ABSTRACT

Emerging smart contract systems over decentralized cryptocurrencies allow mutually distrustful parties to transact safely with each other without trusting a third-party intermediary. In the event of contractual breaches or aborts, the decentralized blockchain ensures that other honest parties can commensurate remuneration. Existing systems, however, lack transactional privacy. All transactions, including flow of money between pseudonyms and amount transacted, are exposed in the clear on the blockchain.

We present **Hawk**, a decentralized smart contract system that does not store financial transactions in the clear on the blockchain, thus retaining transactional privacy from the public’s view. A Hawk programmer can write a private smart contract in an intuitive manner without having to implement cryptography, and our compiler automatically generates an efficient cryptographic protocol where contractual parties interact with the blockchain, using cryptographic primitives such as succinct zero-knowledge proofs.

To formally define and reason about the security of our protocols, we are the first to formalize the blockchain model of secure computation. The formal modeling is of independent interest. We advocate the community to adopt such a formal model when designing interesting applications atop decentralized blockchains.

1. INTRODUCTION

Decentralized cryptocurrencies such as Bitcoin [45] and altcoins [18] have rapidly gained popularity, and are often quoted as a glimpse into our future [8]. These emerging cryptocurrency systems build atop a novel blockchain technology where miners run distributed consensus whose security is ensured if no adversary wields a large fraction of the computational (or other forms of) resource. The terms “blockchain” and “miners” are therefore often used interchangeably.

Blockchains like Bitcoin reach consensus not only about a stream of data (e.g., payment transactions), but also about computations involving this data (e.g., applying transactional semantics and updating the ledger). When we generalize the blockchain’s computation to arbitrary Turing-complete logic, we obtain an expressive smart contract system such as the soon-to-be-launched Ethereum [52]. Smart contracts are programs executed by all miners. Assuming that the decentralized consensus protocol is secure, smart contracts can be thought of as a conceptual party (in reality decentralized) that can be trusted for correctness but not for privacy. Specifically, a blockchain’s data and computation are publicly visible. Further, since the blockchain advances in well-defined block intervals defined by the “mining” process, a discrete notion of time exists.

These features combined make the blockchain a new computation model empowered to enforce notions of financial fairness even in the presence of aborts. As is well-known in the cryptography literature, fairness against aborts is in general impossible in standard models of interactive protocol execution (e.g., secure multi-party computation), when the majority of parties can be corrupted [5][15][23]. In the blockchain model, protocol aborts can be made evident by timeouts. Therefore, the blockchain can enforce financial remuneration to the honest counterparties in the presence of protocol breach or aborts. In summary, decentralized smart contracts allow parties mutually unbeknownst to transact securely with each other, without trusting any central intermediary or incurring high legal and transactional cost.

Despite the expressiveness and power of blockchain and smart contracts, the present form of these technologies lacks transactional privacy. The entire sequence of actions taken in a smart contract are propagated across the network and/or recorded on the blockchain, and therefore are publicly visible. Even though parties can create new pseudonymous public keys to increase their anonymity, the values of all transactions and balances for each (pseudonymous) public key are publicly visible. Further, recent works have also demonstrated deanonymization attacks by analyzing the transactional graph structures of cryptocurrencies [40][48]. The privacy requirements of many financial transactions will likely preclude the use of existing smart contract systems. Although there has been progress in designing privacy-preserving cryptocurrencies such as Zerocash [10] and several others [24][41][50], these systems forgo programmability, and it is unclear a priori how to enable programmability without exposing transactions and data in cleartext to miners.

1.1 Hawk Overview

We propose Hawk, a framework for building privacy-preserving smart contracts. With Hawk, a non-specialist programmer can easily write a Hawk program without having to implement any cryptography. Our Hawk compiler automatically compiles the program to a cryptographic protocol between the blockchain and the users. As shown in Figure 1, a Hawk program contains two parts:

- A private contract program denoted \( \phi_{\text{priv}} \) which takes in parties’ input data (e.g., choice in a “rock, paper, scissors” game) as well as currency units (e.g., bids in an auction).
The program $\phi_{\text{priv}}$ performs computation to determine the payout distribution amongst the parties. For example, in an auction, winner’s bid goes to the seller, and others’ bids are refunded. The private contract $\phi_{\text{priv}}$ is meant to protect the participants’ data and the exchange of money.

- A public contract program denoted $\phi_{\text{pub}}$ that does not touch private data or money.

Our Hawk compiler will compile the public contract to execute directly on the public blockchain, and compile the private contract $\phi_{\text{priv}}$ into a cryptographic protocol involving the following pieces: i) protocol logic to be executed by the blockchain; and ii) protocol logic to be executed by contractual parties.

Security guarantees. Hawk’s security guarantees encompass two aspects:

- On-chain privacy. On-chain privacy stipulates that transactional privacy be provided against the public (i.e., against any party not involved in the contract) – unless the contractual parties themselves voluntarily disclose information. Although in Hawk protocols, users exchange data with the blockchain, and rely on it to ensure fairness against aborts, the flow of money and amount transacted in the private contract $\phi_{\text{priv}}$ is cryptographically hidden from the public’s view.

- Contractual security. While on-chain privacy protects contractual parties’ privacy against the public (i.e., parties not involved in the contract), contractual security protects parties in the same contractual agreement from each other. Hawk assumes that contractual parties act selfishly to maximize their own financial interest. In particular, they can arbitrarily deviate from the prescribed protocol or even abort prematurely. Therefore, contractual security is a multi-faceted notion that encompasses not only cryptographic notions of confidentiality and authenticity, but also financial fairness in the presence of cheating and aborting behavior. The best way to understand contractual security is through a concrete example, and we refer the reader to Section 4.2 for a more detailed explanation.

Minimally trusted manager. Hawk-generated protocols assume a special contractual party called a manager (e.g., an auction manager) besides the normal users. The manager aids the efficient execution of our cryptographic protocols while being minimally trusted. A skeptical reader may worry that our use of such a party trivializes our security guarantees; we dispel this notion by directly explaining what a corrupt manager can and cannot do: Although a manager can see the transactions that take place in a private contract, it cannot affect the outcome of the contract, even when it colludes with other users. In the event that a manager aborts the protocol, it can be financially penalized, and users obtain remuneration accordingly. The manager also need not be trusted to maintain the security or privacy of the underlying currency (e.g., it cannot double-spend, inflate the currency, or deanonymize users). Furthermore, if multiple contract instances run concurrently, each contract may specify a different manager and the effects of a corrupt manager are confined to that instance. Finally, the manager role may be instantiated with trusted computing hardware like Intel SGX, or replaced with a multiparty computation among the users themselves, as we describe in Section 4.3.

1.2 Example: Sealed Auction

Example program. Figure 2 shows a Hawk program for implementing a sealed, second-price auction where the highest bidder wins, but pays the second highest price. Second-price auctions are known to incentivize truthful bidding under certain assumptions, and it is important that bidders submit bids without knowing the bid of the other people. Our example auction program contains a private contract $\phi_{\text{priv}}$ that determines the winning bidder and the price to be paid; and a public contract $\phi_{\text{pub}}$ that relies on public deposits to protect bidders from an aborting manager.

Contractual security requirements. Hawk will compile this auction program to a cryptographic protocol. As mentioned earlier, as long as the bidders and the manager do not voluntarily disclose information, transaction privacy is maintained against the public. Hawk also guarantees the following contractual security requirements for parties in the contract:

- Input independent privacy. Each user does not see others’ bids before committing to their own. This way, users bids are independent of others’ bids. Hawk guarantees input independent privacy even against a malicious manager.

- Posterior privacy. As long as the manager does not disclose information, users’ bids are kept private from each other (and from the public) even after the auction.

- Financial fairness. If a party aborts or if the auction manager aborts, the aborting party should be financially penalized while the remaining parties receive compensation. Such fairness guarantees are not attainable in general by off-chain only protocols such as secure multi-party computation. As explained later, Hawk offers built-in mechanisms for enforcing refunds of private bids after certain timeouts. Hawk also allows the programmer to define additional rules, as part of the Hawk contract, that govern financial fairness.

- Security against a dishonest manager. We ensure authenticity against a dishonest manager: besides aborting, a dishonest manager cannot affect the outcome of the auction and the redistribution of money, even when it colludes with a subset of the users. We stress that to ensure the above, input independent privacy against a faulty manager is a prerequisite. Moreover, if the manager aborts, it can be financially penalized, and the participants obtain corresponding remuneration.

An auction with the above security and privacy requirements cannot be trivially implemented atop existing cryptocurrency systems such as Ethereum or Zerocash. The former allows for programmability but does not guaran-
specifically, in our sealed auction program, distributes the manager’s public deposit if it aborts. More

\[ \text{Tax} \quad \text{is declared where} \quad \text{outs} \quad \text{parameters using the} \quad \text{HawkDeclareTimeouts} \quad \text{Aborting are dealt with using} \quad \text{Aborting and timeouts.} \]

... program such as Figure 2 declares time-
outs. A function will be invoked when the Hawk contract completes execution within \( T_3 \), i.e., manager did not abort. Otherwise, if the Hawk contract does not complete execution within \( T_3 \), the managerTimeOut function will be invoked. We remark that although not explicitly written in the code, all Hawk contracts have an implicit default entry point for accepting parties’ deposits – these deposits are withheld by the contract till they are redistributed by the contract. Bidders should check that the manager has made a public deposit before submitting their bids.

Additional applications. Besides the sealed auction example, Hawk supports various other applications. We give more sample programs in Section 5.2.

1.3 Contributions

To the best of our knowledge, Hawk is the first to simultaneously offer transactional privacy and programmability in a decentralized cryptocurrency system.

The blockchain UC formalization could be presented on its own, but we gain evidence of its usefulness by implementing it and applying it to interesting practical examples. Likewise, our system implementation benefits from the formalism because we can use our framework to provide provable security.

Formal models for decentralized smart contracts. We are among the first ones to initiate a formal, academic treatment of the blockchain model of cryptography. We present a formal, UC-compatible model for the blockchain model of cryptography – this formal model is of independent interest, and can be useful in general for defining and modeling the security of protocols in the blockchain model. Our formal model has also been adopted by the Gyges work \(^3\) in designing criminal smart contracts.

In defining for formal blockchain model, we rely on a notion called wrappers to modularize our protocol design and to simplify presentation. Wrappers handle a set of common details such as timers, pseudonyms, global ledgers in a centralized place such that they need not be repeated in every protocol.

New cryptography suite. We implement a new cryptography suite that binds private transactions with programmable logic. Our protocol suite contains three essential primitives freeze, compute, and finalize. The freeze primitive allows parties to commit to not only normal data, but also coins. Committed coins are frozen in the contract, and the payout distribution will later be determined by the program \( \phi_{\text{priv}} \). During compute, parties open their committed data and currency to the manager, such that the manager can compute the function \( \phi_{\text{priv}} \). Based on the outcome of \( \phi_{\text{priv}} \), the manager now constructs new private coins to be paid to each recipient. When the manager submits both the new private coins as well as zero-knowledge proofs of their well-formedness, the previously frozen coins are now redistributed among the users. Our protocol suite strictly generalizes Zerocash since Zerocash implements only private money transfers between users without programmability.

We define the security of our primitives using ideal functionalities, and formally prove security of our constructions under a simulation-based paradigm.

Implementation and evaluation. We built a Hawk prototype and evaluated its performance by implementing sev-

```
1 HawkDeclareParties(Seller,/* N parties */);
2 HawkDeclareTimeouts(/* hardcoded timeouts */);

3 // Private contract \( \phi_{\text{priv}} \)
4 private contract auction(inp &in, Outp &out) {
5    int winner = -1;
6    int bestprice = -1;
7    int secondprice = -1;
8    for (int i = 0; i < N; i++) {
9        if (in.party[i].$val > bestprice) {
10            secondprice = bestprice;
11            bestprice = in.party[i].$val;
12            winner = i;
13        } else if (in.party[i].$val > secondprice) {
14            secondprice = in.party[i].$val;
15        }
16    }
17    // Winner pays secondprice to seller
18    // Everyone else is refunded
19    out.Seller.$val = secondprice;
20    out.party[winner].$val = bestprice - secondprice;
21    out.winner = winner;
22    for (int i = 0; i < N; i++) {
23        if (i != winner)
24            out.party[i].$val = in.party[i].$val;
25    }
26 }

27 // Public contract \( \phi_{\text{pub}} \)
28 public contract deposit {} // Manager deposits $N
29    def check():
30        send $N to Manager
31        def managerTimeOut();
32            for (i in range($N)):
33                send $1 to party[i]
34    }
```

Figure 2: Hawk contract for a second-price sealed auction. Code described in this paper is an approximation of our real implementation. In the public contract, the syntax “send $N to \( P \)” corresponds to the following semantics in our cryptographic formalism: \( \text{ledger}[P] := \text{ledger}[P] + \$N \) – see Section 2. In meaning, the latter guarantees transactional privacy but at the price of even reduced programmability than Bitcoin.

Aborting and timeouts. Aborting is dealt with using timeouts. A Hawk program such as Figure 2 declares timeout parameters using the HawkDeclareTimeouts special syntax. Three timeouts are declared where \( T_1 < T_2 < T_3 \):

\( T_1 \) : The Hawk contract stops collecting bids after \( T_1 \).

\( T_2 \) : All users should have opened their bids to the manager within \( T_2 \); if a user submitted a bid but fails to open by \( T_2 \), its input bid is treated as 0 (and any other potential input data treated as \( \perp \)), such that the manager can continue.

\( T_3 \) : If the manager aborts, users can reclaim their private bids after time \( T_3 \).

The public Hawk contract \( \phi_{\text{pub}} \) can additionally implement incentive structures. Our sealed auction program redistributes the manager’s public deposit if it aborts. More specifically, in our sealed auction program, \( \phi_{\text{pub}} \) defines two functions, namely check and managerTimeOut. The check
eral example applications, including a sealed-bid auction, a “rock, paper, scissors” game, a crowdfunding application, and a swap financial instrument. We demonstrate interesting protocol optimizations that gained us a factor of $18 \times$ in performance relative to a straightforward implementation. We show that for at about 100 parties (e.g., auction and crowdfunding), the manager’s cryptographic computation (the most expensive part of the protocol) is under 2.8min using 4 cores, translating to under $0.13 of EC2 time. Further, all on-chain computation (performed by all miners) is very cheap, and under 17ms for all cases. We will open source our Hawk framework in the near future.

1.4 Background and Related Work

1.4.1 Background

The original Bitcoin offers limited programmability through a scripting language that is neither Turing-complete nor user-friendly. Numerous previous endeavors at creating smart contract-like applications atop Bitcoin (e.g., lottery [7, 15], micropayments [4], verifiable computation [25]) have demonstrated the difficulty of in retrofitting Bitcoin’s scripting language – this serves well to motivate a Turing-complete, user-friendly smart contract language.

Ethereum is the first Turing-complete decentralized smart contract system. With Ethereum’s imminent launch, companies and hobbyists are already building numerous smart contract applications either atop Ethereum or by forking off Ethereum, such as prediction markets [3], supply chain provenance [6], crowd-based fundraising [1], and security and derivatives trading [27].

Security of the blockchain. Like earlier works that design smart contract applications for cryptocurrencies, we rely on the underlying decentralized blockchain to be secure. Therefore, we assume the blockchain’s consensus protocol attains security when an adversary does not wield a large fraction of the computational power. Existing cryptocurrencies are designed with heuristic security. On one hand, researchers have identified attacks on various aspects of the system [28, 33], on the other, efforts to formally understand the security of blockchain consensus have begun [32, 43].

Minimizing on-chain costs. Since every miner will execute the smart contract programs while verifying each transaction, cryptocurrencies including Bitcoin and Ethereum collect transaction fees that roughly correlate with the cost of execution. While we do not explicitly model such fees, we design our protocols to minimize on-chain costs by performing most of the heavy-weight computation off-chain.

1.4.2 Additional Related Works

Leveraging blockchain for financial fairness. A few prior works have explored how to leverage the blockchain technology to achieve fairness in protocol design. For example, Bentov et al. [16], Andrychowicz et al. [7], Kumaresan et al. [28], Kiayias et al. [36], as well as Zyskind et al. [54], show how Bitcoin can be used to ensure fairness in secure multi-party computation protocols. These protocols also perform off-chain secure computation of various types, but do not guarantee transactional privacy (i.e., hiding the currency flows and amounts transacted). For example, it is not clear how to implement our sealed auction example using these earlier techniques. Second, these earlier works either do not offer system implementations or provide implementations only for specific applications (e.g., lottery). In comparison, Hawk provides a generic platform such that non-specialist programmers can easily develop privacy-preserving smart contracts.

Smart contracts. The conceptual idea of programmable electronic “smart contracts” dates back nearly twenty years [49]. Besides recent decentralized cryptocurrencies, which guarantee authenticity but not privacy, other smart contract implementations rely on trusted servers for security [44]. Our work therefore comes closest to realizing the original vision of parties interacting with a trustworthy “virtual computer” that executes programs involving money and data.

Programming frameworks for cryptography. Several works have developed programming frameworks that take in high-level program as specifications and generate cryptographic implementations. For example, previous works have developed compilers for secure multi-party computation [17, 37, 39, 17], authenticated data structures [42], and (zero-knowledge) proofs [12, 30, 31, 46]. Zheng et al. show how to generate secure distributed protocols such as sealed auctions, battleship games, and banking applications [53]. These works support various notions of security, but none of them interact directly with money or leverage public blockchains for ensuring financial fairness. Thus our work is among the first to combine the “correct-by-construction” cryptography approach with smart contracts.

Concurrent work. Our framework is the first to provide a full-fledged formal model for decentralized blockchains as embodied by Bitcoin, Ethereum, and many other popular decentralized cryptocurrencies. In concurrent and independent work, Kiayias et al. [36] also propose a blockchain model in the (Generalized) Universal Composability framework [31] and use it to derive results that are similar to what we describe in Appendix C.1, i.e., fair MPC with public deposits. However, the “programmability” of their formalism is limited to their specific application (i.e., fair MPC with public deposits). In comparison, our formalism is designed with much broader goals, i.e., to facilitate protocol designers to design a rich class of protocols in the blockchain model. In particular, both our real-world wrapper (Figure 4) and ideal-world wrapper (Figure 3) model the presence of arbitrary user defined contract programs, which interact with both parties and the ledger. Our formalism has also been adopted by the Gyges work [35] demonstrating its broad usefulness.

2. THE BLOCKCHAIN MODEL OF CRYPTOGRAPHY

We describe a formal model for smart contract execution under which we give formal proofs of security for our protocols. Our model conforms to the Universal Composability (UC) framework [19].

Protocols in the blockchain model. Our model makes use of a special party (in reality decentralized) called the contract that is entrusted to enforce correctness but not privacy. All messages sent to the contract and all

1Disambiguation: the contract C in our cryptography formalism always refers to the native contract executed on the blockchain, not the user-facing Hawk program.
of its internal states are publicly visible. During the protocol, users interact with the contract by exchanging messages (also referred to as transactions). Money can be expressed as special bits stored on the blockchain interpreted as a ledger. Our contracts can access and update the ledger to implement money transfers between users (as represented by their pseudonymous public keys).

First, our model allows us to easily capture the time and pseudonym features of cryptocurrencies. In cryptocurrencies such as Bitcoin and Ethereum, time progresses in block intervals, and a smart contract program is allowed to query the current time, and make decisions accordingly, e.g., make a refund operation after a timeout. Second, our model captures the role of a smart contract as a party trusted for correctness but not for privacy. Third, our formalism modularizes our notations by factoring out common specifics related to the smart contract execution model, and implementing these in central wrappers.

In a real-life cryptocurrency system such as Bitcoin or Ethereum, users can make up any number of identities by generating new public keys. In our formal model, for simplicity, we assume that there can be any number of identities in the system, and that they are fixed a priori. It is easy to extend our model to capture registration of new identities dynamically. As mentioned later, we allow each identity to generate an arbitrary (polynomial) number of pseudonyms.

### 2.1 Programs, Functionalities, and Wrappers

To make our notations simple for writing ideal functionalities and smart contracts, we make a conscious notational choice of introducing wrapper functionalities. Wrapper functionalities implement in a central place a set of common features (e.g., timer, ledger, pseudonyms) that are applicable to all ideal functionalities and contracts in our smart contract model of execution. In this way, we can modularize our notational system such that these common and tedious details need not be repeated in writing ideal functionalities and contract programs.

In particular, we will have two types of wrapper functionalities:

- **Contract functionality wrapper** \( G \): A contract functionality wrapper \( G(C) \) takes in a contract program denoted \( C \), and produces a contract functionality. Our real-world protocols will be defined in the \( G(C) \)-hybrid world. Our contract functionality wrapper is formally presented in Figure 4.

Like the ideal functionality wrapper \( F() \), the contract wrapper also implements standard features such as a timer, a global ledger, and money transfers between parties. In addition, the contract wrapper also models the specifics of the contract program. We point out the following important facts about the \( G() \) wrapper:

- **Trusted for correctness but not privacy.** The contract functionality wrapper \( G() \) stipulates that a smart contract is trusted for correctness but not for privacy. In particular, the contract wrapper exposes the contract’s internal state to any party that makes a query.

- **Time and batched processing of messages.** In popular decentralized cryptocurrencies such as Bitcoin and Ethereum, time progresses in block intervals marked by the creation of each new block. Intuitively, our \( G() \) wrapper captures the following fact. In each round (i.e., block interval), the smart contract program may receive multiple messages (also referred to as transactions in the cryptocurrency literature). The order of processing these transactions is determined by the miner who mines the next block. In our model, we allow the adversary to specify an ordering of the messages collected in a round, and our contract program will then process the messages in this adversary-specified ordering.

  - **Rushing adversary.** The contract wrapper \( G() \) captures a rushing adversary. Specifically, the adversary can first see all messages sent to the contract by honest parties, and then decide its own messages for this round, as well as an ordering in which the contract should process the messages in the next round. Modeling a rushing adversary is important, since it captures a class of well-known front-running attacks, e.g., those that exploit transaction malleability [10][20]. For example, in a “rock, paper, scissors” game, if inputs are sent in the clear, an adversary can decide its input based on the other party’s input. An adversary can also try to maul transactions submitted by honest parties to potentially redirect payments to itself. Since our model captures a rushing adversary, we can write ideal functionalities that preclude such front-running attacks.

#### Ideal functionality wrapper \( F \)

An ideal functionality \( F(\text{idealP}) \) takes in an ideal program denoted \( \text{idealP} \). Specifically, the wrapper \( F() \) part defines standard features such as time, pseudonyms, a public ledger, and money transfers between parties. Our ideal functionality wrapper is formally presented in Figure 4.

#### Protocol wrapper II

Our protocol wrapper allows us to modularize the presentation of user protocols. Our protocol wrapper is formally presented in Figure 5.

**Terminology.** For disambiguation, we always refer to the user-defined portions as *programs*. Programs alone do not have complete formal meanings. However, when programs are wrapped with functionality wrappers (including \( F(\cdot) \) and \( G(\cdot) \)), we obtain functionalities with well-defined formal meanings. Programs can also be wrapped by a protocol wrapper to obtain a full protocol with formal meanings.

### 2.2 Modeling Time

At a high level, we express time in a way that conforms to the Universal Composability framework [19]. In the ideal world execution, time is explicitly encoded by a variable \( T \) in an ideal functionality \( F(\text{idealP}) \). In the real world execution, time is explicitly encoded by a variable \( T \) in our contract functionality \( G(C) \). Time progresses in rounds. The environment \( \mathcal{E} \) has the choice of when to advance the timer.

We assume the following convention: to advance the timer, the environment \( \mathcal{E} \) sends a “tick” message to all honest parties. Honest parties’ protocols would then forward this message to \( F(\text{idealP}) \) in the ideal-world execution, or to the \( G(C) \) functionality in the real-world execution. On collecting “tick” messages from all honest parties, the \( F(\text{idealP}) \) or \( G(C) \) functionality would then advance the time \( T := T + 1 \). The functionality also allows parties to query the current time \( T \).

As mentioned earlier, when multiple messages arrive at the blockchain in a time interval, we allow the adversary...
Given an ideal program denoted idealP, the $\mathcal{F}$(idealP) functionality is defined as below:

**Init**: Upon initialization, perform the following:
- **Time**: Set current time $T := 0$. Set the receive queue $rqueue := \emptyset$.
- **Pseudonyms**: Set $\text{nyms} := \{(P_1, P_1), \ldots, (P_N, P_N)\}$, i.e., initially every party’s true identity is recorded as a default pseudonym for the party.
- **Ledger**: A ledger dictionary structure $\text{ledger}[P]$ stores the endowed account balance for each identity $P \in \{P_1, \ldots, P_N\}$. Before any new pseudonyms are generated, only true identities have endowed account balances. Send the array $\text{ledger}[]$ to the ideal adversary $S$.

**idealP.Init**: Run the **Init** procedure of the idealP program.

**Tick**: Upon receiving tick from an honest party $P$: notify $S$ of $(\text{tick}, P)$. If the functionality has collected $\text{tick}$ confirmations from all honest parties since the last clock tick, then
- Call the **Timer** procedure of the idealP program.
- Apply the adversarial permutation $\text{perm}$ to the $rqueue$ to reorder the messages received in the previous round.
- For each $(m, \bar{P}) \in rqueue$ in the permuted order, invoke the delayed actions (in gray background) defined by ideal program idealP at the activation point named “Upon receiving message $m$ from pseudonym $\bar{P}$”. Notice that the program idealP speaks of pseudonyms instead of party identifiers. Set $rqueue := \emptyset$.
- Set $T := T + 1$

**Other activations**: Upon receiving a message of the form $(m, \bar{P})$ from a party $P$:
- Assert that $(\bar{P}, P) \in \text{nyms}$.
- Invoke the immediate actions defined by ideal program idealP at the activation point named “Upon receiving message $m$ from pseudonym $\bar{P}$”.
- Queue the message by calling $rqueue.add(m, \bar{P})$.

**Permute**: Upon receiving $(\text{permute}, \text{perm})$ from the adversary $S$, record $\text{perm}$.

**GetTime**: On receiving $\text{gettime}$ from a party $P$, notify the adversary $S$ of $(\text{gettime}, P)$, and return the current time $T$ to party $P$.

**GenNym**: Upon receiving $\text{gennym}$ from an honest party $P$:
- Notify the adversary $S$ of $\text{gennym}$. Wait for $S$ to respond with a new nym $\bar{P}$ such that $\bar{P} \notin \text{nyms}$. Now, let $\text{nyms} := \text{nyms} \cup \{(P, \bar{P})\}$, and send $\bar{P}$ to $P$. Upon receiving $(\text{gennym}, \bar{P})$ from a corrupted party $P$: if $\bar{P} \notin \text{nyms}$, let $\bar{P} := \text{nyms} \cup \{(P, P)\}$.

**Ledger operations**: // inner activation

**Transfer**: Upon receiving $(\text{transfer}, \text{amount}, \bar{P}_r)$ from some pseudonym $\bar{P}_s$:
- Notify $(\text{transfer}, \text{amount}, \bar{P}_r, \bar{P}_s)$ to the ideal adversary $S$.
  - Assert that $\text{ledger}[\bar{P}_s] \geq \text{amount}$.
  - $\text{ledger}[\bar{P}_s] := \text{ledger}[\bar{P}_s] - \text{amount}$
  - $\text{ledger}[\bar{P}_r] := \text{ledger}[\bar{P}_r] + \text{amount}$

/* $\bar{P}_s, \bar{P}_r$ can be pseudonyms or true identities. Note that each party’s identity is a default pseudonym for the party. */

**Expose**: On receiving $\text{exposeledger}$ from a party $P$, return $\text{ledger}$ to the party $P$.

Figure 3: The $\mathcal{F}$(idealP) functionality is parameterized by an ideal program denoted idealP. An ideal program idealP can specify two types of activation points, immediate activations and delayed activations. Activation points are invoked upon recipient of messages. Immediate activations are processed immediately, whereas delayed activations are collected and batch processed in the next round. The $\mathcal{F}()$ wrapper allows the ideal adversary $S$ to specify an order $\text{perm}$ in which the messages should be processed in the next round. For each delayed activation, we use the leak notation in an ideal program idealP to define the leakage which is *immediately* exposed to the ideal adversary $S$ upon recipient of the message.
The $G(C)$ functionality

Given a contract program denoted $C$, the $G(C)$ functionality is defined as below:

**Init:** Upon initialization, perform the following:

- A ledger data structure $\text{ledger}[P]$ stores the account balance of party $P$. Send the entire balance $\text{ledger}$ to $A$.
- Set current time $T := 0$. Set the receive queue $\text{rqueue} := \emptyset$.
- Run the **Init** procedure of the $C$ program.
- Send the $C$ program’s internal state to the adversary $A$.

**Tick:** Upon receiving $\text{tick}$ from an honest party, if the functionality has collected $\text{tick}$ confirmations from all honest parties since the last clock tick, then

- Apply the adversarial permutation $\text{perm}$ to the $\text{rqueue}$ to reorder the messages received in the previous round.
- Call the **Timer** procedure of the $C$ program.
- Pass the reordered messages to the $C$ program to be processed. Set $\text{rqueue} := \emptyset$.
- Set $T := T + 1$

**Other activations:**

- **Authenticated receive:** Upon receiving a message ($\text{authenticated}, m$) from party $P$:
  - Send $(m, P)$ to the adversary $A$.
  - Queue the message by calling $\text{rqueue}.add(m, P)$.
- **Pseudonymous receive:** Upon receiving a message of the form ($\text{pseudonymous}, m, \bar{P}, \sigma$) from any party:
  - Send $(m, \bar{P}, \sigma)$ to the adversary $A$.
  - Parse $\sigma := (\text{nonce}, \sigma')$, and assert $\text{Verify}(\bar{P}.\text{spk}, (\text{nonce}, T, \bar{P}.\text{epk}, m), \sigma') = 1$
  - If message ($\text{pseudonymous}, m, \bar{P}, \sigma$) has not been received earlier in this round, queue the message by calling $\text{rqueue}.add(m, P)$.
- **Anonymous receive:** Upon receiving a message ($\text{anonymous}, m$) from party $P$:
  - Send $m$ to the adversary $A$.
  - If $m$ has not been seen before in this round, queue the message by calling $\text{rqueue}.add(m)$.

**Permute:** Upon receiving ($\text{permute}, \text{perm}$) from the adversary $A$, record $\text{perm}$.

**Expose:** On receiving $\text{exposestate}$ from a party $P$, return the functionality’s internal state to the party $P$. Note that this also implies that a party can query the functionality for the current time $T$.

**Ledger operations:** // inner activation

**Transfer:** Upon recipient of ($\text{transfer}, \text{amount}, \bar{P}_r$) from some pseudonym $\bar{P}_s$:

- Assert $\text{ledger}[\bar{P}_s] \geq \text{amount}$
- $\text{ledger}[\bar{P}_s] := \text{ledger}[\bar{P}_s] - \text{amount}$
- $\text{ledger}[\bar{P}_r] := \text{ledger}[\bar{P}_r] + \text{amount}$

**Figure 4:** The $G(C)$ functionality is parameterized by a contract program denoted $C$. The $G(\cdot)$ wrapper mainly performs the following: *i)* exposes all of its internal states and messages received to the adversary; *ii)* makes the functionality time-aware: messages received in one round and queued and processed in the next round. The $G(\cdot)$ wrapper allows the adversary to specify an ordering to the messages received by the contract in one round.
Π(prot) protocol wrapper in the $G(C)$-hybrid world

Given a party’s local program denoted prot, the Π(prot) functionality is defined as below:

**Pseudonym related:**

- **GenNym:** Upon receiving input gennym from the environment $E$, generate $(epk,esk) \leftarrow \text{Keygen}_{\text{enc}}(1^\lambda)$, and $(spk,ssk) \leftarrow \text{Keygen}_{\text{sign}}(1^\lambda)$. Call $\text{payload} := \text{prot.GenNym}(1^\lambda,(epk,spk))$. Store $\text{nyms} := \text{nyms} \cup \{(epk,spk,\text{payload})\}$, and output $(epk,spk,\text{payload})$ as a new pseudonym.

- **Send:** Upon receiving internal call $(\text{send}, m, P)$:
  
  If $\bar{P} == P$: send $(\text{authenticated}, m)$ to $G(C)$.  // this is an authenticated send
  
  Else, // this is a pseudonymous send
  
  Assert that pseudonym $\bar{P}$ has been recorded in nyms;
  
  Query current time $T$ from $G(C)$. Compute $\sigma' := \text{Sign}(ssk,(\text{nonce},T, epk,m))$ where $ssk$ is the recorded secret signing key corresponding to $\bar{P}$, $\text{nonce}$ is a freshly generated random string, and $epk$ is the recorded public encryption key corresponding to $P$. Let $\sigma := (\text{nonce},\sigma')$.

  Send $(\text{pseudonymous}, m, \bar{P},\sigma)$ to $G(C)$.

- **AnonSend:** Upon receiving internal call $(\text{anonsend}, m, \bar{P})$: send $(\text{anonymous}, m)$ to $G(C)$.

**Timer and ledger transfers:**

- **Transfer:** Upon receiving input $(\text{transfer}, \$\text{amount}, P_r, P)$ from the environment $E$:
  
  Assert that $\bar{P}$ is a previously generated pseudonym.

  Send $(\text{transfer}, \$\text{amount}, P_r)$ to $G(C)$ as pseudonym $\bar{P}$.

- **Tick:** Upon receiving tick from the environment $E$, forward the message to $G(C)$.

**Other activations:**

- **Act as pseudonym:** Upon receiving any input of the form $(m,\bar{P})$ from the environment $E$:

  Assert that $\bar{P}$ was a previously generated pseudonym.

  Pass $(m,\bar{P})$ the party’s local program to process.

- **Others:** Upon receiving any other input from the environment $E$, or any other message from a party: Pass the input/message to the party’s local program to process.

Figure 5: Protocol wrapper.
to choose a permutation to specify the order in which the blockchain will process the messages. This captures potential network attacks such as delaying message propagation, and front-running attacks (a.k.a. rushing attacks) where an adversary determines its own message after seeing what other parties send in a round.

2.3 Modeling Pseudonyms

We model a notion of “pseudonymity” that provides a form of privacy, similar to that provided by typical cryptocurrencies such as Bitcoin. Any user can generate an arbitrary (polynomially-bounded) number of pseudonyms, and each pseudonym is “owned” by the party who generated it. The correspondence of pseudonyms to real identities is hidden from the adversary.

Effectively, a pseudonym is a public key for a digital signature scheme, the corresponding private key to which is known by the party who “owns” the pseudonym. The public contract functionality allows parties to publish authenticated messages that are bound to a pseudonym of their choice. Thus each interaction with the public contract is, in general, associated with a pseudonym but not to a user’s real identity.

We abstract away the details of pseudonym management by implementing them in our wrappers. This allows user-defined applications to be written very simply, as though using ordinary identities, while enjoying the privacy benefits of pseudonymity.

Our wrapper provides a user-defined hook, “gennym”, that is invoked each time a party creates a pseudonym. This allows the application to define an additional per-pseudonym payload, such as application-specific public keys. From the point-of-view of the application, this is simply an initialization subroutine invoked once for each participant.

Our wrapper provides several means for users to communicate with a contract. The most common way is for a user to publish an authenticated message associated with one of their pseudonyms, as described above. Additionally, “anon-send” allows a user to publish a message without reference to any pseudonym at all.

In spite of pseudonymity, it is sometimes desirable to assign a particular user to a specific role in a contract (e.g., “auction manager”). The alternative is to assign roles on sending “transfer” messages to the public contract (as with funds to each other (or among their own pseudonyms) by using a party identifier as a parameter or hardcoded string.

We abstract away the details of pseudonym management by implementing them in our wrappers. This allows user-defined applications to be written very simply, as though using ordinary identities, while enjoying the privacy benefits of pseudonymity.

Our wrapper provides a user-defined hook, “gennym”, that is invoked each time a party creates a pseudonym. This allows the application to define an additional per-pseudonym payload, such as application-specific public keys. From the point-of-view of the application, this is simply an initialization subroutine invoked once for each participant.

Our wrapper provides several means for users to communicate with a contract. The most common way is for a user to publish an authenticated message associated with one of their pseudonyms, as described above. Additionally, “anon-send” allows a user to publish a message without reference to any pseudonym at all.

In spite of pseudonymity, it is sometimes desirable to assign a particular user to a specific role in a contract (e.g., “auction manager”). The alternative is to assign roles on sending “transfer” messages to the public contract (as with funds to each other (or among their own pseudonyms) by using a party identifier as a parameter or hardcoded string.

We abstract away the details of pseudonym management by implementing them in our wrappers. This allows user-defined applications to be written very simply, as though using ordinary identities, while enjoying the privacy benefits of pseudonymity.

Our wrapper provides a user-defined hook, “gennym”, that is invoked each time a party creates a pseudonym. This allows the application to define an additional per-pseudonym payload, such as application-specific public keys. From the point-of-view of the application, this is simply an initialization subroutine invoked once for each participant.

Our wrapper provides several means for users to communicate with a contract. The most common way is for a user to publish an authenticated message associated with one of their pseudonyms, as described above. Additionally, “anon-send” allows a user to publish a message without reference to any pseudonym at all.

In spite of pseudonymity, it is sometimes desirable to assign a particular user to a specific role in a contract (e.g., “auction manager”). The alternative is to assign roles on sending “transfer” messages to the public contract (as with funds to each other (or among their own pseudonyms) by using a party identifier as a parameter or hardcoded string.

2.4 Modeling Money

We model money as a public ledger, which associates quantities of money to pseudonyms. Users can transfer funds to each other (or among their own pseudonyms) by sending “transfer” messages to the public contract (as with other messages, these are delayed until the next round and may be delivered in any order). The ledger state is public knowledge, and can be queried immediately using the “exposeledger” instruction.

There are many conceivable policies for introducing new currency into such a system: for example, Bitcoin “mints” new currency as a reward for each miner who solves a proof-of-work puzzle. We take a simple approach of defining an arbitrary, publicly visible (i.e., common knowledge) initial allocation that associates a quantity of money to each party’s real identity. Except for this initial allocation, no money is created or destroyed.

2.5 Conventions for Writing Programs

Thanks to our wrapper-based modularized notational system, the ideal program and the contract program are the main locations where user-supplied, custom program logic is defined. We use the following conventions for writing the ideal program and the contract program.

Timer activation points. Every time $F(\text{idealP})$ or $G(C)$ advances the timer, it will invoke a `Timer` interrupt call. Therefore, by convention, we allow the ideal program or the contract program can define a `Timer` activation point. Timer operations (e.g., refunding money after a certain timeout) can be implemented under the `Timer` activation point.

Delayed processing in ideal programs. When writing the contract program, every message received by the contract program is already delayed by a round due to the $G(\cdot)$ wrapper.

When writing the ideal program, we introduce a simple convention to denote delayed computation. Program instructions that are written in gray background denote computation that does not take place immediately, but is deferred to the beginning of the next timer click. This is a convenient shorthand because in our real-world protocol, effectively any computation done by a contract functionality will be delayed. For example, in our `IdealCash` ideal program (see Figure 7), whenever the ideal functionality receives a `mint` or `pour` message, the ideal adversary $S$ is notified immediately; however, processing of the messages is deferred till the next timer click. Formally, delayed processing can be implemented simply by storing state and invoking the delayed program instructions on the next `Timer` call.

To avoid ambiguity, we assume that by convention, the delayed instructions are invoked at the beginning of the `Timer` call. In other words, upon the next timer click, the delayed instructions are executed first.

Pseudonymity. All party identifiers that appear in ideal programs, contract programs, and user-side programs by default refer to pseudonyms. When we write “upon receiving message from some P”, this accepts a message from any pseudonym. Whenever we write “upon receiving message from $P'$, without the keyword some, this accepts a message from a fixed pseudonym $P$, and typically which pseudonym we refer to is clear from the context.

Whenever we write “send $m$ to $G(\text{Contract})$ as nym $P$” inside a user program, this sends an internal message (“send”, $m$, $P$) to the protocol wrapper $\Pi$. The protocol wrapper will then authenticate the message appropriately under pseudonym $P$. When the context is clear, we avoid writ-
Ledger and money transfers. A public ledger is denoted ledger in our ideal programs and contract programs. When a party sends $\text{amt}$ to an ideal program or a contract program, this represents an ordinary message transmission. Money transfers only take place when ideal programs or contract programs update the public ledger ledger. In other words, the symbol $\$\$ is only adopted for readability (to distinguish variables associated with money and other variables), and does not have special meaning or significance. One can simply think of this variable as having the money type.

2.6 Composability and Multiple Contracts

Extending to multiple contracts. So far, our formalism only models a single running instance of a user-specified contract ($\theta_{\text{priv}}, \theta_{\text{pub}}$). It will not be too hard to extend the wrappers to support multiple contracts sharing a global ledger, clock, pseudonyms, and ContractCash (i.e. private cash). While such an extension is straightforward (and would involve segregating different instances by associating them with a unique session string or subsession string, which we omit in our presentation), one obvious drawback is that this would result in a monolithic functionality consisting of all contract instances. This means that the proof also has to be done in a monolithic manner simultaneously proving all active contracts in the system.

Future work. To further modularize our functionality and proof, new composition theorems will be needed that are not covered by the current UC [19] or extended models such as GUC [21] and GNUC [34]. We give a brief discussion of the issues below. Since our model is expressed in the Universal Composability framework, we could apply to our functionalities and protocols standard composition operators, such as the multi-session extension [22]. However, a direct application of this operator to the wrapped functionality $\mathcal{F}(\text{IdealCash})$ would give us multiple instances of separate timers and ledgers, one for each contract - which is not what we want! The Generalized UC (GUC) framework [21] is a better starting point; it provides a way to compose multiple instances of arbitrary functionalities along with a single instance of a shared functionality as a common resource. To apply this to our scenario, we would model the timer and ledger as a single shared functionality, composed with an arbitrary number of instances of Hawk contracts. However, even the GUC framework is inadequate for our needs since it does not allow interaction between the shared functionality and others, so this approach cannot be applied directly. In our ongoing work, we further generalize GUC and overcome these technical obstacles and more. As these details are intricate and unrelated to our contributions here, we defer further discussion to a forthcoming manuscript.

A remark about UC and Generalized UC. A subtle distinction between our work and that of Kiayias et al. [53] is that while we use the ordinary UC framework, Kiayias et al. define their model in the GUC framework [21]. Generalized UC definitions appear a priori to be stronger. However, we believe the GUC distinction is unnecessary, and our definition is equally strong; in particular, since the clock, ledger,
we then define Hawk-specific primitives including freeze, compute, and finalize that are essential for enabling transactional privacy and programmability simultaneously.

### 3.1 Private Cash Specification \textit{Ideal\textsubscript{Cash}}

At a high-level, the \textit{Ideal\textsubscript{Cash}} specifies the requirements of a private ledger and currency transfer. We adopt the same “mint” and “pour” terminology from Zerocash \cite{Zerocash}.

**Mint.** The \textit{mint} operation allows a user \( P \) to transfer money from the public ledger denoted \textit{ledger} to the private pool denoted \( \text{Coins}[P] \). With each transfer, a private coin for user \( P \) is created, and associated with a value \( v \).

For correctness, the ideal program \textit{Ideal\textsubscript{Cash}} checks that the user \( P \) has sufficient funds in its public ledger \textit{ledger}[\( P \)] before creating the private coin.

**Pour.** The \textit{pour} operation allows a user \( P \) to spend money in its private bank privately. For simplicity, we define the simple case with two input coins and two output coins. This is sufficient for users to transfer any amount of money by “making change,” although it would be straightforward to support more efficient batch operations as well.

For correctness, the ideal program \textit{Ideal\textsubscript{Cash}} checks the following: 1) for the two input coins, party \( P \) indeed possesses private coins of the declared values; and 2) the two input coins sum up to equal value as the two output coins, i.e., coins neither get created or vanish.

**Privacy.** When an honest party \( P \) mints, the ideal-world adversary \( A \) learns the pair \((\( P \), \( v \))\) – since minting is raising coins from the public pool to the private pool. Operations on the public pool are observable by \( A \).

When an honest party \( P \) pours, however, the adversary \( A \) learns only the output pseudonyms \( P_1 \) and \( P_2 \). It does not learn which coin in the private pool \( \text{Coins} \) is being spent nor the name of the spender. Therefore, the spent coins are anonymous with respect to the private pool \( \text{Coins} \). To get strong anonymity, new pseudonyms \( P_1 \) and \( P_2 \) can be generated on the fly to receive each pour. We stress that as long as pour hides the sender, this “breaks” the transaction graph, thus preventing linking analysis.

If a corrupted party is the recipient of a pour, the adversary additionally learns the value of the coin it receives.

**Additional subtleties.** Later in our protocol, honest parties keep track of a wallet of coins. Whenever an honest party pours, it first checks if an appropriate coin exists in its local wallet – and if so it immediately removes the coin from the wallet (i.e., without delay). In this way, if an honest party makes multiple pour transactions in one round, it will always choose distinct coins for each pour transaction. Therefore, in our \textit{Ideal\textsubscript{Cash}} functionality, honest pourers’ coins are immediately removed from \( \text{Coins} \). Further, an honest party is not able to spend a coin paid to itself until the next round. By contrast, corrupted parties are allowed to spend coins paid to them in the same round – this is due to the fact that any message is routed immediately to the adversary, and the adversary can also choose a permutation for all messages received by the contract in the same round (see Section \ref{sec:3.3}).

Another subtlety in the \textit{Ideal\textsubscript{Cash}} functionality is that honest parties will always pour to existing pseudonyms. However, the functionality allows the adversary to pour to non-existing pseudonyms denoted \( \perp \). – in this case, effectively the private coin goes into a blackhole and cannot be retrieved.

This enables a performance optimization in our ProtCash and ContractCash protocol later – where we avoid including the \( c_t \)'s in the NIZK of \( L_{\text{Cash}} \) (see Section \ref{sec:3.3.1}). If a malicious pourer chooses to compute the wrong \( c_t \), it is as if the recipient \( P \) did not receive the pour, i.e., the pour is made to \( \perp \).

### 3.2 Hawk Specification \textit{Ideal\textsubscript{Hawk}}

To enable transactional privacy and programmability simultaneously, we now describe the specifications of new Hawk primitives, including freeze, compute, and finalize. The formal specification of the ideal program \textit{Ideal\textsubscript{Hawk}} is provided in Figure \ref{fig:ideal_hawk}. Below, we provide some explanations. We also refer the reader to Section \ref{sec:3.3} for higher-level explanations.

**Freeze.** In freeze, a party tells \textit{Ideal\textsubscript{Hawk}} to remove one coin from the private coins pool \( \text{Coins} \), and freeze it in the contract by adding it to \( \text{ContractCoins} \). The party’s private input denoted \( i \) is also recorded in \( \text{ContractCoins} \). \textit{Ideal\textsubscript{Hawk}} checks that the \( P \) has not called freeze earlier, and that a coin \((\( P \), \( v \))\) exists in \( \text{Coins} \) before proceeding with the freeze.

**Compute.** When a party \( P \) calls compute, its private input \( i \) and the value of its frozen coin \( v \) are disclosed to the manager \( P_M \).

**Finalize.** In finalize, the manager \( P_M \) submits a public input \( i_{\text{pub}} \) to \textit{Ideal\textsubscript{Hawk}}. \textit{Ideal\textsubscript{Hawk}} now computes the outcome of \( \phi_{\text{priv}} \) on all parties’ inputs and frozen coin values, and redistributes the \( \text{ContractCoins} \) based on the outcome of \( \phi_{\text{priv}} \).

To ensure money conservation, the ideal program \textit{Ideal\textsubscript{Hawk}} checks that the sum of frozen coins is equal to the sum of output coins.

**Interaction with public contract.** The \textit{Ideal\textsubscript{Hawk}} functionality is parameterized by an ordinary public contract \( \phi_{\text{pub}} \), which is included in \textit{Ideal\textsubscript{Hawk}} as a sub-module. During a finalize, \textit{Ideal\textsubscript{Hawk}} calls \( \phi_{\text{pub}}.\text{check} \). The public contract \( \phi_{\text{pub}} \) typically serves the following purposes:

- **Check the well-formedness of the manager’s input \( i_{\text{M}} \).** For example, in our financial derivatives application (Section \ref{sec:5.2}), the public contract \( \phi_{\text{pub}} \) asserts that the input corresponds to the price of a stock as reported by the stock exchange’s authentic data feed.

- **Redistribute public deposits.** If parties or the manager have aborted, or if a party has provided invalid input (e.g., less than a minimum bet) the public contract \( \phi_{\text{pub}} \) can now redistribute the parties’ public deposits to ensure financial fairness. For example, in our “Rock, Paper, Scissors” example (see Section \ref{sec:5.2}), the private contract \( \phi_{\text{priv}} \) checks if each party has frozen the minimal bet. If not, \( \phi_{\text{priv}} \) includes that information in \textit{out} so that \( \phi_{\text{pub}} \) pays that party’s public deposit to others.

**Security and privacy requirements.** The \textit{Ideal\textsubscript{Hawk}} specifies the following privacy guarantees. When an honest party \( P \) freezes money (e.g., a bid), the adversary should not observe the amount frozen. However, the adversary can observe the party’s pseudonym \( P \). We note that leaking the pseudonym \( P \) does not hurt privacy, since a party can simply create a new pseudonym \( P \) and pour to this new pseudonym immediately before the freeze.

When an honest party calls compute, the manager \( P_M \)
Figure 7: Definition of Idealhawk. Notations: ContractCoins denotes the global private coin pool defined by Idealcash; and \((i_n, val_i)\) denotes the input data and frozen coin value of party \(P_i\).

gets to observe its input and frozen coin’s value. However, the public and other contractual parties do not observe anything (unless the manager voluntarily discloses information).

Finally, during a finalize operation, the output out is declassified to the public – note that out can be empty if we do not wish to declassify any information to the public.

It is not hard to see that our ideal program Idealhawk satisfies input independent privacy and authenticity against a dishonest manager. Further, it satisfies posterior privacy as long as the manager does not voluntarily disclose information. Intuitive explanations of these security/privacy properties were provided in Section 4.2.

Timing and aborts. Our ideal program Idealhawk requires that freeze operations be done by time \(T_1\), and that compute operations be done by time \(T_2\). If a user froze coins but did not open by time \(T_2\), our ideal program Idealhawk treats \((i_n, val_i) := (0, \perp)\), and the user \(P_i\) essentially forfeits its frozen coins. Managerial aborts is not handled inside Idealhawk, but by the public portion of the contract.

Simplifying assumptions. For clarity, our basic version of Idealhawk is a stripped down version of our implementation. Specifically, our basic Idealhawk and protocols do not realize refunds of frozen coins upon managerial abort. As mentioned in Section 4.3, it is not hard to extend our protocols to support such refunds.

Other simplifying assumptions we made include the following. Our basic Idealhawk assumes that the set of pseudonyms participating in the contract as well as timeouts \(T_1\) and \(T_2\) are hard-coded in the program. This can also be easily relaxed as mentioned in Section 4.3.

4. CRYPTOGRAPHIC PROTOCOLS

4.1 Protocol Description

Our protocols are broken down into two parts: 1) the private cash part that implements direct money transfers between users; and 2) the Hawk-specific part that binds transactional privacy with programmable logic. The formal protocol descriptions are given in Figures 8 and 9. Below we explain the high-level intuition.

4.1.1 Warmup: Private Cash and Money Transfers

Our construction adopts a Zerocash-like protocol for implementing private cash and private currency transfers. For completeness, we give a brief explanation below, and we mainly focus on the pour operation which is technologically more interesting. The contract ContractCash maintains a set Coins of private coins. Each private coin is stored in the format

\[(P, \text{coin} := \text{Comm}_s(\text{val}))\]

where \(P\) denotes a party’s pseudonym, and coin commits to the coin’s value \(\text{val}\) under randomness \(s\).

During a pour operation, the spender \(P\) chooses two coins in Coins to spend, denoted \((P, \text{coin}_1)\) and \((P, \text{coin}_2)\) where \(\text{coin}_i := \text{Comm}_{s_i}(\text{val}_i)\) for \(i \in \{1, 2\}\). The pour operation pays \(\text{val}_1\) and \(\text{val}_2\) amount to two output pseudonyms denoted \(P_1\) and \(P_2\) respectively, such that \(\text{val}_1 + \text{val}_2 = \text{val}_1' + \text{val}_2'\). The spender chooses new randomness \(s_i'\) for \(i \in \{1, 2\}\), and computes the output coins as \((P_i, \text{coin}_i := \text{Comm}_{s_i'}(\text{val}_i'))\)
ContractCash

Init:  
- \text{crs}: a reference string for the underlying NIZK system
- \text{Coins}: a set of coin commitments, initially \emptyset

SpentCoins: set of spent serial numbers, initially \emptyset

Mint: Upon receiving \((\text{mint}, \text{val}, s)\) from some party \(\mathcal{P}\),
- \text{coin} := \text{Comm}_s(\text{val})
- assert \((\mathcal{P}, \text{coin}) \notin \text{Coins}
- assert \text{ledger}[\mathcal{P}] \geq \text{val}
- \text{ledger}[\mathcal{P}] := \text{ledger}[\mathcal{P}] - \text{val}
- add \((\mathcal{P}, \text{coin})\) to Coins

Pour: Anonymous receive \((\text{pour}, \pi, \{\text{sn}, \mathcal{P}_i, \text{coin}_i, \text{ct}_i\}_{i \in \{1,2\}})\)
- let \(\text{MT} := (\text{MT}.\text{root}, \{\text{sn}, \mathcal{P}_i, \text{coin}_i\}_{i \in \{1,2\}})\)
- assert NIZK.Verify(\(\mathcal{L}_{\text{POUR}}, \pi, \text{statement}\)
for \(i \in \{1,2\},
- assert \text{sn} \notin \text{SpentCoins}
- assert \((\mathcal{P}_i, \text{coin}_i) \notin \text{Coins}
- add \text{sn} to \text{SpentCoins}
- add \((\mathcal{P}_i, \text{coin}_i)\) to Coins
- send \((\text{pour}, \text{coin}_i, \text{ct}_i)\) to \(\mathcal{P}_i\),

Relation \((\text{statement}, \text{witness}) \in \mathcal{L}_{\text{POUR}}\) is defined as:
- parse \text{statement} as \((\text{MT}.\text{root}, \{\text{sn}, \mathcal{P}_i, \text{coin}_i\}_{i \in \{1,2\}})\)
- parse \text{witness} as \((\mathcal{P}, \text{sk}_\text{perf}, \{\text{branch}_i, \text{sn}, \text{val}_i, \text{coin}_i, \text{ct}_i\}_{i \in \{1,2\}})\)
- assert \(\mathcal{P}.\text{sk}_\text{perf} = \text{PRF}_\text{sk}_\text{perf}(0)\)
- assert \text{val}_1 + \text{val}_2 = \text{val}'_1 + \text{val}'_2\)
for \(i \in \{1,2\},
- \text{coin}_i := \text{Comm}_s(\text{val}_i)
- assert \text{MerkleBranch}(\text{MT}.\text{root}, \text{branch}_i, (\mathcal{P}||\text{coin}_i))
- assert \text{sn} = \text{PRF}_{\text{sk}_\text{perf}}(\mathcal{P}||\text{coin}_i)
- assert \text{coin}'_i = \text{Comm}_s(\text{val}'_i)

\text{Protocol ProtCash}

Init: \text{Wallet}: stores \(\mathcal{P}\)'s spendable coins, initially \emptyset

GenNym: sample a random seed \text{sk}_\text{perf}
- \text{pk}_\text{perf} := \text{PRF}_{\text{sk}_\text{perf}}(0)
- return \text{pk}_\text{perf}

Mint: On input \((\text{mint}, \text{val}),\)
- sample a commitment randomness \(s\)
- \text{coin} := \text{Comm}_s(\text{val})
- store \((s, \text{val}, \text{coin})\) in \text{Wallet}
- send \((\text{mint}, \text{val}, s)\) to \(\mathcal{G}(\text{ContractCash})\)

Pour \text{(as sender)}: On input \((\text{pour}, \text{val}_1, \text{val}_2, \mathcal{P}_1, \mathcal{P}_2, \text{val}'_1, \text{val}'_2)\),
- assert \text{val}_1 + \text{val}_2 = \text{val}'_1 + \text{val}'_2\)
for \(i \in \{1,2\}, \text{assert} \((s_i, \text{val}_i, \text{coin}_i) \in \text{Wallet}\)
- for some \((s_i, \text{coin}_i)\)
- let \(\text{MT} := \text{a merkle tree over ContractCash.Coin}\)
for \(i \in \{1,2\},
- remove one \((s_i, \text{val}_i, \text{coin}_i)\) from \text{Wallet}
- \text{sn} := \text{PRF}_{\text{sk}_\text{perf}}(\mathcal{P}||\text{coin}_i)
- let \text{branch}_i be the branch of \((\mathcal{P}, \text{coin}_i)\) in MT
- sample randomness \(s'_i, r_i\)
- \text{coin}'_i := \text{Comm}_s(\text{val}'_i)
- \text{ct}_i := \text{ENC}(\mathcal{P}_i, \text{epk}, r_i, \text{val}'_i||s'_i)\)
- \text{statement} := \((\text{MT}.\text{root}, \{\text{sn}, \mathcal{P}_i, \text{coin}_i\}_{i \in \{1,2\}})\)
- \text{witness} := \((\mathcal{P}, \text{sk}_\text{perf}, \{\text{branch}_i, s_i, \text{val}_i, s'_i, r_i, \text{val}'_i\})\)
- \text{Pi} := \text{NIZK.Prove}(\mathcal{L}_{\text{POUR}}, \text{statement}, \text{witness})
- \text{AnonSend}((\text{pour}, \pi, \{\text{sn}, \mathcal{P}_i, \text{coin}_i, \text{ct}_i\}_{i \in \{1,2\}})\)
- to \(\mathcal{G}(\text{ContractCash})\)

Pour \text{(as recipient)}: On receive \((\text{pour}, \text{coin}, ct)\) from \(\mathcal{G}(\text{ContractCash})\):
- let \((\text{val}(s)) := \text{DEC}(\text{sk}\_\text{anc}, ct)\)
- assert \text{Comm}_s(\text{val}) = \text{coin}
- store \((s, \text{val}, \text{coin})\) in \text{Wallet}
- output \((\text{pour}, \text{val})\)

Figure 8: ProtCash construction. A trusted setup phase generates the NIZK’s common reference string \text{crs}. For notational convenience, we omit writing the \text{crs} explicitly in the construction. The Merkle tree MT is stored in the contract and not computed on the fly – we omit stating this in the protocol for notational simplicity. The protocol wrapper \(\mathcal{H}(\cdot)\) invokes GenNym whenever a party creates a new pseudonym.
Figure 9: ProtHawk construction.
The spender gives the values $s'_i$ and $val'_i$ to the recipient $P$, for $P$ to be able to spend the coins later.

Now, the spender computes a zero-knowledge proof to show that the output coins are constructed appropriately, where correctness compasses the following aspects:

- **Existence of coins being spent.** The coins being spent $(P, \text{coin}_1)$ and $(P, \text{coin}_2)$ are indeed part of the private pool $\text{Coins}$. We remark that here the zero-knowledge property allows the spender to hide which coins it is spending – this is the key idea behind transactional privacy.

To prove this efficiently, $\text{ContractCash}$ maintains a Merkle tree $\text{MT}$ over the private pool $\text{Coins}$. Membership in the set can be demonstrated by a Merkle branch consistent with the root hash, and this is done in zero-knowledge.

- **No double spending.** Each coin $(P, \text{coin})$ has a cryptographically unique serial number $sn$ that can be computed as a pseudorandom function of $P$’s secret key and coin. To pour a coin, its serial number $sn$ must be disclosed, and a zero-knowledge proof given to show the correctness of $sn$.

$\text{ContractCash}$ checks that no $sn$ is used twice.

- **Money conservation.** The zero-knowledge proof also attests to the fact that the input coins and the output coins have equal total value.

We make some remarks about the security of the scheme. Intuitively, when an honest party pours to an honest party, the adversary $A$ does not learn the values of the output coins assuming that the commitment scheme $\text{Comm}$ is hiding, and the NIZK scheme we employ is computational zero-knowledge. The adversary $A$ can observe the nyms that receive the two output coins. However, as we remarked earlier, since these nyms can be one-time, leaking them to the adversary would be okay. Essentially we only need to break linkability at spend time to ensure transactional privacy.

When a corrupted party $P'$ pours to an honest party $P$, even though the adversary knows the opening of the coin, it cannot spend the coin $(P, \text{coin})$ once the transaction takes effect by the $\text{ContractCash}$, since $P'$ cannot demonstrate knowledge of $P$’s secret key. We stress that since the contract binds the owner’s nym $P$ to the coin, only the owner can spend it even when the opening of coin is disclosed.

**Technical subtleties.** Our $\text{ContractCash}$ uses a modified version of Zerocash to achieve stronger security in the simulation paradigm. In comparison, Zerocash adopts a strictly weaker, indistinguishability-based privacy notion called ledger indistinguishability. In multi-party protocols, indistinguishability-based security notions are strictly weaker than simulation security. Not only so, the particular ledger indistinguishability notion adopted by Zerocash [10] appears subtly questionable upon a careful examination, which we elaborate on in the Appendix. This does not imply that the Zerocash construction is necessarily insecure – however, there is no obvious path to proving their scheme secure under a simulation based paradigm.

### 4.1.2 Binding Privacy and Programmable Logic

So far, $\text{ContractCash}$, similar to Zerocash [10], only supports direct money transfers between users. We allow transactional privacy and programmable logic simultaneously.

**Freeze.** We support a new operation called *freeze*, that does not spend directly to a user, but commits the money as well as an accompanying private input to a smart contract. This is done using a pour-like protocol:

- The user $P$ chooses a private coin $(P, \text{coin}) \in \text{Coins}$, where $\text{coin} := \text{Comm}_i(sval)$. Using its secret key, $P$ computes the serial number $sn$ for $\text{coin}$ – to be disclosed with the *freeze* operation to prevent double-spending.

- The user $P$ computes a commitment $(val[|in|][k])$ to the contract where in denotes its input, and $k$ is a symmetric encryption key that is introduced due to a practical optimization explained later in Section 4.2.

- The user $P$ now makes a zero-knowledge proof attesting to similar statements as in a pour operation, i.e., that the spent coin exists in the pool $\text{Coins}$, the $sn$ is correctly constructed, and that the $val$ committed to the contract equals the value of the coin being spent. See $L_{\text{FREEZE}}$ in Figure 9 for details of the NP statement being proven.

**Compute.** Next, computation takes place off-chain to compute the payout distribution $\{val'_i\}_{i \in [n]}$ and a proof of correctness. In $\text{Hawk}$, we rely on a minimally trusted manager $M$ to perform computation. All parties would open their inputs to the manager $M$, and this is done by encrypting the opening to the manager’s public key:

$$ct := \text{ENC}(P_M.epk, r, (sval[|in|[k]|s'))$$

The ciphertext $ct$ is submitted to the smart contract along with appropriate zero-knowledge proofs of correctness. While the user can also directly send the opening to the manager off-chain, passing the ciphertext $ct$ through the smart contract would make any aborts evident such that the contract can financially punish an aborting user.

After obtaining the openings, the manager now computes the payout distribution $\{val'_i\}_{i \in [n]}$ and public output $out$ by applying the private contract $\phi_{\text{priv}}$. The manager also constructs a zero-knowledge proof attesting to the outcomes.

**Finalize.** When the manager submits the outcome of $\phi_{\text{priv}}$ and a zero-knowledge proof of correctness to $\text{ContractHawk}$, $\text{ContractHawk}$ verifies the proof and redistributes the frozen money accordingly. Here $\text{ContractHawk}$ also passes the manager’s public input $in_M$ and public output $out$ to a public contract denoted $C$. The public contract $C$ can be invoked to check the validity of the manager’s input, as well as redistribute public collateral deposit.

**Theorem 1.** Assuming that the hash function in the Merkle tree is collision resistant, the commitment scheme $\text{Comm}$ is perfectly binding and computationally hiding, the NIZK scheme is computationally zero-knowledge and simulation sound extractable, the encryption schemes $\text{ENC}$ and $\text{SENC}$ are perfectly correct and semantically secure, the PRF scheme $\text{PRF}$ is secure, then, our protocols in Figures 8 and 9 securely emulate the ideal functionality $F(\text{IdealHawk})$.

**Proof.** Deferred to the Appendix.

### 4.2 Practical Considerations

Our scheme’s main performance bottleneck is computing NIZK proofs. Specifically, NIZK proofs must be computed whenever $i)$ a user invokes a *pour*, *freeze*, or a *compute* operation; and $ii)$ the manager calls *finalize*. In our implementation, we use state-of-the-art SNARK constructions to efficiently instantiate the NIZK proofs.
Efficient SNARK circuits. A SNARK prover’s performance is mainly determined by the number of multiplication gates in the algebraic circuit to be proven [12, 46]. To achieve efficiency, we designed optimized circuits through two ways: 1) using cryptographic primitives that are SNARK-friendly, i.e., efficiently realizable as arithmetic circuits under a specific SNARK parametrization. For example, we use a SNARK-friendly collision-resistant hash function [14] to realize the Merkle tree circuit. 2) Building customized circuit generators instead of relying on compilers to translate higher level implementation. For example, our circuits rely also on standard SHA-256, RSA-OAEP encryption, and RSA signature verification (with 2048-bit keys for both), so hand-optimized these components to reduce circuit size.

Subtleties related to the NIZK proofs. Some subtle technicalities arise when we use SNARKs to instantiate NIZK proofs. As mentioned in Theorem 1, we assume that our NIZK scheme satisfies simulation sound extractability. Unfortunately, ordinary SNARKs do not offer simulation sound extractability – and our simulator cannot simply use the SNARK’s extractor since the SNARK extractor is non-blackbox, and using the SNARK extractor in a UC-style simulation proof would require running the environment in the extractor!

We therefore rely a generic transformation (see Appendix B.2) to build a simulation sound extractable NIZK from an ordinary SNARK scheme.

Optimizations. We describe an optimization that greatly enhances our performance. We therefore focus on optimizing the $O(N)$-sized finalize circuit since this is our main performance bottleneck. All other SNARK proofs in our scheme are for $O(1)$-sized circuits. Two key observations allow us to greatly improve the performance of the proof generation during finalize.

Optimization 1: Minimize SSE-secure NIZKs. First, we observe that in our proof, the simulator need not extract any new witnesses when a corrupted manager submits proofs during a finalize operation. All witnesses necessary will have been learned or extracted by the simulator at this point. Therefore, we can employ an ordinary SNARK instead of a stronger simulation sound extractable NIZK during finalize. For freeze and compute, we still use the stronger NIZK. This optimization reduces our SNARK circuit sizes by a factor of $11 \times$ as shown in Table 2 of Section 5.

Optimization 2: Minimize public-key encryption in SNARKs. Second, during finalize, the manager encrypts each party $P_i$’s output coins to $P_i$’s key, resulting in a ciphertext $c_t$. The ciphertexts $\{c_t\}_{i \in [N]}$ would then be submitted to the contract along with appropriate SNARK proofs of correctness. Here, if a public-key encryption is employed to generate the $c_t$’s, it would result in relatively large SNARK circuit size. Instead, we rely on a symmetric-key encryption scheme denoted $\text{SENC}$ in Figure 9. This requires that the manager and each $P_i$ perform a key exchange to establish a symmetric key $k_i$. During a compute, the user encrypts this $k_i$ to the manager’s public key $P_{M.epk}$ and prove that the $k$ encrypted is consistent with the $k$ committed to earlier in commit. The SNARK proof during finalize now only needs to include commitments and symmetric encryptions instead of public key encryptions in the circuit – the latter much more expensive. This second optimization additionally gains us a factor of $11 \times$ as shown in Table 2 of Section 5.

4.3 Extensions and Discussions

Open enrollment of pseudonyms. In our current formalism, parties’ pseudonyms are hardcoded and known a priori. We can easily relax this to allow open enrollment of any pseudonym that joins the contract (e.g., in an auction). Our implementation supports open enrollment.

Refunding frozen coins to users. In our implementation, we extend our basic scheme to allow the users to reclaim their frozen money after a timeout $T_3 > T_2$. To achieve this, user $P$ simply sends the contract a newly constructed coin $(P, \text{coin} := \text{Comm}_i(\text{val}))$ and proves in zero-knowledge that its value $\text{val}$ is equal to that of the frozen coin. In this case, the user can identify the previously frozen coin in the clear, i.e., there is no need to compute a zero-knowledge proof of membership within the frozen pool as is needed in a pour transaction.

Instantiating the manager with trusted hardware. In some applications, it may be a good idea to instantiate the manager using trusted hardware such as the emerging Intel SGX. In this case, the off-chain computation can take place in a secret SGX enclave that is not visible to any untrusted software or users. Alternatively, in principle, the manager role can also be split into two or more parties that jointly run a secure computation protocol – although this approach is likely to incur higher overhead.

We stress that our model is fundamentally different from placing full trust in any centralized node. Trusted hardware cannot serve as a replacement of the blockchain. Any off-chain only protocol that does not interact with the blockchain cannot offer financial fairness in the presence of aborts – even when trusted hardware is employed.

Furthermore, even the use of SGX does not obviate the need for our cryptographic protocol. If the SGX is trusted only by a subset of parties (e.g., just the parties to a particular private contract), rather than globally, then those users can benefit from the efficiency of an SGX-managed private contract, while still utilizing the more widely trusted underlying currency.

Pouring anonymously to long-lived pseudonyms. In our basic formalism of $\text{Ideal}_{\text{cash}}$, the pour operation discloses the recipient’s pseudonyms to the adversary. This means that $\text{Ideal}_{\text{cash}}$ only retains full privacy if the recipient generates a fresh, new pseudonym every time. In comparison, Zerocash [10] provides an option of anonymously spending to a long-lived pseudonym (in other words, having $\text{Ideal}_{\text{cash}}$ not reveal recipients’ pseudonyms to the adversary).

It is not too hard to extend our protocols to allow pouring anonymously to long-lived pseudonyms. However, in most applications, the transfer is typically subsequent to some interaction between the sender and recipient anyway, we opted for the approach of revealing recipients’ pseudonyms. This not only keeps our core formalism simple, but also achieves a constant factor speedup for our protocols.

Other modes of off-chain computation. In our current Hawk compiler, off-chain computation of the private contract $\phi_{priv}$ (e.g., determining the winner of an auction) is performed by a manager. It is also not hard to extend our formalism to support other modes of off-chain computation,
i.e., through an interactive protocol among participants such as secure multiparty computation. We defer the implementation of this to future work.

Remarks about the common reference string. SNARK schemes require the generation of a common reference string (CRS) during a pre-processing step. This common reference string consists of an evaluation key for the prover, and a verification key for the verifier. Unless we employ recursively composed SNARKs whose costs are significantly higher, the evaluation key is circuit-dependent, and its size is proportional to the circuit’s size. In comparison, the verification key is $O(|\text{in}| + |\text{out}|)$ in size, i.e., depends on the total length of inputs and outputs, but independent of the circuit size. We stress that only the verification key portion of the CRS needs to be included in the public contract that lives on the blockchain.

We also remark that the CRS for protocol ProtCash is shared globally, and can be generated in a one-time setup. In comparison, the CRS for each Hawk contract would depend on the Hawk contract, and therefore exists per instance of Hawk contract. To minimize the trust necessary in the CRS generation, one can employ either trusted hardware or use secure multi-party computation techniques as described by Ben-Sasson et al. [13].

5. IMPLEMENTATION

To demonstrate feasibility and estimate the practical costs, we developed a prototype of Hawk.

5.1 Compiler Implementation

Our compiler consists of several steps, which we illustrate in Figure 10 and describe below:

Preprocessing: First, the input Hawk program is split into its public contract and private contract components. The public contract is Serpent code, and can be executed directly atop an ordinary cryptocurrency platform such as Ethereum. The private contract is written in a subset of the Serpent language, and is passed as input to the Pinocchio arithmetic circuit compiler [46]. Keywords such as HawkDeclareParties are implemented as C preprocessors macros, and serve to define the input (Inp) and output (Outp) datatypes. Currently, our private contract inherits the limitations of the Pinocchio compiler, e.g., cannot support dynamic-length loops. In the future, we can relax these limitations by employing recursively compositional SNARKs.

Circuit Augmentation: After compiling the preprocessed private contract code with Pinocchio, we have an arithmetic circuit representing the input/output relation $\phi_{priv}$. This becomes a subcomponent of a larger arithmetic circuit, which we assemble using a customized circuit assembly tool. This tool is parameterized by the number of parties and the input/output datatypes, and attaches cryptographic constraints, such as computing commitments and encryptions over each party’s output value, and asserting that the input and output values satisfy the balance property.

Cryptographic Protocol: Finally, the augmented arithmetic circuit is used as input to a state-of-the-art zkSNARK library, Libsnark [15]. To avoid implementing SNARK verification in Ethereum’s Serpent language, we must add a SNARK verification opcode to Ethereum’s stack machine. We finally compile an executable program for the parties to compute the Libsnark proofs according to our protocol.

5.2 Additional Examples

Besides our running example of a sealed-bid auction (Figure 2), we implemented several other examples in Hawk, demonstrating various capabilities:

Crowdfunding: A Kickstarter-style crowdfunding campaign, (also known as an assurance contract in economics literature [2]) overcomes the “free-rider problem,” allowing a large number of parties to contribute funds towards some social good. If the minimum donation target is reached before the deadline, then the donations are transferred to a designated party (the entrepreneur); otherwise, the donations are refunded. Hawk preserves privacy in the following sense: a) the donations pledged are kept private until the deadline; and b) if the contract fails, only the manager learns the amount by which the donations were insufficient. These privacy properties may conceivably have a positive effect on the willingness of entrepreneurs to launch a crowdfunding campaign and its likelihood of success.

Rock Paper Scissors: A two-player lottery game, and naturally generalized to an $N$-player version. Our Hawk implementation provides the same notion of financial fairness as in [7,16] and provides stronger security/privacy guarantees. If any party (including the manager), cheats or aborts, the remaining honest parties receive the maximum amount they might have won otherwise. Furthermore, we go beyond prior works [7,10] by concealing the players’ moves and the pseudonym of the winner to everyone except the manager.

“Swap” Financial Instrument: An individual with a risky investment portfolio (e.g., one who owns a large number of Bitcoins) may hedge his risks by purchasing insurance (e.g., by effectively betting against the price of Bitcoin with another individual). Our example implements a simple swap instrument where the price of a stock at some future date (as reported by a trusted authority specified in the public contract) determines which of two parties receives a payout. The private contract ensures the privacy of both the details of the agreement (i.e., the price threshold) and the outcome.

The full Hawk programs for these examples are provided in the Appendix.

5.3 Performance Evaluation

We evaluated the performance for various examples, using an Amazon EC2 r3.8xlarge virtual machine. Our benchmarks actually consume at most 23.7GB of memory and 4...
We highlight some important observations:

- **On-chain computation** (dominated by zk-SNARK verification time) is very small in all cases, ranging from 8 to 16.6 milliseconds, while the cryptographic proof is constant (288 bytes). This makes the running time of the verification algorithm just linearly dependent on the size of the public statement, which is far smaller than the size of the computation, resulting in small verification time.

- **On-chain public parameters**: As mentioned in Section 4.3, not the entire SNARK common reference string (CRS) need to be on the blockchain, but only the verification key part of the CRS needs to be on-chain. Our implementation suggests the following: the private cash protocol requires a verification key of 39KB to be stored on-chain – this verification key is globally shared and there is only a single instance. Besides the globally shared public parameters, each Hawk contract will additionally require 13-114 KB of verification key to be stored on-chain, for 10 to 100 users. This per-contract verification key is circuit-dependent, i.e., depends on the contract program. We refer the readers to Section 4.3 for more discussions on techniques for performing trusted setup.

- **Manager computation**: Running private auction or crowdfunding protocols with 100 participants requires under 6.25 min proof time for the manager on a single core, and under 2.8 min on 4 cores. This translates to under $0.13 of EC2 time.

- **User computation**: Users’ proof times for pour, freeze and compute are under one minute, and independent of the number of parties.

## 6. FREQUENTLY ASKED QUESTIONS

In this section, we address some of the frequently asked questions. Some of this content repeats what is already stated earlier in the paper, but we hope that addressing these points again in a centralized section will help reiterate some important points that may be missed by a reader.

### 6.1 Motivational

"Does privacy come at the expense of transparency?"

Our work preserves the most important aspects of transparency; in particular it is **efficiently and publicly** verifiable that the balance of all coins is conserved. It is also possible to engineer the cryptography to make other properties publicly audit-able too if needed. In general, it is well-known that cryptography allows one to achieve privacy and transparency simultaneously.

We stress that privacy is of **utmost importance** in a variety of financial transactions. For example, Section 5.2 and Appendix A Figure 13 describe a financial swap/insurance contract. In real life, companies do not wish to reveal their financial dealings to the public, and the present blockchain technology will preclude any such applications that require privacy. Privacy may also be important in our auction or crowdfunding examples. This is certainly not an exhaustive list, and there are a large variety of financial instruments and contracts (e.g., futures, options, multilateral credit settlement transactions, and double-auctions) that may be modeled in our framework.

"Why are the recipient pseudonyms $P_1$ and $P_2$ revealed to the adversary? And what about Zero-cash’s persistent addresses feature?"

For simplicity, we define our protocol (see Section 4.1.1) to reveal the recipients’ pseudonyms. This only provides full privacy when the recipient generates a fresh pseudonym for each transaction. Zerocash takes an alternative approach, where the recipient has a long-term master pseudonym and the sender derives from this an ephemeral public key. It would be straightforward to add this feature to our system as well (at the cost of a constant factor blowup in performance); however, we believe in most applications (e.g., a payment made after receiving an invoice), the transfer is subsequent to some interaction between the recipient and sender. We have added a paragraph to Section 13 to discuss this issue.

"How does Hawk’s programming model differ from Ethereum?"

Our high-level approach may be superior than Ethereum: Ethereum’s language only defines the public on-chain contract. By contrast, our idea is to allow the programmer to write a single program in a centralized fashion, and we auto-generate not only the on-chain contract, but also the protocols for user agents. Having said this, we remark that presently, for the public portion of the Hawk contract, we support Ethereum’s Serpent language as is.

A gold standard for smart contract languages is non-existent,
and it remains an open research question how to design safe and usable programming languages for smart contracts. As Delmolino et al. [26] and Juels et al. [35] demonstrate, smart contract programming is an error-prone process. In future work, we would like to give a more systematic and in-depth study on how to design programming languages for smart contracts.

6.2 Technical

“SNARKs do not offer simulation extractability required for UC.” Our UC proofs require our NIZKs to be simulation extractable. In general, SNARKs are not simulation extractable and cannot suffice for UC proofs. We stress that our simulator is not using the SNARK extractor to extract, since doing so would require running the environment’s algorithm inside the extractor! Section 6.2 discusses our generic transformation from SNARK to simulation extractable NIZK (see paragraph entitled “Subtleties related to the NIZK proofs”) and we formalize this in the Appendix (see Theorem 2). Our experimental results include this transformation, and suggest the practicality of this approach. This issue is also related to the technical in-correctness of Zerocash (see paragraph entitled “Technical subtleties” in Section 6.1 and Appendix C).

Trust in the manager. “Can’t a dishonest manager forge proofs?” As we mention upfront in Sections 1.1 and 1.2, the manager need not be trusted for correctness and input independence. Further, each contract instance can choose its own manager, and the manager of one contract instance cannot affect the security of another contract instance. Similarly, the manager also need not be trusted to retain the security of the crypto-currency as a whole. Therefore, the only thing we trust the manager for is posterior privacy. To see that the manager need not be trusted for correctness, notice that in our cryptographic construction, the manager must give zero-knowledge proofs to attest to the correctness of its computation, as well as the conservation of money. The unforgeability of the proofs is cryptographically enforced by the soundness of the NIZK scheme, and a dishonest manager cannot forge proofs.

As mentioned in Section 1.3 we note that one can possibly rely on secure multi-party computation (MPC) to avoid having to trust the manager even for posterior privacy – however such a solution is unlikely to be practical in the near future, especially when a large number of parties are involved. The theoretical formulation of this full-generality MPC-based approach is detailed in Appendix C.

In our implementation, we made a conscious design choice and opted for the approach with a minimally trusted manager (rather than MPC), since we believe that this is a desirable sweet-spot that simultaneously attains practical efficiency and strong enough security for realistic applications. We stress that practical efficiency is an important goal of Hawk’s design.

In Section 1.3, we also discuss practical considerations for instantiating this manager. For the reader’s convenience, we iterate: we think that a particularly promising choice is to rely on trusted hardware such as Intel SGX to obtain higher assurance of posterior privacy. We stress again that even when we use the SGX to realize the manager, the SGX should not have to be trusted for retaining the global security of the cryptocurrency. In particular, it is a very strong assumption to require all participants to globally trust a single or a handful of SGX processor(s). With Hawk’s design, the SGX is only very minimally trusted, and is only trusted within the scope of the current contract instance.

“Is fair MPC impossible without the blockchain?” As is well known in cryptography, fairness against aborts is in general impossible in the plain model of secure multi-party computation (without a blockchain), when the majority of parties are corrupted [6,10,23]. There have been a plethora of results on this topic, beginning with the well-known result by Cleve [23], to more recent works that aim to establish a complete characterization on when fairness is possible and when not [8].

Acknowledgments

We gratefully acknowledge Jonathan Katz, Rafael Pass, and abhi shelat for helpful technical discussions about the zero-knowledge proof constructions. We also acknowledge Ari Juels and Dawn Song for general discussions about cryptocurrency smart contracts. This research is partially supported by NSF grants CNS-1314857, CNS-1445887, CNS-1518765, a Sloan Fellowship, and two Google Research Awards.

7. REFERENCES


APPENDIX

A. ADDITIONAL EXAMPLE PROGRAMS  

Crowdfunding example. In the crowdfunding example
In this financial swap instrument, the money is unrecoverable by anyone. If a party does not open after freezing its money, it is returned to the donors after a timeout. In this case, no public collateral deposits are used to guarantee financial fairness such that if any of the parties cheat, the remaining honest parties are guaranteed a payout distribution that stochastically dominates the payout distribution they would expect if every party was honest.

This is achieved using standard “collateral deposit” techniques \[7,16\]. For example, if Alice aborts, then her deposit is used to compensate Bob by the maximum amount $4. If the Manager aborts, then both Alice and Bob receive $8.

Unlike the lottery games found in Bitcoin and Ethereum \[7\] \[16\] \[20\], our contract also provides privacy. If the Manager and both parties do not voluntarily disclose information, then no one else in the system learns which of Alice or Bob won. Even when the Manager, Alice, and Bob are all corrupted, the underlying escrow cash system still provides privacy for other contracts and guarantees that the total amount of money is conserved.

B. PRELIMINARIES

Notation. In the remainder of the paper, \( f(\lambda) \approx g(\lambda) \) means that there exists a negligible function \( \nu(\lambda) \) such that \( |f(\lambda) - g(\lambda)| < \nu(\lambda) \).

B.1 Non-Interactive Zero-Knowledge Proofs

A non-interactive zero-knowledge proof system (NIZK) for an NP language \( \mathcal{L} \) consists of the following algorithms:

\begin{itemize}
  \item \( \text{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}) \), also written as \( \text{crs} \leftarrow \text{KeyGen}(1^\lambda, \mathcal{L}) \):
  \begin{itemize}
    \item Takes in a security parameter \( \lambda \), a description of the language \( \mathcal{L} \), and generates a common reference string \( \text{crs} \).
  \end{itemize}
\end{itemize}
user-defined private Hawk contract. Specifically, \( \{\text{\$val}_i\}_{i\in[N]}, \text{out} \) := \( \phi(\{\text{\$val}_i, \text{in}_i\}_{i\in[N]}, \text{in}_M) \), i.e., \( \phi \) takes in the parties’ private inputs \( \{\text{\$val}_i, \text{in}_i\}_{i\in[N]} \), private coin values \( \{\text{\$val}_i\}_{i\in[N]} \), the manager’s public input \( \text{P}_M \), and outputs the payout of each party \( \{\text{\$val}_i\}_{i\in[N]} \), and a public output out.

<table>
<thead>
<tr>
<th>( \phi_{\text{pub}} )</th>
<th>user-defined public Hawk contract.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{IdealP} )</td>
<td>ideal program</td>
</tr>
<tr>
<td>( \text{simP} )</td>
<td>simulator program</td>
</tr>
<tr>
<td>( \text{C, ContractP} )</td>
<td>contract program</td>
</tr>
<tr>
<td>( \text{prot, ProtP} )</td>
<td>user-side program</td>
</tr>
<tr>
<td>( \mathcal{F}(\cdot) )</td>
<td>ideal functionality wrapper, ( \mathcal{F}(\text{IdealP}) ) denotes an ideal functionality</td>
</tr>
<tr>
<td>( \mathcal{G}(\cdot) )</td>
<td>contract functionality wrapper, ( \mathcal{G}(\text{C}) ) denotes a contract functionality</td>
</tr>
<tr>
<td>( \Pi(\cdot) )</td>
<td>protocol wrapper, ( \Pi(\text{prot}) ) denotes user-side protocol</td>
</tr>
<tr>
<td>( \text{P} )</td>
<td>party or its pseudonym</td>
</tr>
<tr>
<td>( \text{P}_M )</td>
<td>minimally trusted manager (or its pseudonym)</td>
</tr>
<tr>
<td>( \mathcal{A} )</td>
<td>adversary</td>
</tr>
<tr>
<td>( \mathcal{E} )</td>
<td>environment</td>
</tr>
<tr>
<td>( T )</td>
<td>current time</td>
</tr>
<tr>
<td>\text{ledger}</td>
<td>global public ledger</td>
</tr>
<tr>
<td>Coins (in ideal programs)</td>
<td>private ledger, maintained by the ideal functionality</td>
</tr>
<tr>
<td>Coins (in contract programs)</td>
<td>a set of cryptographic coins stored by a contract. Private spending (including pours and freezes) must demonstrate a zero-knowledge proof of the spent coin’s membership in Coins. Further, private spending must demonstrate a cryptographic serial number sn that prevents double spending.</td>
</tr>
</tbody>
</table>

- \( \pi \leftarrow \mathcal{P}(\text{crs}, \text{stmt}, w) \): Takes in \( \text{crs} \), a statement \( \text{stmt} \), a witness \( w \) such that \( (\text{stmt}, w) \in \mathcal{L} \), and produces a proof \( \pi \).
- \( b \leftarrow \mathcal{V}(\text{crs}, \text{stmt}, \pi) \): Takes in a \( \text{crs} \), a \( \text{stmt} \), and a proof \( \pi \), and outputs 0 or 1, denoting accept or reject.

**Perfect completeness.** A NIZK system is said to be perfectly complete, if an honest prover with a valid witness can always convince an honest verifier. More formally, for any \((\text{stmt}, w) \in R\), we have that

\[
\Pr \left[ \text{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}), \pi \leftarrow \mathcal{P}(\text{crs}, \text{stmt}, w) : \mathcal{V}(\text{crs}, \text{stmt}, \pi) = 1 \right] = 1
\]

**Computational zero-knowledge.** Informally, a NIZK system is computationally zero-knowledge, if the proof does not reveal any information about the witness to any polynomial-time adversary. More formally, a NIZK system is said to be computationally zero-knowledge, if there exists a polynomial-time simulator \( S = (\hat{\mathcal{K}}, \hat{\mathcal{P}}) \), such that for all non-uniform polynomial-time adversary \( \mathcal{A} \), we have that

\[
\Pr \left[ \text{crs} \leftarrow \hat{\mathcal{K}}(1^\lambda, \mathcal{L}) : \mathcal{A}^{\hat{\mathcal{P}}(\text{crs}, \cdot, \cdot)}(\text{crs}) = 1 \right] \approx \Pr \left[ \hat{\mathcal{K}}(1^\lambda, \mathcal{L}) : \mathcal{A}^{\hat{\mathcal{P}}(\text{crs}, \cdot, \cdot)}(\text{crs}) = 1 \right]
\]

In the above, \( \hat{\mathcal{P}}(\text{crs}, \tau, \text{stmt}, w) \) verifies that \( (\text{stmt}, w) \in \mathcal{L} \), and if so, outputs \( \hat{\mathcal{P}}(\text{crs}, \tau, \text{stmt}) \) which simulates a proof without knowing a witness.

**Computational soundness.** A NIZK scheme for the language \( \mathcal{L} \) is said to be computationally sound, if for all polynomial-time adversaries \( \mathcal{A} \),

\[
\Pr \left[ \text{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}), (\text{stmt}, \pi) \leftarrow \mathcal{A}(\text{crs}) : (\mathcal{V}(\text{crs}, \text{stmt}, \pi) = 1) \land (\text{stmt} \notin \mathcal{L}) \right] \approx 0
\]

**Simulation sound extractability.** Simulation sound extractability says that even after seeing many simulated proofs, whenever the adversary makes a new proof, a simulator is able to extract a witness. Simulation extractability implies simulation soundness and non-malleability, since if the simulator can extract a valid witness from an adversary’s proof, the statement must belong to the language. More formally, a NIZK system is said to be simulation sound extractable, if there exist polynomial-time algorithms \( (\hat{\mathcal{K}}, \hat{\mathcal{P}}, \mathcal{E}) \), such that for any polynomial-time adversary \( \mathcal{A} \), it holds that

\[
\begin{align*}
\Pr \left[ (\hat{\mathcal{K}}(\hat{\text{crs}}, \tau, \text{ek}) \leftarrow \hat{\mathcal{K}}(1^\lambda, \mathcal{L}) ; \\
(\text{stmt}, \pi) \leftarrow \hat{\mathcal{A}}^{\hat{\mathcal{P}}( \mathcal{G}_a, \cdot)} (\hat{\text{crs}}, \text{ek}) ; \\
\text{w} \leftarrow \mathcal{E}(\hat{\text{crs}}, \text{ek}, \text{stmt}, \pi) : \text{stmt} \notin Q \text{ and} \\
(\text{stmt}, \text{w}) \notin \mathcal{L} \text{ and} \mathcal{V}(\text{crs}, \text{stmt}, \pi) = 1 \right] \approx 0
\end{align*}
\]

In the above, \( Q \) is the list of simulation queries. Here the \( \hat{\mathcal{K}} \) is identical to the zero-knowledge simulation setup algorithm when restricted to the first two terms.

Note that in the above definition, the adversary may be able to fake a (different) proof for a statement that has been queried, however, it is not able to forge a proof for any other invalid statement. There is a natural strengthening of the above notion where the adversary cannot even fake a different proof for a statement queried. In our paper, however, the weaker notion defined above would suffice.

**B.2 Simulation Sound Extractable Proofs from Ordinary NIZKs**

We give a generic transformation that turns an ordinary NIZK into one that satisfies simulation sound extractability. In our implementation, we use Zero-Knowledge Succinct Non-interactive ARguments of Knowledge (SNARKs) as the underlying NIZK, and apply our transformation here to make it a NIZK with simulation sound extractability.
typedef enum {OK, A_CHEAT, B_CHEAT} Output

HawkDeclareParty(Alice, Bob);
HawkDeclareTimeouts(/* hardcoded timeouts */);
HawkDeclarePublicInput(int stockprice, int threshold[5]);
HawkDeclareOutput(Output o);

int threshold_comm[5] = {/* harcoded */};

private contract swap (Imp &in, Outp &out) {
  if (shal(in.Alice.threshold) != threshold_comm)
    out.o = A_CHEAT;
  if (in.Alice.val != $10) out.o = A_CHEAT;
  if (in.Bob.val != $10) out.o = B_CHEAT;
  if (in.stockprice < in.Alice.threshold[0])
    out.Alice.val = $20;
  else out.Bob.val = $20;
}

public contract deposit {
  def receiveStockPrice(stockprice):
    // Alice and Bob each deposits $10
    // Assume the stock price authority is trusted
    // to send this contract the price
    assert msg.sender == StockPriceAuthority
    self.stockprice = stockprice
    def check(int stockprice, Output o):
      self.stockprice = stockprice
      assert msg.sender == StockPriceAuthority
      assert stockprice == self.stockprice

      if (o == OK):
        23
        21
        20
        19
        18
        17
        16
        15
        14
        13
        12
        11

        public contract deposit {
          public contract deposit {

        C. TECHNICAL SUBTLETIES IN ZEROCASH

        In general, a simulation-based security definition is more
straightforward to write and understand than ad-hoc indistingui-
shability games – although it is often more difficult to
prove or require a protocol with more overhead. Below we
highlight a subtle weakness with Zerocash’s security definition
[10], which motivates our stronger definition.

Ledger Indistinguishability leaks unintended information. The privacy guarantees of Zerocash [10] are
defined by a “Ledger Indistinguishability” game (in [10], Ap-
pendix C.1). In this game, the attacker (adaptively) generates
two sequences of queries, Q\text{left} and Q\text{right}. Each query
can either be a raw “insert” transaction (which corresponds
in our model to a transaction submitted by a corrupted
party) or else a “mint” or “pour” query (which corresponds
in our model to an instruction from the environment to an
honest party). The attacker receives (incrementally) a pair
of views of protocol executions, V\text{left} and V\text{right}, according
to one of the following two cases, and tries to discern which
case occurred: either V\text{right} is generated by applying all the
queries in Q\text{right} and respectively for V\text{right}; or else V\text{left} is
generated by interleaving the “insert” queries of Q\text{left} with the
“mint” and “pour” queries of Q\text{right}, and V\text{right} is generated
by interleaving the “insert” queries of Q\text{right} with the
“mint” and “pour” queries of Q\text{left}. The two sequences of
queries are constrained to be “publicly consistent”, which
effectively defines the information leaked to the adversary.

THEOREM 2. Assume that the SNARK scheme satisfies
perfect completeness, computational soundness, and com-
putational zero-knowledge, and that the encryption scheme
is perfectly correct, then the above construction is a zero-
knowledge proof system satisfying perfect completeness, com-
putational zero-knowledge, and simulation sound extractabil-
ity.

Proof. The proofs of perfect completeness and computa-
tional zero-knowledge are obvious. We now show that this
transformation gives a simulation sound extractable NIZK.
We construct the following simulation and extractor:

\[ \mathcal{K}(\lambda, \mathcal{L}) : \text{Run the honest } \mathcal{K} \text{ algorithm, but retain the signing key } \text{sk} \text{ as the simulation trapdoor } \tau := \text{sk}. \text{ The extraction key } ek := \text{ek}, \text{ the simulated } \mathcal{E} := \mathcal{E}(\mathcal{K}, \text{pk}, \text{pk}_e). \]

\[ \mathcal{P}(\mathcal{E}, \tau, \text{stmt}) : \text{the simulator calls } \pi := \mathcal{K}(\mathcal{E}, \text{ek}, \text{stmt}, \pi) \text{, and let } (w, \sigma) := \mathcal{D}(\text{sk}_e, c). \text{ Output } w. \]

We now show that no polynomial-time adversary \( A \) can win the simulation sound extractable game except with neg-
ligible probability. Given that the encryption scheme is per-
fecly correct, and that snark is computationally sound, the
witness \((w, \sigma)\) decrypted by \( \mathcal{E} \) must satisfy one of the fol-
lowing two cases except with negligible probability: 1) \( w \) is
a valid witness for stmt under language \( \mathcal{L} \); or 2) \( \sigma \) is a
valid signature for stmt. If stmt has not been queried by the
adversary \( A \), then it must be that \( w \) is a valid witness for
stmt, since otherwise, the simulator can easily leverage the
adversary to break the security of the signature scheme. \( \square \)
For example, the $i^{th}$ queries in both sequences must be of the same type (either “mint”, “pour”, or “insert”), and if a “pour” query includes an output to a corrupted recipient, then the output value must be the same in both queries.

However, the definition of “public consistency” is subtly overconstraining: it requires that if the $i^{th}$ query in one sequence is an (honest) “pour” query that spends a coin previously created by a (corrupt) “insert” query, then the $i^{th}$ queries in both sequences must spend coins of equal value created by prior “insert” queries. Effectively, this means that if a corrupted party sends a coin to an honest party, then the adversary may be alerted when the honest party spends it.

We stress that this does not imply any flaw with the Zerocash construction itself — however, there is no obvious path to proving their scheme secure under a simulation based paradigm.

D. SIMULATOR WRAPPER

Before we describe our proofs, we first define a simulator wrapper that will help modularize our proofs.

Simulator wrapper $S$: The ideal adversary $S$ can typically be obtained by applying the simulator wrapper $S(\cdot)$ to the user-defined portion of the simulator $\hat{\mathcal{S}}(\cdot)$. The simulator wrapper modularizes the simulator construction by factoring out the common part of the simulation pertaining to all protocols in this model of execution.

The simulator is defined formally in Figure 14.

E. FORMAL PROOF FOR PRIVATE CASH

We now prove that the protocol in Figure 8 is a secure and correct implementation of $\mathcal{F}(\text{ideal}_{\text{hawk}})$. For any real-world adversary $A$, we construct an ideal-world simulator $S$, such that no polynomial-time environment $E$ can distinguish whether it is in the real or ideal world. We first describe the construction of the simulator $S$ and then argue the indistinguishability of the real and ideal worlds.

THEOREM 3. Assuming that the hash function in the Merkle tree is collision resistant, the commitment scheme $\text{Comm}$ is perfectly binding and computationally hiding, the NIZK scheme is computationally zero-knowledge and simulation sound extractable, the encryption schemes $\text{ENC}$ and $\text{SENC}$ are perfectly correct and semantically secure, the PRF scheme $\text{PRF}$ is secure, then our protocol in Figure 3 securely simulates the ideal functionality $\mathcal{F}(\text{ideal}_{\text{cash}})$.

E.1 Ideal World Simulator

Due to Canetti [19], it suffices to construct a simulator $S$ for the dummy adversary that simply passes messages to and from the environment $E$. The ideal-world simulator $S$ also interacts with the $\mathcal{F}(\text{ideal}_{\text{cash}})$ ideal functionality. Below we construct the user-defined portion of our simulator $\hat{\mathcal{S}}$. Our ideal adversary $S$ can be obtained by applying the simulator wrapper $\hat{\mathcal{S}}(\hat{\mathcal{S}})$.

Recall that the simulator wrapper performs the ordinary setup procedure, but retains the “trapdoor” information used in creating the $\text{crs}$ for the NIZK proof system, allowing it to forge proofs for false statement and to extract witnesses from valid proofs. Since the real world adversary would see the entire state of the contract, the simulator allows the environment to see the entire state of the local instance of the contract. The environment can also submit transactions directly to the contract on behalf of corrupt parties. Such a pour transaction contains a zero-knowledge proof involving the values of coins being spent or created; the simulator must rely on its ability to extract witnesses in order to learn these values and trigger $\mathcal{F}(\text{ideal}_{\text{cash}})$ appropriately.

The environment may also send $\text{mint}$ and $\text{pour}$ instructions to honest parties that in the ideal world would be forwarded directly to $\mathcal{F}(\text{ideal}_{\text{cash}})$. These activate the simulator, but only reveal partial information about the instruction — in particular, the simulator does not learn the values of the coins being spent. The simulator handles this by writing bogus (but plausible-seeming) information to the contract.

Thus the simulator must translate transactions submitted by corrupt parties to the contract into ideal world instructions, and must translate ideal world instructions into transactions published on the contract.

The simulator $\hat{\mathcal{S}}$ is defined in more detail below:

Init. The simulator $\hat{\mathcal{S}}$ runs $(\text{crs}, \tau, \text{sk}) \leftarrow \text{NIZK.\hat{\mathcal{K}}}(1^k)$, and gives $\text{crs}$ to the environment $E$.

Simulating corrupted parties. The following messages are sent by the environment $E$ to the simulator $\hat{\mathcal{S}}(\hat{\mathcal{S}})$ which then forwards it on to both the internally simulated contract $\hat{\text{Contract}}(\text{Cash})$ and the inner simulator $\hat{\mathcal{S}}(\hat{\mathcal{S}})$.

- $\hat{\mathcal{S}}$ receives a pseudonymous mint message $(\text{mint}, \text{val}, r)$. No extra action is necessary.
- $\hat{\mathcal{S}}$ receives an anonymous pour message, $(\text{pour}, \{\text{sn}i, P_i, \text{coin}_i, \text{ct}_i\}_{i \in \{1, 2\}})$. The simulator uses $\tau$ to extract the witness from $\pi$, which includes the sender $P$ and values $\text{val}_1$, $\text{val}_2$, $\text{val}_l$, and $\text{val}_r$. If $P_i$ is an uncorrupted party, then the simulator must check whether each encryption $\text{ct}_i$ is performed correctly, since the NIZK proof does not guarantee that this is the case. The simulator performs a trial decryption using $P_i.\text{sk}$; if the decryption is not a valid opening of $\text{coin}_i$, then the simulator must avoid causing $P_i$ in the ideal world to output anything (since $P_i$ in the real world would not output anything either). The simulator therefore substitutes some default value (e.g., the name of any corrupt party $P$) for the recipient’s pseudonym. The simulator forwards $(\text{pour}, \text{val}_1, \text{val}_2, P_i, P_i', P_i', \text{val}_1', \text{val}_2')$ anonymously to $\mathcal{F}(\text{ideal}_{\text{cash}})$, where $P_i' = P_i$ if $P_i$ is uncorrupted and decryption fails, and $P_i' = P_i$ otherwise.

Simulating honest parties. When the environment $E$ sends inputs to honest parties, the simulator $S$ needs to simulate messages that corrupted parties receive, from honest parties or from functionalities in the real world. The honest parties will be simulated as below:

- $\text{GenNym}(\text{epk}, \text{spk})$: The simulator $\hat{\mathcal{S}}$ generates and records the PRF keypair, $(\text{pk}_{\text{prf}}, \text{sk}_{\text{prf}})$ and returns payload := $\text{pk}_{\text{prf}}$.
- Environment $E$ gives a $\text{mint}$ instruction to party $P$. The simulator $\hat{\mathcal{S}}$ receives $(\text{mint}, P, \text{val}, r)$ from the ideal functionality $\mathcal{F}(\text{ideal}_{\text{cash}})$. The simulator has enough information to run the honest protocol, and posts a valid $\text{mint}$ transaction to the contract.
\section*{Simulation}

\textbf{Simulating honest parties.}

- \textbf{Tick:} Environment $E$ sends input $\mathit{tick}$ to an honest party $P$: simulator $S$ receives notification ($\mathit{tick}, P$) from the ideal functionality. Simulator forwards the tick message to the simulated $G(\mathit{contract})$ functionality.

- \textbf{GenNym:} Environment $E$ sends input $\mathit{gennym}$ to an honest party $P$: simulator $S$ receives notification $\mathit{gennym}$ from the ideal functionality. Simulator $S$ honestly generates an encryption key and a signing key as defined in Figure 5, and remembers the corresponding secret keys. Simulator $S$ now calls $\mathit{simP.GenNym}(epk, spk)$ and waits for the returned value payload. Finally, the simulator passes the nym $P = (epk, spk, \mathit{payload})$ to the ideal functionality.

- \textbf{Other activations.} // From the inner idealP

If ideal functionality sends ($\mathit{transfer}, \$\mathit{amount}, P_r, P_s$), then update the ledger in the simulated $G(\mathit{Contract})$ instance accordingly.

Else, forward the message to the inner $\mathit{simP}$.

\textbf{Simulating corrupted parties.}

- \textbf{Permute:} Upon receiving ($\mathit{permute}, \mathit{perm}$) from the environment $E$, forward it to the internally simulated $G(\mathit{contract})$ and the ideal functionality.

- \textbf{Expose.} Upon receiving $\mathit{exposestate}$ from the environment $E$, expose all states of the internally simulated $G(\mathit{contract})$.

- \textbf{Other activations.}
  
  - Upon receiving ($\mathit{authenticated}, m$) from the environment $E$ on behalf of corrupted party $P$: Forward to internally simulated $G(\mathit{contract})$. If the message is of the format ($\mathit{transfer}, \$\mathit{amount}, P_i, P_s$), then forward it to the ideal functionality. Otherwise, forward to $\mathit{simP}$.
  
  - Upon receiving ($\mathit{pseudonymous}, m, P, \sigma$) from the environment $E$ on behalf of corrupted party $P$: Forward to internally simulated $G(\mathit{contract})$. Now, assert that $\sigma$ verifies just like in $G(\mathit{contract})$. If the message is of the format ($\mathit{transfer}, \$\mathit{amount}, P_i, P_s$), then forward it to the ideal functionality. Else, forward to $\mathit{simP}$.
  
  - Upon receiving ($\mathit{anonymous}, m$) from the environment $E$ on behalf of corrupted party $P$: Forward to internally simulated $G(\mathit{contract})$. If the message is of the format ($\mathit{transfer}, \$\mathit{amount}, P_i, P_s$), then forward it to the ideal functionality. Else, forward to $\mathit{simP}$.

\begin{figure}[ht]
\centering
\includegraphics[width=\textwidth]{simulator_wrapper.png}
\caption{Simulator wrapper.}
\end{figure}

- Environment $E$ gives a \texttt{pour} instruction to party $P$. The simulator $\mathit{simP}$ receives ($\mathit{pour}, P_1, P_2$) from $\mathcal{F}_{\mathit{CASH}}$. However, the simulator does not learn the name of the honest sender $P$, or the correct values for each input $\mathit{coin}_i$ (for $i \in \{1, 2\}$). Instead, the simulator uses $\tau$ to create a false proof using arbitrary values for these values in the witness. To generate each serial number $\mathit{sn}_i$ in the witness, the simulator chooses a random element from the codomain of PRF. For each recipient $P_i$ (for $i \in \{1, 2\}$), the simulator behaves differently depending on whether or not $P_i$ is corrupted:

  Case 1: $P_i$ is honest. The simulator does not know the correct output value, so instead sets $\mathit{val}_i = 0$, and computes $\mathit{coin}_i'$ and $\mathit{ct}_i$ as normal. The environment therefore sees a commitment and an encryption of 0, but without $P_i.\mathit{esk}$ it cannot distinguish between an encryption of 0 or of the correct value.

  Case 2: $P_i$ is corrupted. Since the ideal world recipient would receive $\mathit{val}_i'$ from $\mathcal{F}_{\mathit{CASH}}$, and since $P_i$ is corrupted, the simulator learns the correct value $\mathit{val}_i'$ directly. Hence $\mathit{coin}_i$ is a correct encryption of $\mathit{val}_i'$ under $P_i$'s registered encryption public key.

\section{E2 Indistinguishability of Real and Ideal Worlds}

To prove indistinguishability of the real and ideal worlds from the perspective of the environment, we will go through a sequence of hybrid games.

\textbf{Real world.} We start with the real world with a dummy adversary that simply passes messages to and from the environment $E$.

\textbf{Hybrid 1.} Hybrid 1 is the same as the real world, except that now the adversary (also referred to as the simulator) will call ($\mathcal{E}, \tau, \mathit{ek}$) $\leftarrow \mathit{NIZK.K}(1^\lambda)$ to perform a simulated setup for the NIZK scheme. The simulator will pass the simulated $\mathcal{E}$ to the environment $E$. When an honest party $P$ publishes a NIZK proof, the simulator will replace the real proof with a simulated NIZK proof before passing it onto the environment $E$. The simulated NIZK proof can be computed by calling the $\mathit{NIZK.P}(\mathcal{E}, \tau, \cdot)$ algorithm which takes only the statement as input but does not require knowledge of a witness.
FACT 1. It is immediately clear that if the NIZK scheme is computational zero-knowledge, then no polynomial-time environment $E$ can distinguish Hybrid 1 from the real world except with negligible probability.

Hybrid 2. The simulator simulates the $\mathcal{G}(\text{ContractCash})$ functionality. Since all messages to the $\mathcal{G}(\text{ContractCash})$ functionality are public, simulating the contract functionality is trivial. Therefore, Hybrid 2 is identically distributed as Hybrid 1 from the environment $E$’s view.

Hybrid 3. Hybrid 3 is the same as Hybrid 2 except the following changes. Whenever the environment passes a message to the contract (now simulated by the simulator $S$), it will sign the message with a signature verifiable under an honestly generated nym. In Hybrid 3, the simulator will replace all honest parties’ nyms and generate these nyms itself. In this way, the simulator will simulate honest parties’ signatures by signing them itself. Hybrid 3 is identically distributed as Hybrid 2 from the environment $E$’s view.

Hybrid 4. Hybrid 4 is the same as Hybrid 3 except for the following changes:

- When an honest party $P$ produces a ciphertext $ct_i$ for a recipient $P_i$, and if the recipient is also uncorrupted, then the simulator will replace this ciphertext with an encryption of 0 before passing it onto the environment $E$.

- When an honest party $P$ produces a commitment coin, then the simulator replaces this commitment with a commitment to 0.

- When an honest party $P$ computes a pseudorandom serial number $sn$, the simulator replaces this with a randomly chosen value from the codomain of PRF.

FACT 2. It is immediately clear that if the encryption scheme is semantically secure, if PRF is a pseudorandom function, and if Comm is a perfectly hiding commitment scheme, then no polynomial-time environment $E$ can distinguish Hybrid 4 from Hybrid 3 except with negligible probability.

Hybrid 5. Hybrid 5 is the same as Hybrid 4 except for the following changes. Whenever the environment $E$ passes a message to the simulator $S$ a message signed on behalf of an honest party’s nym, if the message and signature pair was not among the ones previously passed to the environment $E$, then the simulator $S$ aborts.

FACT 3. Assume that the signature scheme employed is secure; then the probability of aborting in Hybrid 5 is negligible. Notice that from the environment $E$’s view, Hybrid 5 would otherwise be identically distributed as Hybrid 4 modulo aborting.

Hybrid 6. Hybrid 6 is the same as Hybrid 5 except for the following changes. Whenever the environment passes $(\text{pour}, \pi, \{sn, P_i, \text{coin}, ct_i\})$ to the simulator (on behalf of corrupted party $P$), if the proof $\pi$ verifies under statement, then the simulator will call the NIZK’s extractor algorithm $E$ to extract witness. If the NIZK $\pi$ verifies but the extracted witness does not satisfy the relation $L_{\text{POUR}}(\text{statement}, \text{witness})$, then abort the simulation.

FACT 4. Assume that the NIZK is simulation sound extractable, then the probability of aborting in Hybrid 6 is negligible. Notice that from the environment $E$’s view, Hybrid 6 would otherwise be identically distributed as Hybrid 5 modulo aborting.

Finally, observe that Hybrid 6 is computationally indistinguishable from the ideal simulation $S$ unless one of the following bad events happens:

- A value val’ decrypted by an honest recipient is different from that extracted by the simulator. However, given that the encryption scheme is perfectly correct, this cannot happen.

- A commitment coin is different than any stored in ContractCash.coins, yet it is valid according to the relation $L_{\text{POUR}}$. Given that the merkle tree MT is computed using collision-resistant a hash function, this occurs with at most negligible probability.

- The honest public key generation algorithm results in key collisions. Obviously, this happens with negligible probability if the encryption and signature schemes are secure.

FACT 5. Given that the encryption scheme is semantically secure and perfectly correct, and that the signature scheme is secure, then Hybrid 6 is computationally indistinguishable from the ideal simulation to any polynomial-time environment $E$.

F. FORMAL PROOF FOR HAWK

We now prove our main result, Theorem 1 (see Section 4.1.2). Just as we did for private cash in Theorem 3 we will construct an ideal-world simulator $S$ for every real-world adversary $A$, such that no polynomial-time environment $E$ can distinguish whether it is in the real or ideal world.

F.1 Ideal World Simulator

Our ideal program (IdealHawk) and construction (ContractHawk and Hawk) borrows from our private cash definition and construction in a non-blackbox way (i.e., by duplicating the relevant behaviors). As such, our simulator program $\text{simP}$ also duplicates the behavior of the simulator from Appendix E involving mint and pour interactions. Hence we will here explain the behavior involving the additional freeze, compute, and finalize interactions.

Init. Same as in Appendix E

Simulating corrupted parties. The following messages are sent by the environment $E$ to the simulator $S(\text{simP})$ which then forwards it on to both the internally simulated contract $G(\text{ContractHawk})$ and the inner simulator $\text{simP}$.

- Corrupt party $\mathcal{P}$ submits a transaction $(\text{freeze}, \pi, sn, cm)$ to the contract. The simulator forwards this transaction to the contract, but also uses the trapdoor $\tau$ to extract a witness from $\pi$, including $\text{val}$ and in. The simulator then sends $(\text{freeze}, \text{val}, \text{in})$ to $\mathcal{F}_{\text{Hawk}}$.

- Corrupt party $\mathcal{P}$ submits a transaction $(\text{compute}, \pi, ct)$ to the contract. The simulator forwards this to the contract and sends compute to $\mathcal{F}_{\text{Hawk}}$. The simulator also uses $\tau$ to extract a witness from $\pi$, including $k_\ell$, which is used later. These is stored as $\text{CorruptOpen}_i := k_i$. 

26
Simulating honest parties. When the environment $E$ sends inputs to honest parties, the simulator $S$ needs to simulate messages that corrupted parties receive, from honest parties or from functionalities in the real world. The honest parties will be simulated as below:

- **Environment $E$ gives a freeze instruction to party $P$.** The simulator $S$ receives $(\text{freeze}, P)$ from $F(\text{ideal\_hawk})$. The simulator does not have any information about the actual committed values for $\text{val}$ or in. Instead, the simulator creates a bogus commitment $cm := \text{Comm}_i(0||\bot||\bot)$ that will later be opened (via a false proof) to an arbitrary value. To generate the serial number $sn$, the simulator chooses a random element from the codomain of PRF. Finally, the simulator uses $\tau$ to generate a forged proof $\pi$ and sends $(\text{freeze}, \pi, sn, cm)$ to the contract.

- **Environment $E$ gives a compute instruction to party $P$.** The simulator $S$ receives $(\text{compute}, P)$ from $F(\text{ideal\_hawk})$. The simulator behaves differently depending on whether or not the manager $P_M$ is corrupted.

  **Case 1:** $P_M$ is honest. The simulator does not know values $\text{val}$ or in. Instead, the simulator samples an encryption randomness $r$ and generates an encryption of $0$, $ct := \text{ENC}(P_M.\text{epk}, r, 0)$. Finally, the simulator uses the trapdoor $\tau$ to create a false proof $\pi$ that the commitment $cm$ and ciphertext $ct$ are consistent. The simulator then passes $(\text{compute}, \pi, ct)$ to the contract.

  **Case 2:** $P_M$ is corrupted. Since the manager $P_M$ in the ideal world would learn $\text{val}$, $in$, and $k$ at this point, the simulator learns these values instead. Hence it samples an encryption randomness $r$ and computes a valid encryption $ct := \text{ENC}(P_M.\text{epk}, r, (\text{val}||\text{in}||k))$. The simulator next uses $\tau$ to create a proof $\pi$ attesting that $ct$ is consistent with $cm$. Finally, the simulator sends $(\text{compute}, \pi, ct)$ to the contract.

- **Environment $E$ gives a finalize instruction to party $P_M$.** The simulator $S$ receives $(\text{finalize}, \text{in}, \text{out})$ from $F(\text{ideal\_hawk})$. The simulator generates the output $\text{coin}'_i$ for each party $P_i$ depending on whether $P_i$ is corrupted or not:
  
  - $P_i$ is honest: The simulator does not know the correct output value for $P_i$, so instead creates a bogus commitment $\text{coin}'_i := \text{Comm}_i(0)$ and a bogus ciphertext $ct'_i := \text{SENC}_k_i(s'_i||0)$ for sampled randomnesses $k_i$ and $s'_i$.
  
  - $P_i$ is corrupted: Since the ideal world recipient would receive $\text{val}'_i$, the simulator learns the correct value $\text{val}'_i$ directly. Notice that since $P_i$ is corrupted, the simulator has access to $k_i := \text{CorruptOpen}_i$, which it extracted earlier. The simulator therefore draws a randomness $s'_i$, and computes $\text{coin}'_i := \text{Comm}_i(\text{val}'_i)$ and $ct_i := \text{SENC}_k_i(s'_i||\text{val}'_i)$.

The simulator finally constructs a forged proof $\pi$ using the trapdoor $\tau$, and then passes $(\text{finalize}, \pi, \text{in}, \text{out}, \text{coin}', ct_i)_{i \in N}$ to $F_{\text{HAWK}}$.

F.2 Indistinguishability of Real and Ideal Worlds

To prove indistinguishability of the real and ideal worlds from the perspective of the environment, we will go through a sequence of hybrid games.

**Real world.** We start with the real world with a dummy adversary that simply passes messages to and from the environment $E$.

**Hybrid 1.** Hybrid 1 is the same as the real world, except that now the adversary (also referred to as the simulator) will call $(\text{crs}, \tau, \text{ek}) \rightarrow \text{NIZK}._{\text{K}}(1^\lambda)$ to perform a simulated setup for the NIZK scheme. The simulator will pass the simulated $\text{crs}$ to the environment $E$. When an honest party $P$ publishes a NIZK proof, the simulator will replace the real proof with a simulated NIZK proof before passing it onto the environment $E$. The simulated NIZK proof can be computed by calling the NIZK$_{\text{K}}(\text{crs}, \tau, \cdot)$ algorithm which takes only the statement as input but does not require knowledge of a witness.

**Fact 6.** It is immediately clear that if the NIZK scheme is computational zero-knowledge, then no polynomial-time environment $E$ can distinguish Hybrid 1 from the real world except with negligible probability.

**Hybrid 2.** The simulator simulates the $G(\text{ContractHawk})$ functionality. Since all messages to the $G(\text{ContractHawk})$ functionality are public, simulating the contract functionality is trivial. Therefore, Hybrid 2 is identically distributed as Hybrid 1 from the environment $E$’s view.

**Hybrid 3.** Hybrid 3 is the same as Hybrid 2 except the following changes. When an honest party sends a message to the contract (now simulated by the simulator $S$), it will sign the message with a signature verifiable under an honestly generated nym. In Hybrid 3, the simulator will replace all honest parties’ nymns and generate these nymns itself. In this way, the simulator will simulate honest parties’ signatures by signing them itself. Hybrid 3 is identically distributed as Hybrid 2 from the environment $E$’s view.

**Hybrid 4.** Hybrid 4 is the same as Hybrid 3 except for the following changes:

- **When an honest party $P$ produces a ciphertext $ct_i$ for a recipient $P_i$, and if the recipient is also uncorrupted, then the simulator will replace this ciphertext with an encryption of 0 before passing it onto the environment $E$.**

- **When an honest party $P$ produces a commitment $\text{coin}$ or $cm$, then the simulator replaces this commitment with a commitment to 0.**

- **When an honest party $P$ computes a pseudorandom serial number $sn$, the simulator replaces this with a randomly chosen value from the codomain of PRF.**

**Fact 7.** It is immediately clear that if the encryption scheme is semantically secure, if PRF is a pseudorandom function, and if $\text{Comm}$ is a perfectly hiding commitment scheme, then no polynomial-time environment $E$ can distinguish Hybrid 4 from Hybrid 3 except with negligible probability.
Hybrid 5. Hybrid 5 is the same as Hybrid 4 except for the following changes. Whenever the environment $E$ passes to the simulator $S$ a message signed on behalf of an honest party’s nym, if the message and signature pair was not among the ones previously passed to the environment $E$, then the simulator $S$ aborts.

Fact 8. Assume that the signature scheme employed is secure; then the probability of aborting in Hybrid 5 is negligible. Notice that from the environment $E$‘s view, Hybrid 5 would otherwise be identically distributed as Hybrid 4 modulo aborting.

Hybrid 6. Hybrid 6 is the same as Hybrid 5 except for the following changes. Whenever the environment passes $(\text{pour}, \pi, \{s_n, P_1, \text{coin}_i, ct\})$ (or $(\text{freeze}, \pi, s_n, cm)$) to the simulator (on behalf of corrupted party $P$), if the proof $\pi$ verifies under statement, then the simulator will call the NIZK’s extractor algorithm $E$ to extract witness. If the NIZK $\pi$ verifies but the extracted witness does not satisfy the relation $L_{\text{P2Z}}(\text{statement}, \text{witness})$ (or $L_{\text{FREEZE}}(\text{statement}, \text{witness})$), then abort the simulation.

Fact 9. Assume that the NIZK is simulation sound extractable, then the probability of aborting in Hybrid 6 is negligible. Notice that from the environment $E$‘s view, Hybrid 6 would otherwise be identically distributed as Hybrid 5 modulo aborting.

Finally, observe that Hybrid 6 is computationally indistinguishable from the ideal simulation $S$ unless one of the following bad events happens:

- A value $val'$ decrypted by an honest recipient is different from that extracted by the simulator. However, given that the encryption scheme is perfectly correct, this cannot happen.
- A commitment coin is different than any stored in $\text{ContractHawk}$.coins, yet it is valid according to the relation $L_{\text{POA}}$. Given that the merkle tree MT is computed using collision-resistant a hash function, this occurs with at most negligible probability.
- The honest public key generation algorithm results in key collisions. Obviously, this happens with negligible probability if the encryption and signature schemes are secure.

Fact 10. Given that the encryption scheme is semantically secure and perfectly correct, and that the signature scheme is secure, then Hybrid 6 is computationally indistinguishable from the ideal simulation to any polynomial-time environment $E$.

G. ADDITIONAL THEORETICAL RESULTS

In this section, we describe additional theoretical results for a more general model that “shares” the role of the (minimally trusted) manager among $n$ designated parties. In contrast to our main construction, where posterior privacy relies on a specific party (the manager) following the protocol, in this section posterior privacy is guaranteed even if a majority of the designated parties follow the protocol. Just as in our main construction, even if all the manager parties are corrupted, the correctness of the outputs as well as the security and privacy of the underlying cryptocurrency remains intact.

### G.1 Financially Fair MPC with Public Deposits

We describe a variant of the financially fair MPC result by Kumaresan et al. [38], reformulated under our formal model. We stress that while Bentov et al. [16] and Kumaresan et al. [38] also introduce formal models for cryptocurrency-based secure computation, their models are somewhat restrictive and insufficient for reasoning about general protocols in the blockchain model of secure computation — especially protocols involving pseudonymity, anonymity, or financial privacy, including the protocols described in this paper, Zerocash-like protocols [10] and other protocols of interest [35]. Further, their models are not UC compatible since they adopt special opaque entities such as coins.

Therefore, to facilitate designing and reasoning about the security of general protocols in the blockchain model of secure computation, we propose a new and comprehensive model for blockchain-based secure computation in this paper.

#### G.1.1 Definitions

Our ideal program for fair secure function evaluation is given in Figure 15. We make the following remarks about this ideal program. First, in a $\text{deposit}$ phase, parties are required to commit their inputs to the ideal functionality and make deposits of the amount $\text{Samt}$. Next, parties send a $\text{compute}$ command to the ideal functionality. When all
in an on-chain fair exchange, all parties reconstruct $\rho$. Each party deposits some money, and can only redeem its deposit if it releases its share of $\rho$. If a party aborts without releasing its share of $\rho$, its deposit will be redistributed to other honest parties.

**Figure 16:** Contract program for fair secure function evaluation.

Honest parties have issued a `compute` command, then the adversary learns the outputs of the corrupt parties. If all parties (including honest and corrupt) have issued an `compute` command, then all parties learn their respective outputs, and the deposits are returned. If not, finally, if a timeout happens defined by $T_1$, the ideal functionality checks to see if all parties have deposited. If not, this means that the computation has not even started. Therefore, simply return the deposits to those who have deposited, and no one needs to be punished. However, if some corrupt parties called `deposit` but did not call `compute`, then these parties’ deposits are redistributed to honest parties.

**G.1.2 Construction**

We now describe how to construct a protocol that realizes the functionality $F_{\text{ideal,uc}}$ in the most general case.

Our contract construction and user-side protocols are described in Figures 16 and 17 respectively. The protocol is a variant of Bentov et al. [16] and Kumaresan et al. [38], but reformulated under our formal framework. The intuition is that all parties first run an off-chain MPC protocol – at the end of this off-chain protocol, party $P_i$ obtains $y_i$, which is a secret share of its output $y$. The other share needed to recover output $y_i$ is $\rho_i$, i.e., $y_i := y_i \oplus \rho_i$. Denote $\rho := (\rho_1, \ldots, \rho_n)$. All parties also obtain random shares of the vector $\rho$ at the end of the off-chain MPC protocol. Then, in an on-chain fair exchange, all parties reconstruct $\rho$. Here, each party deposits some money, and can only redeem its deposit if it releases its share of $\rho$. If a party aborts without releasing its share of $\rho$, its deposit will be redistributed to other honest parties.

**Figure 17:** User program for fair secure function evaluation.

**Theorem 4.** Assume that the underlying MPC protocol $\Pi_f$ is UC-secure against an arbitrary number of corruptions, that the secret sharing scheme is perfectly secret against any $n – 1$ collusions, and that the commitment scheme $\text{commit}$ is perfectly binding, computationally hiding, and equivocal.

Then, the protocols described in Figures 16 and 17 securely emulate $F_{\text{ideal,uc}}$ in the presence of an arbitrary number of corruptions.

**Proof.** Suppose that $\Pi_f$ securely emulates the ideal functionality $F_{\text{sfe}}(\hat{f})$. For the proof, we replace the $\Pi_f$ in Figure 17 with $F_{\text{sfe}}(\hat{f})$, and prove the security of the protocol in the $(F_{\text{sfe}}(\hat{f}), G(\text{ContractSFE}))-\text{hybrid world}$. We describe the user-defined portion of the simulator program $\text{simP}$. The simulator wrapper was described earlier in Figure 14. During the simulation, $\text{simp}$ will receive a `deposit` instruction from the environment on behalf of corrupt parties. The ideal functionality will also notify the simulator that an honest party has deposited (without disclosing honest parties’ inputs). If the simulator has collected `deposit` instructions on behalf of all parties (from both the ideal functionality and environment), at this point the simulator
• Simulates \( n - 1 \) shares. Among these \(|K|\) shares will be assigned to corrupt parties.

• Simulates all commitments \( \{\text{com}_i\}_{i \in [n]} \): \( n - 1 \) of these commitments will be computed honestly from the simulated tokens. The last commitment will be simulated by committing to 0.

Now the simulator collects compute instructions from the ideal functionality on behalf of honest parties, and from the environment on behalf of corrupt parties. When the simulator receives a notification (compute, \( s_i, r_i \)) from the environment on behalf of a corrupt party \( P_i \), if \( s_i \) and \( r_i \) are not consistent with what was previously generated by the simulator, ignore the message. Otherwise, send compute to the ideal functionality on behalf of corrupt party \( P_i \). When the simulator receives a notification (compute, \( P_i \)) from the ideal functionality for some honest \( P_i \), unless this is the last honest \( P_i \), the simulator returns one of the previously generated and unused \( (s_i, r_i) \)'s. If this is the last honest \( P_i \), then the simulator will also get the corrupt parties' outputs \( \{y_i\}_{i \in K} \) from the ideal functionality. At this point, the simulator simulates the last honest party’s opening to be consistent with the corrupt parties’ outputs – this can be done if the secret sharing scheme is perfectly simulatable (i.e., zero-knowledge) against \( n - 1 \) collisions and the commitment scheme is equivocable.

It is not hard to see that the environment cannot distinguish between the real world and the ideal world simulation.

**Optimizations and on-chain costs.** Since \( \mathcal{F}(\text{Ideal}_{\text{sf}}) \) is simultaneously a generalization of Zerocash [11] and of earlier cryptocurrency-based MPC protocols [16,36,38], our construction satisfies the strongest definition so far. However, our construction above requires compiling a generic NIZK prover algorithm with a generic MPC compiler, it is likely slow. Our main construction, ProtHawk (see Section 1.1), can be seen as an optimization when \( n = 1 \) (i.e., the MPC is executed by only a single party). Similarly, the earlier off-chain MPC protocols [16,36,38] can be used in place of ours if the user-specified program does not involve any private money.

Even our general construction can be optimized in several ways. One obvious optimization is that not all parties need to send the commitment set \( \{\text{com}_i\}_{i \in [n]} \) to the contract. After the first party sends the commitment set, all other parties can simply send a bit to indicate that they agree. In the special case when all parties share the same output, i.e., \( y_1 = y_2 = \ldots = y_n \), it is not hard to see that the on-chain cost can be reduced to \( O(|y_1| + \lambda) \).

If we were to rely on a (programmable) random oracle model, [29] we could further reduce the on-chain cost to \( O(\lambda) \) per party (i.e., independent of the total output size). In a nutshell, we could modify the protocol to adopt a \( \rho \) of length \( \lambda \). We then apply a random oracle to expand \( \rho \) to \( |y| \) bits. Our simulation proof would still go through as long as the simulator can choose the outputs of the random oracle.

### G.2 Fair MPC with Private Deposits

The construction above leaks nothing to the public except the size of the public collateral deposit. For some applica-

---

<table>
<thead>
<tr>
<th>Ideal\text{sf-priv}({P_i}_{i \in [N]}, T_i, f)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Init:</strong> Call Ideal\text{cash} \text{Init}. Additionally:</td>
</tr>
<tr>
<td>ContractCoins: a set of coins and private inputs received by this contract, each of the form ( (P_i, \text{in}_i, \text{val}_i) ) to initialize ContractCoins := ( \emptyset )</td>
</tr>
<tr>
<td>On receiving the first $ \text{amt} \text{from} some ( P_i ), notify all parties of $ \text{amt}</td>
</tr>
<tr>
<td><strong>Deposit:</strong> Upon receiving ( \text{deposit}, $ \text{val}_i, x_i ) from ( P_i ) for some ( i \in [n] ):</td>
</tr>
<tr>
<td>assert $\text{val}_i \geq $ \text{amt} \text{and} \ T \leq T_i</td>
</tr>
<tr>
<td>assert at least one copy of ( (P_i, $\text{val}_i) \in \text{Coins} )</td>
</tr>
<tr>
<td>assert ( P_i ) has not called deposit earlier</td>
</tr>
<tr>
<td>send ( \text{deposit}(P_i) ) to ( A )</td>
</tr>
<tr>
<td>add ( (P_i, $\text{val}_i, x_i) ) to ContractCoins</td>
</tr>
<tr>
<td>remove one ( (P_i, $\text{val}_i) ) from Coins</td>
</tr>
<tr>
<td>record that ( P_i ) has called deposit</td>
</tr>
<tr>
<td><strong>Compute:</strong> Upon receiving compute from ( P_i ) for some ( i \in [N] ):</td>
</tr>
<tr>
<td>send ( \text{compute}(P_i) ) to ( A )</td>
</tr>
<tr>
<td>assert current time ( T \leq T_i )</td>
</tr>
<tr>
<td>assert that all parties called deposit</td>
</tr>
<tr>
<td>Let ( (y_1, \ldots, y_n) := f(x_1, \ldots, x_n) ).</td>
</tr>
<tr>
<td>If all honest parties have called compute, notify the adversary ( A ) of ( {y_i}_{i \in K}) where K is the set of corrupt parties.</td>
</tr>
<tr>
<td>record that ( P_i ) has called compute</td>
</tr>
<tr>
<td>If all parties have called compute:</td>
</tr>
<tr>
<td>Send each ( y_i ) to ( P_i ).</td>
</tr>
<tr>
<td>For each party ( P_i ) that deposited: add one ( (P_i, $\text{val}_i) ) to Coins</td>
</tr>
<tr>
<td><strong>Refund:</strong> Upon receiving ( \text{refund} ) from ( P_i ):</td>
</tr>
<tr>
<td>notify ( \text{refund}(P_i) ) to ( A )</td>
</tr>
<tr>
<td>assert ( T &gt; T_i )</td>
</tr>
<tr>
<td>assert ( P_i ) has not called refund earlier</td>
</tr>
<tr>
<td>assert ( P_i ) has called compute</td>
</tr>
<tr>
<td>If not all parties have called deposit, add one ( (P_i, $\text{val}_i) ) to Coins</td>
</tr>
<tr>
<td>Else $r := (k \cdot $\text{val}_i)/(n - k) \text{ where} k \text{ is the number of parties who did not call compute, and add one} (P_i, $\text{val}_i + $r) \text{to Coins}</td>
</tr>
</tbody>
</table>

---

**Figure 18:** Definition of Ideal\text{sf-priv} with private deposit. Notations: ContractCoins denotes frozen coins owned by the contract; Coins denotes the global private coin pool defined by Ideal\text{cash}.
tions, even revealing this information may leak unintended details about the application. As an example, an appropriate deposit for a private auction might correspond to the seller’s estimate of the item’s value. Therefore, we now describe the same task as in Appendix G but with private deposits instead.

7.2.1 Ideal Functionality

Figure 18 defines the ideal program for fair MPC with private deposits, Ideal_{sfe-priv}. Here, the deposit amount is known to all parties \{P_i\}_{i \in [n]} participating in the protocol, but it is not revealed to other users of the blockchain. In particular, if all parties behave honestly in the protocol, then the adversary will not learn the deposit amount. Therefore, in the Init part of this ideal functionality, some party \(P_i\) sends the deposit amount \(amt\) to the functionality, and the functionality notifies all parties of \(amt\). Otherwise, the functionality in Figure 18 is very similar to Figure 15 except that when all of \{P_i\}_{i \in [n]} are honest, the adversary does not learn the deposit amount.

7.2.2 Protocol

Figures 19 and 20 depict the user-side program and the contract program for fair MPC with private deposits.

At the beginning of the protocol, all parties \{P_i\}_{i \in [n]} agree on a deposit amount \(amt\), and \(cm_0\) and publish a commitment to \(amt\) on the blockchain. As in the case with public deposits, all parties first run an off-chain protocol after which each party \(P_i\) obtains \(\hat{y}_i\), \(\hat{y}_i\) is random by itself, and must be combined with another share \(ρ_i\) to recover \(y_i\) (i.e., the output is recovered as \(y_i := \hat{y}_i \oplus ρ_i\)). Denote \(ρ := (ρ_1, \ldots, ρ_n)\). All parties also obtain random shares of the vector \(ρ\) at the end of the off-chain MPC protocol. The vector \(ρ\) can be reconstructed when parties reveal their shares on the blockchain, such that each party \(P_i\) can obtain its outcome \(y_i\). To ensure fairness, parties make private deposits of \(amt\) to the blockchain, and can only obtain their private deposit back if they reveal their share of \(ρ\) to the block chain. The private deposit and private refund protocols use the commitment schemes and NIZKs in a similar fashion as Zerocash and Hawk.

**Theorem 5.** **Assuming that the hash function in the Merkle tree is collision resistant, the commitment scheme \(\text{Comm}\) is perfectly binding and computationally hiding, the NIZK scheme is computationally zero-knowledge and simulation sound extractable, the encryption scheme \(\text{ENC}\) is perfectly correct and semantically secure, the \(\text{PRF} \) scheme \(\text{PRF} \) is secure, then, our protocols in Figures 17 and 20 securely emulate the ideal functionality \(\mathcal{F}(\text{Ideal}_{sfe-priv})\) in Figure 18.**

**Proof.** The proof can be done in a similar manner as that of Theorem 1 (see Appendix G). \(\square\)

---

**Figure 19:** User program for fair SFE with private deposit.
ContractSFE-Priv(\{P_i\}_{i \in [n]})

**Init:** Let \( \text{crs} \) denote an appropriate common reference string for the \( \text{NIZK} \).

On first receiving \((\text{init}, \text{cm}_0)\) from \( P_i \) for some \( i \in [n] \), send \( \text{cm}_0 \) to all \( \{P_i\}_{i \in [n]} \).

**Deposit:** On receive \((\text{deposit}, \{\text{com}_j\}_{j \in [n]}, \pi, \text{sn})\) from \( P_i \):
- assert initialization was successful
- assert \( T \leq T_1 \)
- assert \( \text{sn} \notin \text{SpentCoins} \)
- \( \text{statement} := (\text{MT}. \text{root}, \text{sn}, \text{cm}_0) \)
- assert \( \text{NIZK} . \text{Verify}(L_{\text{DEPOSIT}}, \pi, \text{statement}) \)
- assert \( P_i \) has not called \( \text{deposit} \) earlier
- record that \( P_i \) has called \( \text{deposit} \)

**Compute:** Upon receiving \((\text{compute}, s_i, r_i)\) from \( P_i \):
- assert \( T \leq T_1 \)
- assert that all \( P_i \)'s have deposited, and that they have all deposited the same set \( \{\text{com}_j\}_{j \in [n]} \).
- assert that \((s_i, r_i)\) is a valid opening of \( \text{com}_i \).
- record that \( P_i \) has called \( \text{compute} \)

**Refund:** Upon receiving \((\text{refund}, \pi, \text{coin})\) from \( P_i \):
- assert \( T > T_1 \)
- assert \( P_i \) did not call \( \text{refund} \) earlier
- assert \( P_i \) called \( \text{compute} \)
- if not all parties have deposited or parties deposited different \( \{\text{com}_j\}_{j \in [n]} \) sets, \( k := 0 \)
- else \( k := \) (number of aborting parties)
- \( \text{statement} := (\text{coin}, \text{cm}_0, k, n) \)
- assert \( \text{NIZK} . \text{Verify}(L_{\text{REFUND}}, \pi, \text{statement}) \)
- add \((P_i, \text{coin})\) to \( \text{Coins} \)

Relation \((\text{statement}, \text{witness}) \in L_{\text{DEPOSIT}}\) is defined as:
- parse \( \text{statement} := (\text{MT}. \text{root}, \text{sn}, \text{cm}_0) \)
- parse \( \text{witness} := (P, \text{coin}, \text{sk}_{\text{prf}}, \text{branch}, s, \text{val}, r_0) \)
- \( \text{coin} := \text{Comm}_s(\text{val}) \)
- \( \text{cm}_0 := \text{Comm}_{r_0}(\text{val}) \)
- assert \( \text{MerkleBranch}(\text{MT}. \text{root}, \text{branch}, (P \parallel \text{coin})) \)
- assert \( P . \text{pk}_{\text{prf}} = \text{sk}_{\text{prf}}(0) \)
- assert \( \text{sn} = \text{PRF}_{\text{sk}_{\text{prf}}}(P \parallel \text{coin}) \)

Relation \((\text{statement}, \text{witness}) \in L_{\text{REFUND}}\) is defined as:
- parse \( \text{statement} := (\text{coin}, \text{cm}_0, k, n) \)
- parse \( \text{witness} := (s, r_0, \text{val}, \text{val}' \text{val}') \)
- assert \( \text{cm}_0 := \text{Comm}_{r_0}(\text{val}) \)
- assert \( \text{val}' := \text{val} + (k \cdot \text{val})/(n - k) \)
- assert \( \text{coin} := \text{Comm}_s(\text{val}') \)

Figure 20: Contract program for fair SFE with private deposit.