state, including any random coins.

Formally, we require that, for all sequences $(R_\lambda)_{\lambda \in \mathbb{N}}$ of polynomially bounded relations in $\{\mathbb{R}_\lambda\}_{\lambda \in \mathbb{N}}$ and non-uniform polynomial time adversaries $\mathcal{A}$, there exists a non-uniform polynomial time extractor $\mathcal{X}_\mathcal{A}$ such that

$$\begin{aligned} \Pr\Big[(\sigma, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R_\lambda); ((u, \pi); w) \leftarrow (\mathcal{A} \parallel \mathcal{X}_\mathcal{A})(\sigma) : \\ (u, w) \notin R_\lambda \text{ and } \mathrm{VfyProof}(\sigma, u, \pi) = 1\Big] \approx 0 \end{aligned}$$

For composition we need a slight variant of this definition with auxiliary input generators: Knowledge soundness for the class $\mathcal{Z}$ of auxiliary input generators requires that, for all sequences $(R_\lambda)_{\lambda \in \mathbb{N}}$ of polynomially bounded relations in $\{\mathbb{R}_\lambda\}_{\lambda \in \mathbb{N}}$, all non-uniform polynomial time auxilliary input generator $Z \in \mathcal{Z}$, and all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a non-uniform polynomial time extractor $\mathcal{X}_\mathcal{A}$ such that

$$\begin{aligned} \Pr\Big[z \leftarrow Z(1^\lambda); (\sigma, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R_\lambda); \\ ((u, \pi); w) \leftarrow (\mathcal{A} \parallel \mathcal{X}_\mathcal{A})(\sigma, z) : \\ (u, w) \notin R_\lambda \text{ and } \mathrm{VfyProof}(\sigma, u, \pi) = 1\Big] \approx 0. \end{aligned}$$

More complex composition in which the extractor is used in a reduction in which he does not have access to the randomness used to generate public input requires interactive input generators:

$$\begin{aligned} \Pr\Big[(\sigma, \tau) \leftarrow \mathsf{Setup}(1^\lambda, R_\lambda); ((u, \pi); w) \leftarrow (\mathcal{A} \parallel \mathcal{X}_\mathcal{A})^{Z(.)}(\sigma) : \\ (u, w) \notin R_\lambda \text{ and } \mathrm{VfyProof}(\sigma, u, \pi) = 1\Big] \approx 0. \end{aligned}$$

Discussion. If all $Z \in \mathcal{Z}$ only ever accept 1^λ the two assumptions are equivalent. But Z may for instance take a message as input and provide a ciphertext under some fixed chosen plain-text secure scheme as output. If on the other hand the encryption scheme is simulatable as defined by [11] (which also implies that it is necessarily chosen-ciphertext secure), then Z can be simulated using the random tape that is already accessible to $\mathcal{A}$ and $\mathcal{X}_\mathcal{A}$ in the assumption without auxiliary input.

This formalizes the intuition that it is the absence of interactive auxiliary input that causes the impossibility result of [11]. Stronger knowledge assumptions could be used to circumvent it. If of course the proof system is malleable it may not be of much use for constructing CCA secure schemes from CPA secure schemes. This is what we consider next.

Computational simulation knowledge soundness.

Computational simulation knowledge soundness is defined analogously, however, the attacker has access to a simulator, the extractor has access to the simulation trapdoor, and the attacker is not allowed to return a proof in the set Q of proofs output by the simulator.

$$\Pr\Big[(\sigma,\tau) \leftarrow \mathsf{Setup}(1^\lambda, R_\lambda); \\ ((u,\pi);w) \leftarrow (\mathcal{A}^{Sim(\tau,\cdot)} \parallel \mathcal{X}_\mathcal{A}^{(\tau)})^{Z(.)}(\sigma) : \\ (u,w) \notin R_\lambda \wedge \mathrm{VfyProof}(\sigma,u,\pi) = 1 \wedge (u,\pi) \notin Q\Big] \approx 0.$$

A.3 Construction of simulation-sound SNARKs

A.3.1 Simulation-sound SNARKs from symmetric primitives

Since some SNARKS are circuit based, constructions involving bitwise manipulation as opposed to field arithmetic can be desirable. As such, we construct a OR-simulator signature style construction but with a PRF and a MAC instead of a signature.

We construct a simulation-sound SNARK $\Pi_{sim}$with input generators $\mathcal{Z}$ for $\mathbb{R}_\lambda$ from a witness--indistinguishable SNARK (Setup, Prove, VfyProof, Sim) with input generators $\mathcal{Z}^{OR}$ for $\mathbb{R}_\lambda^{OR}$, a **PRF**, an EU-CMA secure message authentication code, and a strongly-unforgeable one-time signature scheme ($\mathrm{Kg_{OT}}, \mathrm{Sign_{OT}}, \mathrm{Verify_{OT}}$).

In the CRS, we have a element $pk = \mathbf{PRF}(sk)$. The key sk is the simulator's trapdoor. The relation class $\mathbb{R}_\lambda^{MAC}$ is defined such that for every $R_\lambda \in \mathbb{R}_\lambda$ there is an (efficiently computable) relation

$R_\lambda^{MAC} \in \mathbb{R}_\lambda^{MAC}$ with

$$((u,t,pk,opk),w) \in R_\lambda^{MAC} \text{ iff} \\ (u,w) \in R_\lambda \vee (pk = \mathbf{PRF}(sk) \wedge t = \mathrm{MAC}(opk||sk))$$

$\mathcal{Z}^{OR}$ is $\mathcal{Z}$ with prf "keypair" generation $(pk,sk) \leftarrow \mathrm{Kg}(1^\lambda)$ added to the output of all Z.

For efficiency reasons, we reveal the MAC tag t. This is not necessary for correctness. However, as t is indistinguishable from random, it leaks nothing.

Π_{SS} is defined as follows:

$(\sigma,\tau) \leftarrow \mathrm{SS}.\mathsf{Setup}(1^\lambda, R)$**:** On input λ and relation $R_\lambda \in \mathcal{R}_\lambda$ compute the "prf" key pair $(pk,sk) \leftarrow \mathrm{Kg}(1^\lambda)$ by sampling a random r and outputting $(r, \mathbf{PRF}(r))$, and the proof setup $(\sigma',\tau') = \mathsf{Setup}(1^\lambda, R^{OR})$. R^{OR} is the relation that corresponds to R_λ in $\mathbb{R}_\lambda^{OR}$. The simulation trapdoor τ' is not needed and deleted. Return the reference string $\sigma = (\sigma', pk)$ and the simulation trapdoor $\tau = sk$. We assume that $sk = \tau$ contains pk or that pk can be computed from sk.

$\pi \leftarrow$ SS.Prove(σ, u, w)**:** Parse σ as (σ', pk) and compute $(opk, osk) \leftarrow \mathrm{Kg}_{\mathrm{OT}}(1^\lambda)$. Compute the OR proof with a witness for the left side: $\pi' \leftarrow \mathrm{Prove}(\sigma', (u, pk, opk), (w, \perp))$. Compute the signature $s_{\mathrm{OT}} \leftarrow \mathrm{Sign}(osk, (\pi', u))$ over the proof and the instance; return the proof $\pi = (opk, \pi', s_{\mathrm{OT}})$.

$0/1 \leftarrow$ SS.VfyProof(σ, u, π)**:** Parse σ as (σ', pk) and π as $(opk, \pi', s_{\mathrm{OT}})$. Return 1 (accept) if both $\mathrm{VfyProof}(\sigma', (u, pk, opk), \pi) = 1$ and $\mathrm{Verify}_{\mathrm{OT}}(opk, (\pi', u), s_{\mathrm{OT}}) = 1$, and 0 (reject) otherwise.

$\pi \leftarrow$ SS.Sim(τ, u)**:** The simulator uses τ as sk to compute a righ-side witness and proof for the OR: $opk, osk \leftarrow \mathrm{Kg}_{\mathrm{OT}}(1^\lambda)$, $t \leftarrow \mathrm{MAC}(sk, opk)$, $\pi' \leftarrow \mathrm{Prove}(\sigma', (u, pk, opk), (\perp, t))$. Compute the signature $s \leftarrow \mathrm{Sign}_{\mathrm{OT}}(osk, (\pi', u))$ and return the argument $\pi = (opk, \pi', s)$.

For simplicity we give a proof sketch for non-interactive input generators.

- Game 1. The original computational simulation knowledge soundness game.
- Game 2. The same as Game 1, except that we run an extractor $(w, s) \leftarrow \mathcal{X}_{\mathcal{A}'}$ described below in parallel and abort if $((u, t, pk, opk), w) \notin R^{MAC}$.

  Let $\mathcal{A}'$ be an adversary for the knowledge soundness with auxiliary input game with auxiliary input generator $(z, pk, sk) \leftarrow Z(1^\lambda)$ for relation class $\mathbb{R}_\lambda^{OR}$. We define $\mathcal{A}'$ as $((u, pk, opk), \pi') \leftarrow \mathcal{A}'(\sigma, (z, sk = \tau, pk))$ as run $(u, (opk, \pi', s_{\mathrm{OT}})) \leftarrow \mathcal{A}^{Sim(\tau,\cdot)}(\sigma, z)$; return $((u, pk, opk), \pi')$. Let $\mathcal{X}_{\mathcal{A}'}$ be the extractor for $\mathcal{A}'$ whose existence is stipulated by the knowledge soundness assumption with auxiliary input.

  By the computational knowledge assumption, the probability of $\mathcal{X}_{\mathcal{A}'}$ aborting is negligible. We are simply invoking the standard extractor and there is no simulator (yet).
- Game 3. The same as Game 2, but abort if an existing opk was reused by $\mathcal{A}$ to sign another pair (π', u). Because opk is a key for a strongly-unforgeable one time signature scheme, this probability is negligible.
- Game 4. Abort, if $\mathcal{A}$ returned a verifying MAC on a fresh opk. There are three ways this can happen. First, $\mathcal{A}$ extracted sk from pk, if **PRF** is a secure PRF, the probability of this is negligible. Second, $\mathcal{A}$ provided some different sk such that $pk = \mathbf{PRF}(sk)$. Since **PRF** is a permutation, the probability of this event is also negligible. Finally, $\mathcal{A}$ could have forged the MAC. Since the MAC is EU-CMA secure, this is also negligible.
- Game 5. It remains to construct $\mathcal{X}_{\mathcal{A}}^{\tau}$, the extractor operating on $\mathcal{A}$ with access to the simulator out of $\mathcal{X}_{\mathcal{A}'}$, the extractor operating on an attacker $\mathcal{A}'$ who emulates the simulator and runs $\mathcal{A}$. This is a simple matter of having the extractor directly emulate the simulator. Thus we construct $\mathcal{X}_{\mathcal{A}}^{\tau}$ out of $\mathcal{X}_{\mathcal{A}'}$.

Since Game 5 behaves the same as Game 4, and $\mathcal{X}_{\mathcal{A}}^{\tau}$ always extracts valid witnesses, our scheme is is computationally simulation knowledge sound.

A.3.2 Concrete efficiency for Zerocash

Naturally, this construction adds some overhead to a standard proof scheme for any relation R_λ because of the necessary overhead for the MAC and PRF in the cirucit and the small overhead of the one time signature. However, the existing Zerocash construction already uses a similar technique to bind the Bitcoin transaction to the Zerocash proof: a hash function $\mathcal{H}$ is used to compute two MACs on a one time signature key, using the private key for each of the two input coins. If we instantiate

the PRF with a hash function (as is done elsewhere in Zerocash), then simulation soundness has the exact same cost as the original Zerocash construction (modulo the cost of one additional OR gate) but offers far better theoretical properties. We effectively get simulation sound Zerocash for free. Indeed, if we extend Zerocash to take more than two coins in and two coins out (as is necessary for some of our policies), we actually get improved performance, since we only invoke the MAC once instead of once per coin.

B Zerocash Modifications

Mint

- INPUTS:
  - public parameters *params*
  - coin value $v \in \{0, 1, \ldots, v_{\mathsf{max}}\}$
  - destination address public key $\mathsf{addr}_{\mathsf{pk}}$
- OUTPUTS: coin 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 $\mathsf{PRF}^{\mathsf{sn}}$ seed ρ.
3. Randomly sample two COMM trapdoors r, s.
4. Compute $k := \mathsf{COMM}_r(a_{\mathsf{pk}} \| \rho)$.
5. Compute $\mathsf{cm} := \mathsf{COMM}_s(v \| k)$.
6. Compute $\pi_{\mathtt{MINT}} := \mathsf{Prove}(\mathsf{pk}_{\mathtt{MINT}}, (k, a_{\mathsf{pk}}), (v, \mathsf{cm}))$.
7. Set $c := (\mathsf{addr}_{\mathsf{pk}}, v, \rho, r, s, \mathsf{cm})$.
8. Set $\mathsf{tx}_{\mathsf{Mint}} := (\mathsf{cm}, v, \pi_{\mathtt{MINT}})$,
9. Output c and $\mathsf{tx}_{\mathsf{Mint}}$.

VerifyTransaction

- INPUTS:
  - public parameters *params*
  - a (mint or pour) transaction 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 $(\mathsf{cm}, v, \pi_{\mathtt{MINT}})$.
   (b) Output $b := \mathsf{Verify}(\mathsf{vk}_{\mathtt{MINT}}, \pi_{\mathtt{MINT}})$
2. If given a pour transaction $\mathsf{tx} = \mathsf{tx}_{\mathsf{Pour}}$:
   (a) Proceed as in [3]

Receive

- INPUTS:
  - public parameters *params*
  - 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 $(\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, \mathsf{info}, *)$, and $*$ as $(\mathsf{pk}_{\mathsf{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, \mathbf{C}_1, \mathbf{C}_2, \sigma)$.
   (b) For $i \in \{1, 2\}$:
      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:
         - $\mathsf{cm}_i^{\mathsf{new}}$ equals $\mathsf{COMM}_{s_i}(v_i \| \mathsf{COMM}_{r_i}(a_{\mathsf{pk}} \| \rho_i))$;
         - Check that ρ_i is distinct from any previously received coins.
      iii. If both checks succeed, output $c_i := (\mathsf{addr}_{\mathsf{pk}}, v_i, \rho_i, r_i, s_i, \mathsf{cm}_i^{\mathsf{new}})$.

Figure 3: Modified Mint, VerifyTransaction, and Receive algorithms.

C Proof Sketch for Theorem 2.2

We now present a sketch of the proof of Theorem 2.2.

Proof. We now show that under the assumption of computationally simulation-knowledge-sound zkSNARKs, collision-resistant and statistically hiding commitments, and the existence of a trusted, global append-only log, then for every real-world adversary $\mathcal{A}$ against our modified scheme there exists an ideal-world adversary $\mathcal{A}'$ such that the output of the real-world and ideal-world adversaries are computationally indistinguishable. We will now describe the operation of the ideal-world adversary $\mathcal{A}'$, which runs $\mathcal{A}$ internally and interacts with the ideal functionality. Recall that we assume a static corruption model where $\mathcal{A}$ (resp. $\mathcal{A}'$) emulates a set of statically-corrupted parties, and can read and post transactions to the ledger via a standard interface. Our proof assumes the existence of a global, trusted transaction ledger, which we use as a black box. We leave a complete proof that considers this construction and models it to future work.

We begin by sketching the operation of the ideal-world adversary $\mathcal{A}'$.

Minting a coin When $\mathcal{A}$ emits a Mint transaction to the ledger, the ideal adversary $\mathcal{A}'$ first parses the transaction to obtain $(c, \pi_{\texttt{MINT}})$ and verifies that $\pi_{\texttt{MINT}}$ is valid. If the proof is valid, it runs the knowledge extractor on $\pi_{\texttt{MINT}}$ to obtain $(v, \mathsf{addr}_{\mathsf{pk}})$. $\mathcal{A}'$ checks to make sure that the address $\mathsf{addr}_{\mathsf{pk}}$ has been registered on its list of users. If it has not, $\mathcal{A}'$ calls $\mathsf{RegisterID}(\mathsf{addr}_{\mathsf{pk}})$ and stores $(U', \mathsf{addr}_{\mathsf{pk}})$ where U' is an arbitrary corrupted user. Once $\mathcal{A}'$ has completed these steps, it calls the ideal functionality on $\mathsf{Mint}(v, \mathsf{addr}_{\mathsf{pk}})$ to obtain c. $\mathcal{A}'$ maintains a list of coins minted by corrupted users; it records $(\mathsf{addr}_{\mathsf{pk}}, U, c, \pi_{\texttt{MINT}})$ to this list.

When T_P notifies $\mathcal{A}'$ that some honest user has minted a coin, $\mathcal{A}'$ creates a coin c and uses the simulator for the proof $\pi_{\texttt{MINT}}$ to simulate the associated proof. It then records the resulting coin information $(c, \pi_{\texttt{MINT}})$ on the ledger.

Pouring coins When $\mathcal{A}$ emits $\pi_{\texttt{POUR}}$ to the ledger, $\mathcal{A}'$ first verifies the proof as in the real protocol (rejecting if it is invalid). If the proof verifies, it runs the knowledge extractor on $\pi_{\texttt{POUR}}$ to obtain $v_0, v_1, \mathsf{addr}_{\mathsf{pk},0}, \mathsf{addr}_{\mathsf{pk},1}, c_0^{old}, c_1^{old}, c_0^{new}, c_1^{new}$. If either $\mathsf{addr}_{\mathsf{pk},0}$ and $\mathsf{addr}_{\mathsf{pk},1}$ have not previously been registered, $\mathcal{A}'$ also runs $\mathsf{RegisterID}(\mathsf{addr}_{\mathsf{pk},0})$ and/or $\mathsf{RegisterID}(\mathsf{addr}_{\mathsf{pk},1})$ and stores $(\mathsf{addr}_{\mathsf{pk},0})$ and $(\mathsf{addr}_{\mathsf{pk},1})$.$\mathcal{A}'$ then parses c_0^{old} and c_1^{old} to obtain $\mathsf{addr}_{\mathsf{pk},0}^{old}$ and $\mathsf{addr}_{\mathsf{pk},1}^{old}$. The simulator checks to make sure that the addresses $\mathsf{addr}_{\mathsf{pk},0}^{old}$ and $\mathsf{addr}_{\mathsf{pk},1}^{old}$ have been registered on its list of users. If they have not, $\mathcal{S}$ calls $\mathsf{RegisterID}(\mathsf{addr}_{\mathsf{pk},0}^{old})$ and $\mathsf{RegisterID}(\mathsf{addr}_{\mathsf{pk},1}^{old})$ and stores $(U, \mathsf{addr}_{\mathsf{pk},0}^{old})$ and $(U, \mathsf{addr}_{\mathsf{pk},1}^{old})$. Once $\mathcal{A}'$ has registered the users' addresses or verified they are already registered, it looks up c_0^{old}, c_1^{old} to obtain c_0^{IFOld}, c_1^{IFOld}, and it runs $\mathsf{Pour}([(\mathsf{c}_0^{IFOld}, \mathsf{addr}_{\mathsf{pk},0}, v_0), (\mathsf{c}_1^{IFOld}, \mathsf{addr}_{\mathsf{pk},1}, v_1)], v_{\mathsf{pub}})$ to get $\mathsf{c}_0^{IFNew}, \mathsf{c}_1^{IFNew}$. The simulator then transmits the coin information $(c_0^{old}, \mathsf{c}_0^{IFOld}, c_0^{new}, \mathsf{c}_0^{IFNew})$ and $(c_1^{old}, \mathsf{c}_1^{IFOld}, c_1^{new}, \mathsf{c}_1^{IFNew})$ to the trusted store and acknowledges the coins' issuances. $\mathcal{A}'$ stores $(\mathsf{addr}_{\mathsf{pk},0}, c_0^{new}, \pi_{\texttt{POUR}})$ and $(\mathsf{addr}_{\mathsf{pk},1}, c_1^{new}, \pi_{\texttt{POUR}})$ in its list of coins.

When T_P notifies $\mathcal{A}'$ that some honest user has poured coins to another honest user, $\mathcal{A}'$ generates two random coins and associated data $(\mathsf{sn}_0^{\mathsf{old}}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{cm}_0^{\mathsf{new}}, \mathsf{cm}_1^{\mathsf{new}})$ and runs the zero-knowledge simulator for $\pi_{\texttt{POUR}}$ to simulate a proof. It then generates a random signing key $\mathsf{pk}_{\mathsf{sig}}$ for use with Sig to create σ, h_0, and h_1 and creates random ciphertexts $\mathbf{C}_0$ and $\mathbf{C}_1$. It then writes $(c_0, \mathsf{sn}_0^{\mathsf{old}}, \mathsf{cm}_0^{\mathsf{new}}, \mathsf{pk}_{\mathsf{sig}}, h_0, \mathbf{C}_0, \pi_{\texttt{POUR}}, \sigma)$ to U_0 and $(c_1, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{pk}_{\mathsf{sig}}, h_1, \mathbf{C}_1, \pi_{\texttt{POUR}}, \sigma)$ to the ledger.

When T_P notifies $\mathcal{A}'$ that some honest user has poured coins to a user controlled by $\mathcal{A}$, $\mathcal{A}'$ generates coins as it would in the normal protocol (along with associated data $\mathsf{sn}_0^{\mathsf{old}}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{cm}_0^{\mathsf{new}}, \mathsf{cm}_1^{\mathsf{new}}$) and

runs the zero-knowledge simulator for $\pi_{\texttt{POUR}}$ to simulate a proof. The simulator must also generate correct encryptions of the coin data. To do this, $\mathcal{S}$ looks up the addresses $\mathsf{addr}_{\mathsf{pk},0}$ and $\mathsf{addr}_{\mathsf{pk},1}$ for the specified adversarially-controlled users and uses the associated encryption keys to create the ciphertexts $\mathbf{C}_0$ and $\mathbf{C}_1$. It must also generate the correct signatures, so it picks a random signing key $\mathsf{pk}_{\mathsf{sig}}$ to use with Sig and generates σ, h_0, and h_1. It then transmits $(c_0, \mathsf{sn}_0^{\mathsf{old}}, \mathsf{cm}_0^{\mathsf{new}}, \mathsf{pk}_{\mathsf{sig}}, h_0, \mathbf{C}_0, \pi_{\texttt{POUR}}, \sigma)$ to $\mathsf{addr}_{\mathsf{pk},0}$ and $(c_1, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{pk}_{\mathsf{sig}}, h_1, \mathbf{C}_1, \pi_{\texttt{POUR}}, \sigma)$ to $\mathsf{addr}_{\mathsf{pk},1}$.

We now show that the output of the ideal world experiment is computationally indistinguishable from the output of the real world experiment via a series of hybrids.

1. $\mathbf{H}_0$: This is the real DAP experiment.
2. $\mathbf{H}_1$: In this hybrid, we replace all proofs by honest parties with simulated proofs. By Lemma C.1 we show that if the proof system is computationally simulation sound, then $\mathbf{H}_1 \overset{c}{\approx} \mathbf{H}_0$, that is $\mathbf{H}_1$ is computationally indistinguishable from $\mathbf{H}_0$.
3. $\mathbf{H}_2$: In this hybrid, we run the knowledge extractor when encountering the transactions of any corrupted parties, and abort if the knowledge extractor fails. By Lemma C.2 we show that if the proof extractor fails with negligible probability, then $\mathbf{H}_2 \overset{c}{\approx} \mathbf{H}_1$.
4. $\mathbf{H}_3$: In this hybrid we replace the serial number in Pour Line 1c [3] and in $\mathsf{Receive}$ Line 3b [3] (i.e. the output of $\mathsf{PRF}^{\mathrm{sn}}_{a_{\mathsf{sk}}}$) with a randomly chosen string. When an honest party creates a Pour transaction, the simulator generates random serial numbers for the coins. By Lemma C.3 we show that if $\mathsf{PRF}^{\mathrm{sn}}_{a_{\mathsf{sk}}}$ is a PRF and collision resistant, then $\mathbf{H}_3 \overset{c}{\approx} \mathbf{H}_2$.
5. $\mathbf{H}_4$: In this hybrid we replace the MAC PRFs in Pour Line 4 [3] ($\mathsf{PRF}^{\mathrm{pk}}_{a_{\mathsf{sk}}}$) with randomly chosen values. When an honest party creates a Pour transaction, the simulator generates random values for the MACs. By Lemma C.4 we show that if $\mathsf{PRF}^{\mathrm{pk}}_{a_{\mathsf{sk}}}$ is a PRF, then $\mathbf{H}_4 \overset{c}{\approx} \mathbf{H}_3$.
6. $\mathbf{H}_5$: In this hybrid we replace the commitments (COMM_r and COMM_s) with commitments to random values. By Lemma C.5 we show that if COMM_r and COMM_s are secure commitment schemes, then $\mathbf{H}_5 \overset{c}{\approx} \mathbf{H}_4$.
7. $\mathbf{H}_6$: In this hybrid we replace each ciphertext sent to an honest user Enc with the encryption of a random message under a random public key. By Lemma C.6 we show that if Enc is a key-private encryption scheme, then $\mathbf{H}_6 \overset{c}{\approx} \mathbf{H}_5$.

Note that Hybrid 6 is identical to the simulation. By summation over the previous hybrids we show that Hybrid 6 $\overset{c}{\approx}$ Hybrid 0. We conclude our proof sketch by presenting the supporting lemmas.

Lemma C.1 *For all PPT adversaries $\mathcal{A}$, if simulation sound zkSNARKs exist, then advantage $\mathcal{A}_{\boldsymbol{H}_1} - \mathcal{A}_{\boldsymbol{H}_0} \leq \epsilon$ where ϵ is the simulation failure rate.*

Proof. The simulator operates in the same manner, but we now simulate proofs for honest parties.

By definition, $\pi_{\texttt{MINT}}$ and $\pi_{\texttt{POUR}}$ are both zkSNARKs. From Section 4, we recall that the simulator will fail with at most negligible probability because it deals solely with zkSNARKs, which have efficient simulators. Therefore ϵ is negligible. □

Lemma C.2 *For all PPT adversaries $\mathcal{A}$, if extractable zkSNARKs exist, then advantage $\mathcal{A}_{\boldsymbol{H}_2} - \mathcal{A}_{\boldsymbol{H}_1} \leq 2 \cdot \mathrm{negl}(\lambda)$ where $\mathrm{negl}(\lambda)$ is the extraction failure rate.*

Proof. The simulator operates in the same manner, but we now extract when given the transactions of corrupted parties.

By definition, $\pi_{\texttt{MINT}}$ and $\pi_{\texttt{POUR}}$ are both zkSNARKs. Intuitively, we can see that the extractor will fail with at most negligible probability because it deals solely with zkSNARKs, which have efficient extractors [13]. Therefore, our proofs $\pi_{\texttt{MINT}}$ and $\pi_{\texttt{POUR}}$ have knowledge extractors that succeed with probability $1 - \mathrm{negl}(\lambda)$ for some negligible function $\mathrm{negl}(\cdot)$. □

Lemma C.3 *Advantage $\mathcal{A}_{\boldsymbol{H}_3} - \mathcal{A}_{\boldsymbol{H}_2} \leq \mathrm{negl}(\lambda)$.*

Proof. The simulator operates in the same manner, but we replace the serial numbers (i.e. the output of $\mathsf{PRF}^{\mathrm{sn}}$) with a randomly chosen values.

By definition, $\mathsf{PRF}^{\mathrm{sn}}$ is a collision resistant pseudorandom function [3]. Therefore an adversary can only distinguish the output from random with negligible probability. □

Lemma C.4 *Advantage $\mathcal{A}_{\boldsymbol{H}_4} - \mathcal{A}_{\boldsymbol{H}_3} \leq 2 \cdot \mathrm{negl}(\lambda)$.*

Proof. The simulator operates in the same manner, but now we replace the MACs in Pour Line 4 [3] ($\mathsf{PRF}^{\mathrm{pk}}$) with randomly chosen values.

By definition, $\mathsf{PRF}^{\mathrm{pk}}$ is a pseudorandom function [3]. Therefore an adversary can only distinguish the output from random with negligible probability. □

Lemma C.5 *Advantage $\mathcal{A}_{\boldsymbol{H}_5} - \mathcal{A}_{\boldsymbol{H}_4} \leq \mathrm{negl}(\lambda)$.*

Proof. The simulator continues to operate in the same manner as originally described, but we now replace the commitments (COMM_r and COMM_s) with commitments to randomly chosen values.

By construction COMM is a statistically hiding commitment scheme (see [3] for its construction and a proof thereof) and therefore the probability that an adversary can detect this substitution is negligible. □

Lemma C.6 *Advantage $\mathcal{A}_{\boldsymbol{H}_6} - \mathcal{A}_{\boldsymbol{H}_5} \leq \mathrm{negl}(\lambda)$.*

Proof. We now replace the ciphertexts in Pour with ciphertexts created by random keys.

By definition, the encryption scheme Enc satisfies two security properties: (1) *ciphertext indistinguishability under chosen-ciphertext attack* (IND-CCA security); and (2) *key indistinguishability under chosen-ciphertext attack* (IK-CCA security).

Because Enc is key-private, an adversary can only distinguish between a random ciphertext and one created by an unknown key with negligible probability. □

□