Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts

Ahmed Kosba*, Andrew Miller*, Elaine Shi†, Zikai Wen†, Charalampos Papamanthou*
*University of Maryland and †Cornell University
{akosba, amiller}@cs.umd.edu, {rs2358, zw385}@cornell.edu, cpap@umd.edu

***Abstract*—Emerging smart contract systems over decentralized cryptocurrencies allow mutually distrustful parties to transact safely without trusted third parties. In the event of contractual breaches or aborts, the decentralized blockchain ensures that honest parties obtain commensurate compensation. Existing systems, however, lack transactional privacy. All transactions, including flow of money between pseudonyms and amount transacted, are exposed 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 zero-knowledge proofs.

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

I. Introduction

Decentralized cryptocurrencies such as Bitcoin [48] and altcoins [20] have rapidly gained popularity, and are often quoted as a glimpse into our future [5]. 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 on a stream of *data* but also on *computations* involving this data. In Bitcoin, specifically, the data include money transfer transaction proposed by users, and the computation involves transaction validation and updating a data structure called the unspent transaction output set which, imprecisely speaking, keeps track of users' account balances. Newly emerging cryptocurrency systems such as Ethereum [57] embrace the idea of running arbitrary user-defined programs on the blockchain, thus creating an expressive decentralized smart contract system.

In this paper, we consider smart contract protocols where parties interact with such a blockchain. Assuming that the decentralized concensus protocol is secure, the blockchain can be thought of as a conceptual party (in reality decentralized) that can be *trusted for correctness and availability but not for privacy*. Such a blockchain provides a powerful abstraction for the design of distributed protocols.

The blockchain's expressive power is further enhanced by the fact that blockchains naturally embody a discrete notion of time, i.e., a clock that increments whenever a new block is mined. The existence of such a trusted clock is crucial for attaining *financial fairness* in protocols. In particular, malicious contractual parties may prematurely abort from a protocol to avoid financial payment. However, with a trusted clock, timeouts can be employed to make such aborts evident, such that the blockchain can financially penalize aborting parties by redistributing their collateral deposits to honest, non-aborting parties. This makes the blockchain model of cryptography more powerful than the traditional model without a blockchain where fairness is long known to be impossible in general when the majority of parties can be corrupt [8], [17], [24]. In summary, blockchains allow parties mutually unbeknownst to transact securely without a centrally trusted intermediary, and avoiding high legal and transactional cost.

Despite the expressiveness and power of the 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 [42], [52].

We stress that lack of privacy is a major hindrance towards the broad adoption of decentralized smart contracts, since financial transactions (e.g., insurance contracts or stock trading) are considered by many individuals and organizations as being highly secret. Although there has been progress in designing privacy-preserving cryptocurrencies such as Zerocash [11] and several others [26], [43], [54], these systems forgo programmability, and it is unclear *a priori* how to enable programmability without exposing transactions and data in cleartext to miners.

A. 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

DOI 10.1109/SP.2016.55

implement any cryptography. Our Hawk compiler is in charge of compiling the program to a cryptographic protocol between the blockchain and the users. As shown in Figure 1, a Hawk program contains two parts:

1) A *private* portion denoted ϕ_{priv} which takes in parties' input data (e.g., choices in a "rock, paper, scissors" game) as well as currency units (e.g., bids in an auction). ϕ_{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 Hawk program ϕ_{priv} is meant to protect the participants' data and the exchange of money.
2) A *public* portion denoted ϕ_{pub} that does not touch private data or money.

Our compiler will compile the Hawk program into the following pieces which jointly define a cryptographic protocol between users, the manager, and the blockchain:

- the blockchain's program which will be executed by all consensus nodes;
- a program to be executed by the users; and
- a program to be executed by a special facilitating party called the manager which will be explained shortly.

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 Hawk program ϕ_{priv} is cryptographically hidden from the public's view. Informally, this is achieved by sending "encrypted" information to the blockchain, and relying on zero-knowledge proofs to enforce the correctness of contract execution and money conservation.
- *Contractual security.* While on-chain privacy protects contractual parties' privacy against the public (i.e., parties not involved in the financial 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 I-B for a more detailed explanation.

Minimally trusted manager. The execution of Hawk contracts are facilitated by a special party called the manager. The manager can see the users' inputs and is trusted not to disclose users' private data. However, the manager is NOT to be equated with a trusted third party — *even when the manager can deviate arbitrarily from the protocol or collude with the parties, the manager cannot affect the correct execution of the contract*. In the event that a manager aborts the protocol, it can be financially penalized, and users obtain compensation accordingly.

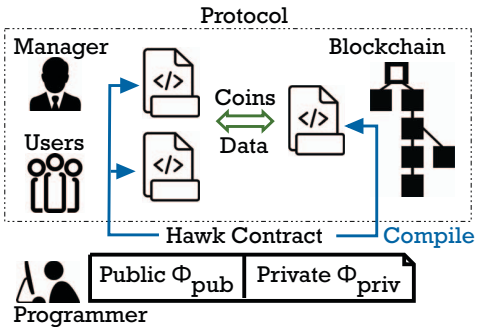

Fig. 1. Hawk **overview.**

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 IV-C and Appendix A.

Terminology. In Ethereum [57], the blockchain's portion of the protocol is called an Ethereum contract. However, this paper refers to the entire protocol defined by the Hawk program as a contract; and the blockchain's program is a constituent of the bigger protocol. In the event that a manager aborts the protocol, it can be financially penalized, and users obtain compensation accordingly.

B. 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, [55] and it is important that bidders submit bids without knowing the bid of the other people. Our example auction program contains a private portion ϕ_{priv} that determines the winning bidder and the price to be paid; and a public portion ϕ_{pub} that relies on public deposits to protect bidders from an aborting manager.

For the time being, we assume that the set of bidders are known *a priori*.

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:

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

3    // Private portion φ_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 portion φ_pub
28 public contract deposit {
29   // Manager deposited $N earlier
30   def check():  // invoked on contract completion
31     send $N to Manager  // refund manager
32   def managerTimeOut():
33     for (i in range($N)):
34       send $1 to party[i]
35 }
```

Fig. 2. Hawk program 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 $N to P" corresponds to the following semantics in our cryptographic formalism: ledger$[P]$:= ledger$[P]$ + $N – see Section II-B.

- *Input independent privacy.* Each user does not see others' bids before committing to their own (even when they collude with a potentially malicious manager). This way, users bids are independent of others' bids.
- *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.* Parties may attempt to prematurely abort from the protocol to avoid payment or affect the redistribution of wealth. If a party aborts or the auction manager aborts, the aborting party will be financially penalized while the remaining parties receive compensation. As is well-known in the cryptography literature, such fairness guarantees are not attainable in general by off-chain only protocols such as secure multi-party computation [7], [17]. 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 [57] or Zerocash [11]. The former allows for programmability but does not guarantee transactional privacy, while 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 ϕ_{pub} can additionally implement incentive structures. Our sealed auction program redistributes the manager's public deposit if it aborts. Specifically, in our sealed auction program, ϕ_{pub} defines two functions, namely check and managerTimeOut. The check 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 VI-B.

C. Contributions

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

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, Universal Composability (UC) 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 [35] 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 ϕ_{priv}. During `compute`, parties open their committed data and currency to the manager, such that the manager can compute the function ϕ_{priv}. Based on the outcome of ϕ_{priv}, the manager now constructs new private coins to be paid to each recipient. The manager then submits to the blockchain both the new private coins as well as zero-knowledge proofs of their well-formedness. At this moment, 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 several example applications, including a *sealed-bid auction*, a *"rock, paper, scissors"* game, a *crowdfunding* application, and a *swap financial instrument*. We propose interesting protocol optimizations that gained us a factor of **10**$\times$ 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 **2.85min using 4 cores**, translating to under **$0.14** of EC2 time. Further, all on-chain computation (performed by all miners) is very cheap, and under **20**ms for all cases. We will open source our Hawk framework in the near future.

D. Background and Related Work

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], [17], micropayments [4],verifiable computation [40]) 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 [28].

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 [29], [34]; on the other, efforts to formally understand the security of blockchain consensus have begun [32], [45].

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.

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. [17], Andrychowicz et al. [7], Kumaresan et al. [40], Kiayias et al. [36], as well as Zyskind et al. [59], 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 [53]. Besides recent decentralized cryptocurrencies, which guarantee authenticity but not privacy, other smart contract implementations rely on trusted servers for security [46]. 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 programs as specifications and generate cryptographic implementations, including compilers for secure multi-party computation [19], [39], [41], [51], authenticated data structures [44], and (zero-knowledge) proofs [12], [30], [31], [49]. Zheng et al. show how to generate secure distributed protocols such as sealed auctions, battleship games, and banking applications [58]. 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 [23] and use it to derive results that are similar to what we describe in the online version [37], 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 11) and ideal-world wrapper (Figure 10) 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.

II. The Blockchain Model of Cryptography

A. The Blockchain Model

We begin by informally describing the trust model and assumptions. We then propose a formal framework for the "blockchain model of cryptography" for specifying and reasoning about the security of protocols.

In this paper, the blockchain refers to a decentralized set of miners who run a secure consensus protocol to agree upon the global state. We therefore will regard the blockchain as a conceptual trusted party who is **trusted for correctness and availability, but not trusted for privacy**. The blockchain not only maintains a global ledger that stores the balance for every pseudonym, but also executes user-defined programs. More specifically, we make the following assumptions:

- *Time.* The blockchain is aware of a discrete clock that increments in *rounds*. We use the terms *rounds* and *epochs* interchangeably.
- *Public state.* All parties can observe the state of the blockchain. This means that all parties can observe the public ledger on the blockchain, as well as the state of any user-defined blockchain program (part of a contract protocol).
- *Message delivery.* Messages sent to the blockchain will arrive at the beginning of the next round. A network adversary may arbitrarily reorder messages that are sent to the blockchain within the same round. This means that the adversary may attempt a front-running attack (also referred to as the rushing adversary by cryptographers), e.g., upon observing that an honest user is trading a stock, the adversary preempts by sending a race transaction trading the same stock. Our protocols should be proven secure despite such adversarial message delivery schedules.

  We assume that all parties have a reliable channel to the blockchain, and the adversary cannot drop messages a party sends to the blockchain. In reality, this means that the overlay network must have sufficient redundancy. However, an adversary *can* drop messages delivered between parties off the blockchain.
- *Pseudonyms.* Users can make up an unbounded polynomial number of pseudonyms when communicating with the blockchain.
- *Correctness and availability.* We assume that the blockchain will perform any prescribed computation correctly. We also assume that the blockchain is always available.

Advantages of a generic blockchain model. We adopt a generic blockchain model where the blockchain can run arbitrary Turing-complete programs. In comparison, previous and concurrent works [7], [17], [40], [50] retrofit the artifacts of Bitcoin's limited and hard-to-use scripting language. In Section VII and the online version [37], we present additional theoretical results demonstrating that our generic blockchain model yields asymptotically more efficient cryptographic protocols.

B. Formally Modeling the Blockchain

Our paper adopts a carefully designed notational system such that readers may understand our constructions without understanding the precise details of our formal modeling.

We stress, however, that we give formal, precise specifications of both functionality and security, and our protocols are formally proven secure under the Universal Composability (UC) framework. In doing so, we make a separate contribution of independent interest: we are the first to propose a formal, UC-based framework for describing and proving the security of distributed protocols that interact with a blockchain — we refer to our formal model as "the blockchain model of cryptography".

Programs, wrappers, and functionalities. In the remainder of the paper, we will describe ideal specifications, as well as pieces of the protocol executed by the blockchain, the users, and the manager respectively as *programs* written in pseudocode. We refer to them as the ideal program (denoted Ideal), the blockchain program (denoted B or $\mathsf{Blockchain}$), and the user/manager program (denoted UserP) respectively.

All of our pseudo-code style programs have precise meanings in the UC framework. To "compile" a program to a UC-style functionality or protocol, we apply a wrapper to a program. Specifically, we define the following types of wrappers:

- The *ideal wrapper* $\mathcal{F}(\cdot)$ transforms an ideal program IdealP into a UC ideal functionality $\mathcal{F}(\mathsf{IdealP})$.
- The *blockchain wrapper* $\mathcal{G}(\cdot)$ transforms a blockchain program B to a blockchain functionality $\mathcal{G}(\mathsf{B})$. The blockchain functionality $\mathcal{G}(\mathsf{B})$ models the program executing on the blockchain.
- The *protocol wrapper* $\Pi(\cdot)$ transforms a user/manager program UserP into a user-side or manager-side protocol $\Pi(\mathsf{UserP})$.

One important reason for having wrappers is that wrappers implement a set of common features needed by every smart con-

tract application, including *time*, *public ledger*, *pseudonyms*, and *adversarial reordering of messages* — in this way, we need not repeat this notation for every blockchain application.

We defer our formal UC modeling to Appendix B. This will not hinder the reader in understanding our protocols as long as the reader intuitively understands our blockchain model and assumptions described in Section II-A. Before we describe our protocols, we define some notational conventions for writing "programs". Readers who are interested in the details of our formal model and proofs can refer to Appendix B.

C. Conventions for Writing Programs

Our wrapper-based system modularizes notation, and allows us to use a set of simple conventions for writing user-defined ideal programs, blockchain programs, and user protocols. We describe these conventions below.

Timer activation points. The ideal functionality wrapper $\mathcal{F}(\cdot)$ and the blockchain wrapper $\mathcal{G}(\cdot)$ implement a clock that advances in rounds. Every time the clock is advanced, the wrappers will invoke the **Timer** activation point. Therefore, by convention, we allow the ideal program or the blockchain program can define a **Timer** activation point. Timeout 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 blockchain program, every message received by the blockchain program is already delayed by a round due to the $\mathcal{G}(\cdot)$ 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 blockchain functionality will be delayed. For example, in our $\mathsf{IdealP}_{\mathsf{cash}}$ ideal program (see Figure 3), whenever the ideal functionality receives a `mint` or `pour` message, the ideal adversary $\mathcal{S}$ 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** click. By convention, we assume that 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, blockchain 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 $\mathcal{G}(\mathsf{B})$ as nym P" inside a user program, this sends an internal message ("send", m, P) to the protocol wrapper Π. The protocol wrapper will then authenticate the message appropriately under pseudonym P.

$\mathsf{IdealP}_{\mathsf{cash}}$

Init: Coins: a multiset of coins, each of the form $(\mathcal{P}, \$\mathsf{val})$

Mint: Upon receiving $(\mathtt{mint}, \$\mathsf{val})$ from some $\mathcal{P}$:
- send $(\mathtt{mint}, \mathcal{P}, \$\mathsf{val})$ to $\mathcal{A}$
- assert $\mathsf{ledger}[\mathcal{P}] \geq \$\mathsf{val}$
- $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] - \$\mathsf{val}$
- append $(\mathcal{P}, \$\mathsf{val})$ to Coins

Pour: On $(\mathtt{pour}, \$\mathsf{val}_1, \$\mathsf{val}_2, \mathcal{P}_1, \mathcal{P}_2, \$\mathsf{val}'_1, \$\mathsf{val}'_2)$ from $\mathcal{P}$:
- assert $\$\mathsf{val}_1 + \$\mathsf{val}_2 = \$\mathsf{val}'_1 + \$\mathsf{val}'_2$
- if $\mathcal{P}$ is honest,
  - assert $(\mathcal{P}, \$\mathsf{val}_i) \in$ Coins for $i \in \{1, 2\}$
  - assert $\mathcal{P}_i \neq \perp$ for $i \in \{1, 2\}$
  - remove one $(\mathcal{P}, \$\mathsf{val}_i)$ from Coins for $i \in \{1, 2\}$
  - for $i \in \{1, 2\}$, if $\mathcal{P}_i$ is corrupted, send $(\mathtt{pour}, i, \mathcal{P}_i, \$\mathsf{val}'_i)$ to $\mathcal{A}$; else send $(\mathtt{pour}, i, \mathcal{P}_i)$ to $\mathcal{A}$
- if $\mathcal{P}$ is corrupted:
  - assert $(\mathcal{P}, \$\mathsf{val}_i) \in$ Coins for $i \in \{1, 2\}$
  - remove one $(\mathcal{P}, \$\mathsf{val}_i)$ from Coins for $i \in \{1, 2\}$
- for $i \in \{1, 2\}$: add $(\mathcal{P}_i, \$\mathsf{val}'_i)$ to Coins
- for $i \in \{1, 2\}$: if $\mathcal{P}_i \neq \perp$, send $(\mathtt{pour}, \$\mathsf{val}'_i)$ to $\mathcal{P}_i$

Fig. 3. Definition of $\mathsf{IdealP}_{\mathsf{cash}}$. Notation: ledger denotes the public ledger, and Coins denotes the private pool of coins. As mentioned in Section II-C, gray background denotes batched and delayed activation. All party names correspond to pseudonyms due to notations and conventions defined in Section II-B.

When the context is clear, we avoid writing "as nym P", and simply write "send m to $\mathcal{G}(\mathsf{B})$". Our formal system also allows users to send messages anonymously to the blockchain – although this option will not be used in this paper.

Ledger and money transfers. A public ledger is denoted ledger in our ideal programs and blockchain programs. When a party sends $\mathsf{amt} to an ideal program or a blockchain program, this represents an ordinary message transmission. Money transfers only take place when ideal programs or blockchain 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.

III. Cryptography Abstractions

We now describe our cryptography abstraction in the form of ideal programs. Ideal programs define the correctness and security requirements we wish to attain by writing a specification assuming the existence of a fully trusted party. We will later prove that our real-world protocols (based on smart contracts) securely emulate the ideal programs. As mentioned earlier, an ideal program must be combined with a wrapper $\mathcal{F}$ to be endowed with exact execution semantics.

Overview. Hawk realizes the following specifications:

- *Private ledger and currency transfer.* Hawk relies on the existence of a private ledger that supports private currency transfers. We therefore first define an ideal functionality called $\mathsf{IdealP}_{\mathsf{cash}}$ that describes the requirements of a private ledger (see Figure 3). Informally speaking, earlier works such as Zerocash [11] are meant to realize (approximations

of) this ideal functionality – although technically this ought to be interpreted with the caveat that these earlier works prove indistinguishability or game-based security instead UC-based simulation security.

- Hawk-*specific primitives.* With a private ledger specified, we then define Hawk-specific primitives including *freeze*, *compute*, and *finalize* that are essential for enabling transactional privacy and programmability simultaneously.

A. Private Cash Specification $\mathsf{Ideal}\mathsf{P}_{cash}$

At a high-level, the $\mathsf{Ideal}\mathsf{P}_{\text{cash}}$ specifies the requirements of a private ledger and currency transfer. We adopt the same "mint" and "pour" terminology from Zerocash [11].

Mint. The `mint` operation allows a user $\mathcal{P}$ to transfer money from the public ledger denoted ledger to the private pool denoted $\mathsf{Coins}[\mathcal{P}]$. With each transfer, a private coin for user $\mathcal{P}$ is created, and associated with a value val.

For correctness, the ideal program $\mathsf{Ideal}\mathsf{P}_{\text{cash}}$ checks that the user $\mathcal{P}$ has sufficient funds in its public ledger $\mathsf{ledger}[\mathcal{P}]$ before creating the private coin.

Pour. The `pour` operation allows a user $\mathcal{P}$ 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 $\mathsf{Ideal}\mathsf{P}_{\text{cash}}$ checks the following: 1) for the two input coins, party $\mathcal{P}$ 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 $\mathcal{P}$ mints, the ideal-world adversary $\mathcal{A}$ learns the pair $(\mathcal{P}, \mathsf{val})$ – since minting is raising coins from the public pool to the private pool. Operations on the public pool are observable by $\mathcal{A}$.

When an honest party $\mathcal{P}$ pours, however, the adversary $\mathcal{A}$ learns only the output pseudonyms $\mathcal{P}_1$ and $\mathcal{P}_2$. It does not learn which coin in the private pool Coins is being spent nor the name of the spender. Therefore, the spent coins are anonymous with respect to the private pool Coins. To get strong anonymity, new pseudonyms $\mathcal{P}_1$ and $\mathcal{P}_2$ 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 $\mathsf{Ideal}\mathsf{P}_{\text{cash}}$ functionality, honest pourers' coins are immediately removed from 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 blockchain in the same round (see Section II and Appendix B).

Another subtlety in the $\mathsf{Ideal}\mathsf{P}_{\text{cash}}$ functionality is while honest parties always pour to existing pseudonyms, 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 $\mathsf{User}\mathsf{P}_{\text{cash}}$ and $\mathsf{Blockchain}_{\text{cash}}$ protocol later – where we avoid including the ct_i's in the NIZK of $\mathcal{L}_{\texttt{POUR}}$ (see Section IV). If a malicious pourer chooses to compute the wrong ct_i, it is as if the recipient $\mathcal{P}_i$ did not receive the pour, i.e., the pour is made to $\perp$.

B. Hawk Specification $\mathsf{Ideal}\mathsf{P}_{hawk}$

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 $\mathsf{Ideal}\mathsf{P}_{\text{hawk}}$ is provided in Figure 4. Below, we provide some explanations. We also refer the reader to Section I-C for higher-level explanations.

Freeze. In `freeze`, a party tells $\mathsf{Ideal}\mathsf{P}_{\text{hawk}}$ to remove one coin from the private coins pool Coins, and freeze it in the blockchain by adding it to FrozenCoins. The party's private input denoted in is also recorded in FrozenCoins. $\mathsf{Ideal}\mathsf{P}_{\text{hawk}}$ checks that $\mathcal{P}$ has not called `freeze` earlier, and that a coin $(\mathcal{P}, \mathsf{val})$ exists in Coins before proceeding with the freeze.

Compute. When a party $\mathcal{P}$ calls `compute`, its private input in and the value of its frozen coin val are disclosed to the manager $\mathcal{P}_{\mathcal{M}}$.

Finalize. In `finalize`, the manager $\mathcal{P}_{\mathcal{M}}$ submits a public input $\mathsf{in}_{\mathcal{M}}$ to $\mathsf{Ideal}\mathsf{P}_{\text{hawk}}$. $\mathsf{Ideal}\mathsf{P}_{\text{hawk}}$ now computes the outcome of ϕ_{priv} on all parties' inputs and frozen coin values, and redistributes the FrozenCoins based on the outcome of ϕ_{priv}. To ensure money conservation, the ideal program $\mathsf{Ideal}\mathsf{P}_{\text{hawk}}$ checks that the sum of frozen coins is equal to the sum of output coins.

Interaction with public contract. The $\mathsf{Ideal}\mathsf{P}_{\text{hawk}}$ functionality is parameterized by a public Hawk contract ϕ_{pub}, which is included in $\mathsf{Ideal}\mathsf{P}_{\text{hawk}}$ as a sub-module. During a `finalize`, $\mathsf{Ideal}\mathsf{P}_{\text{hawk}}$ calls ϕ_{pub}.`check`. The public contract ϕ_{pub} typically serves the following purposes:

- *Check the well-formedness of the manager's input* $\mathsf{in}_{\mathcal{M}}$. For example, in our financial derivatives application (Section VI-B), the public contract ϕ_{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 ϕ_{pub} can now redistribute the parties' public deposits to ensure financial

$\mathsf{IdealP}_{\textbf{hawk}}(\mathcal{P}_\mathcal{M}, \{\mathcal{P}_i\}_{i\in[N]}, T_1, T_2, \phi_{\textbf{priv}}, \phi_{\textbf{pub}})$

Init: Call $\mathsf{IdealP}_{\text{cash}}$.**Init**. Additionally:

FrozenCoins: a set of coins and private inputs received by this contract, each of the form $(\mathcal{P}, \mathsf{in}, \$\mathsf{val})$. Initialize FrozenCoins $:= \emptyset$.

Freeze: Upon receiving $(\texttt{freeze}, \$\mathsf{val}_i, \mathsf{in}_i)$ from $\mathcal{P}_i$ for some $i \in [N]$:

assert current time $T < T_1$
assert $\mathcal{P}_i$ has not called `freeze` earlier.
assert at least one copy of $(\mathcal{P}_i, \$\mathsf{val}_i) \in$ Coins
send $(\texttt{freeze}, \mathcal{P}_i)$ to $\mathcal{A}$

add $(\mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$ to FrozenCoins
remove one $(\mathcal{P}_i, \$\mathsf{val}_i)$ from Coins

Compute: Upon receiving `compute` from $\mathcal{P}_i$ for some $i \in [N]$:

assert current time $T_1 \le T < T_2$
if $\mathcal{P}_\mathcal{M}$ is corrupted, send $(\texttt{compute}, \mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$ to $\mathcal{A}$
else send $(\texttt{compute}, \mathcal{P}_i)$ to $\mathcal{A}$

let $(\mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$ be the item in FrozenCoins corresponding to $\mathcal{P}_i$
send $(\texttt{compute}, \mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$ to $\mathcal{P}_\mathcal{M}$

Finalize: Upon receiving $(\texttt{finalize}, \mathsf{in}_\mathcal{M}, \mathsf{out})$ from $\mathcal{P}_\mathcal{M}$:

assert current time $T \ge T_2$
assert $\mathcal{P}_\mathcal{M}$ has not called `finalize` earlier
for $i \in [N]$:
  let $(\$\mathsf{val}_i, \mathsf{in}_i) := (0, \perp)$ if $\mathcal{P}_i$ has not called `compute`
$(\{\$\mathsf{val}'_i\}, \mathsf{out}^\dagger) := \phi_{\text{priv}}(\{\$\mathsf{val}_i, \mathsf{in}_i\}, \mathsf{in}_\mathcal{M})$
assert $\mathsf{out}^\dagger = \mathsf{out}$
assert $\sum_{i\in[N]} \$\mathsf{val}_i = \sum_{i\in[N]} \$\mathsf{val}'_i$
send $(\texttt{finalize}, \mathsf{in}_\mathcal{M}, \mathsf{out})$ to $\mathcal{A}$
for each corrupted $\mathcal{P}_i$ that called `compute`: send $(\mathcal{P}_i, \$\mathsf{val}'_i)$ to $\mathcal{A}$

call ϕ_{pub}.check$(\mathsf{in}_\mathcal{M}, \mathsf{out})$
for $i \in [N]$ such that $\mathcal{P}_i$ called `compute`:
  add $(\mathcal{P}_i, \$\mathsf{val}'_i)$ to Coins
  send $(\texttt{finalize}, \$\mathsf{val}'_i)$ to $\mathcal{P}_i$

ϕ_{pub}: Run a local instance of public contract ϕ_{pub}. Messages between the adversary to ϕ_{pub}, and from ϕ_{pub} to parties are forwarded directly.
Upon receiving message $(\texttt{pub}, m)$ from party $\mathcal{P}$:
  notify $\mathcal{A}$ of $(\texttt{pub}, m)$

send m to ϕ_{pub} on behalf of $\mathcal{P}$

$\mathsf{IdealP}_{\textbf{cash}}$: include $\mathsf{IdealP}_{\text{cash}}$ (Figure 3).

Fig. 4. Definition of $\mathsf{IdealP}_{\text{hawk}}$. Notations: FrozenCoins denotes frozen coins owned by the contract; Coins denotes the global private coin pool defined by $\mathsf{IdealP}_{\text{cash}}$; and $(\mathsf{in}_i, \mathsf{val}_i)$ denotes the input data and frozen coin value of party $\mathcal{P}_i$.

fairness. For example, in our "Rock, Paper, Scissors" example (see Section VI-B), the private contract ϕ_{priv} checks if each party has frozen the minimal bet. If not, ϕ_{priv} includes that information in out so that ϕ_{pub} pays that party's public deposit to others.

Security and privacy requirements. The $\mathsf{IdealP}_{\text{hawk}}$ specifies the following privacy guarantees. When an honest party $\mathcal{P}$ freezes money (e.g., a bid), the adversary should not observe the amount frozen. However, the adversary can observe the party's pseudonym $\mathcal{P}$. We note that leaking the pseudonym $\mathcal{P}$ does not hurt privacy, since a party can simply create a new pseudonym $\mathcal{P}$ and pour to this new pseudonym immediately before the freeze.

When an honest party calls `compute`, the manager $\mathcal{P}_\mathcal{M}$ 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 $\mathsf{IdealP}_{\text{hawk}}$ 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 I-B.

Timing and aborts. Our ideal program $\mathsf{IdealP}_{\text{hawk}}$ 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 $\mathsf{IdealP}_{\text{hawk}}$ treats $(\mathsf{in}_i, \mathsf{val}_i) := (0, \perp)$, and the user $\mathcal{P}_i$ essentially forfeits its frozen coins. Managerial aborts is not handled inside $\mathsf{IdealP}_{\text{hawk}}$, but by the public portion of the contract.

Simplifying assumptions. For clarity, our basic version of $\mathsf{IdealP}_{\text{hawk}}$ is a stripped down version of our implementation. Specifically, our basic $\mathsf{IdealP}_{\text{hawk}}$ and protocols do not realize refunds of frozen coins upon managerial abort. As mentioned in Section IV-C, it is not hard to extend our protocols to support such refunds.

Other simplifying assumptions we made include the following. Our basic $\mathsf{IdealP}_{\text{hawk}}$ 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 IV-C.

IV. Cryptographic Protocols

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 5 and 6. Below we explain the high-level intuition.

A. 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 technically more interesting. The blockchain program $\mathsf{Blockchain}_{\text{cash}}$ maintains a set Coins of private coins. Each private coin is of the format

$$(\mathcal{P}, \mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val}))$$

where $\mathcal{P}$ denotes a party's pseudonym, and coin commits to the coin's value $\$\mathsf{val}$ under randomness s.

Protocol $\mathsf{UserP}_{\mathbf{cash}}$

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

GenNym: sample a random seed $\mathsf{sk}_{\mathtt{prf}}$
$\mathsf{pk}_{\mathtt{prf}} := \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(0)$
return $\mathsf{pk}_{\mathtt{prf}}$

Mint: On input $(\mathtt{mint}, \$\mathsf{val})$,
sample a commitment randomness s
$\mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val})$
store $(s, \$\mathsf{val}, \mathsf{coin})$ in Wallet
send $(\mathtt{mint}, \$\mathsf{val}, s)$ to $\mathcal{G}(\mathsf{Blockchain}_{\mathsf{cash}})$

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

Pour (as recipient): On receive $(\mathtt{pour}, \mathsf{coin}, \mathsf{ct})$ from $\mathcal{G}(\mathsf{Blockchain}_{\mathsf{cash}})$:
let $(\$\mathsf{val} \| s) := \mathsf{DEC}(\mathsf{esk}, \mathsf{ct})$
assert $\mathsf{Comm}_s(\$\mathsf{val}) = \mathsf{coin}$
store $(s, \$\mathsf{val}, \mathsf{coin})$ in Wallet
output $(\mathtt{pour}, \$\mathsf{val})$

$\mathsf{Blockchain}_{\mathsf{cash}}$

Init: crs: a reference string for the underlying NIZK system
Coins: a set of coin commitments, initially $\emptyset$
SpentCoins: set of spent serial numbers, initially $\emptyset$

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

Pour: Anonymous receive $(\mathtt{pour}, \pi, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}_i, \mathsf{ct}_i\}_{i\in\{1,2\}})$
let MT be a merkle tree built over Coins
$\mathsf{statement} := (\mathsf{MT.root}, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}_i\}_{i\in\{1,2\}})$
assert $\mathsf{NIZK.Verify}(\mathcal{L}_{\mathtt{POUR}}, \pi, \mathsf{statement})$
for $i \in \{1,2\}$,
assert $\mathsf{sn}_i \notin$ SpentCoins
assert $(\mathcal{P}_i, \mathsf{coin}_i) \notin$ Coins
add sn_i to SpentCoins
add $(\mathcal{P}_i, \mathsf{coin}_i)$ to Coins
send $(\mathtt{pour}, \mathsf{coin}_i, \mathsf{ct}_i)$ to $\mathcal{P}_i$,

---

Relation $(\mathsf{statement}, \mathsf{witness}) \in \mathcal{L}_{\mathtt{POUR}}$ is defined as:
parse statement as $(\mathsf{MT.root}, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}'_i\}_{i\in\{1,2\}})$
parse witness as $(\mathcal{P}, \mathsf{sk}_{\mathtt{prf}}, \{\mathsf{branch}_i, s_i, \$\mathsf{val}_i, s'_i, r_i, \$\mathsf{val}'_i\})$
assert $\mathcal{P}.\mathsf{pk}_{\mathtt{prf}} = \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(0)$
assert $\$\mathsf{val}_1 + \$\mathsf{val}_2 = \$\mathsf{val}'_1 + \$\mathsf{val}'_2$
for $i \in \{1,2\}$,
$\mathsf{coin}_i := \mathsf{Comm}_{s_i}(\$\mathsf{val}_i)$
assert $\mathsf{MerkleBranch}(\mathsf{MT.root}, \mathsf{branch}_i, (\mathcal{P} \| \mathsf{coin}_i))$
assert $\mathsf{sn}_i = \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(\mathcal{P} \| \mathsf{coin}_i)$
assert $\mathsf{coin}'_i = \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$

Fig. 5. $\mathsf{UserP}_{\mathsf{cash}}$ construction. A trusted setup phase generates the NIZK's common reference string crs. For notational convenience, we omit writing the crs explicitly in the construction. The Merkle tree MT is stored on the blockchain and not computed on the fly – we omit stating this in the protocol for notational simplicity. The protocol wrapper $\Pi(\cdot)$ invokes $\mathbf{GenNym}$ whenever a party creates a new pseudonym.

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

$$\left(\mathcal{P}_i, \mathsf{coin}_i := \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)\right)$$

The spender gives the values s'_i and val'_i to the recipient $\mathcal{P}_i$ for $\mathcal{P}_i$ 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 $(\mathcal{P}, \mathsf{coin}_1)$ and $(\mathcal{P}, \mathsf{coin}_2)$ are indeed part of the private pool 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, $\mathsf{Blockchain}_{\mathsf{cash}}$ maintains a Merkle tree MT over the private pool 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 $(\mathcal{P}, \mathsf{coin})$ has a cryptographically unique serial number sn that can be computed as a pseudorandom function of $\mathcal{P}$'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. $\mathsf{Blockchain}_{\mathsf{cash}}$ 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 $\mathcal{A}$ does not learn the values of the output coins assuming that the commitment scheme Comm is hiding, and the NIZK scheme we employ is computational zero-knowledge. The adversary $\mathcal{A}$ 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.

$\mathsf{Blockchain}_{\mathrm{hawk}}(\mathcal{P}_{\mathcal{M}}, \{\mathcal{P}_i\}_{i\in[N]}, T_1, T_2, \phi_{\mathrm{priv}}, \phi_{\mathrm{pub}})$

Init: See $\mathsf{IdealP}_{\mathrm{hawk}}$ for description of parameters
Call $\mathsf{Blockchain}_{\mathrm{cash}}$.**Init**.

Freeze: Upon receiving $(\texttt{freeze}, \pi, \mathsf{sn}_i, \mathsf{cm}_i)$ from $\mathcal{P}_i$:
- assert current time $T \leq T_1$
- assert this is the first `freeze` from $\mathcal{P}_i$
- let MT be a merkle tree built over Coins
- assert $\mathsf{sn}_i \notin \mathsf{SpentCoins}$
- statement $:= (\mathcal{P}_i, \mathsf{MT.root}, \mathsf{sn}_i, \mathsf{cm}_i)$
- assert $\mathsf{NIZK.Verify}(\mathcal{L}_{\texttt{FREEZE}}, \pi, \mathsf{statement})$
- add sn_i to SpentCoins and store cm_i for later

Compute: Upon receiving $(\texttt{compute}, \pi, \mathsf{ct})$ from $\mathcal{P}_i$:
- assert $T_1 \leq T < T_2$ for current time T
- assert $\mathsf{NIZK.Verify}(\mathcal{L}_{\texttt{COMPUTE}}, \pi, (\mathcal{P}_{\mathcal{M}}, \mathsf{cm}_i, \mathsf{ct}))$
- send $(\texttt{compute}, \mathcal{P}_i, \mathsf{ct})$ to $\mathcal{P}_{\mathcal{M}}$

Finalize: On receiving $(\texttt{finalize}, \pi, \mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$ from $\mathcal{P}_{\mathcal{M}}$:
- assert current time $T \geq T_2$
- for every $\mathcal{P}_i$ that has not called `compute`, set $\mathsf{cm}_i := \perp$
- statement $:= (\mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{cm}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$
- assert $\mathsf{NIZK.Verify}(\mathcal{L}_{\texttt{FINALIZE}}, \pi, \mathsf{statement})$
- for $i \in [N]$:
  - assert $\mathsf{coin}'_i \notin \mathsf{Coins}$
  - add coin'_i to Coins
  - send $(\texttt{finalize}, \mathsf{coin}'_i, \mathsf{ct}_i)$ to $\mathcal{P}_i$
- Call ϕ_{pub}.check$(\mathsf{in}_{\mathcal{M}}, \mathsf{out})$

$\mathsf{Blockchain}_{\mathrm{cash}}$: include $\mathsf{Blockchain}_{\mathrm{cash}}$
ϕ_{pub} : include user-defined public contract ϕ_{pub}

Relation $(\mathsf{statement}, \mathsf{witness}) \in \mathcal{L}_{\texttt{FREEZE}}$ is defined as:
- parse statement as $(\mathcal{P}, \mathsf{MT.root}, \mathsf{sn}, \mathsf{cm})$
- parse witness as $(\mathsf{coin}, \mathsf{sk}_{\mathtt{prf}}, \mathsf{branch}, s, \$\mathsf{val}, \mathsf{in}, k, s')$
- coin $:= \mathsf{Comm}_s(\$\mathsf{val})$
- assert $\mathsf{MerkleBranch}(\mathsf{MT.root}, \mathsf{branch}, (\mathcal{P}\|\mathsf{coin}))$
- assert $\mathcal{P}.\mathsf{pk}_{\mathtt{prf}} = \mathsf{sk}_{\mathtt{prf}}(0)$
- assert $\mathsf{sn} = \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(\mathcal{P}\|\mathsf{coin})$
- assert $\mathsf{cm} = \mathsf{Comm}_{s'}(\$\mathsf{val}\|\mathsf{in}\|k)$

Relation $(\mathsf{statement}, \mathsf{witness}) \in \mathcal{L}_{\texttt{COMPUTE}}$ is defined as:
- parse statement as $(\mathcal{P}_{\mathcal{M}}, \mathsf{cm}, \mathsf{ct})$
- parse witness as $(\$\mathsf{val}, \mathsf{in}, k, s', r)$
- assert $\mathsf{cm} = \mathsf{Comm}_{s'}(\$\mathsf{val}\|\mathsf{in}\|k)$
- assert $\mathsf{ct} = \mathsf{ENC}(\mathcal{P}_{\mathcal{M}}.\mathsf{epk}, r, (\$\mathsf{val}\|\mathsf{in}\|k\|s'))$

Relation $(\mathsf{statement}, \mathsf{witness}) \in \mathcal{L}_{\texttt{FINALIZE}}$ is defined as:
- parse statement as $(\mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{cm}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$
- parse witness as $\{s_i, \$\mathsf{val}_i, \mathsf{in}_i, s'_i, k_i\}_{i\in[N]}$
- $(\{\$\mathsf{val}'_i\}_{i\in[N]}, \mathsf{out}) := \phi_{\mathrm{priv}}(\{\$\mathsf{val}_i, \mathsf{in}_i\}_{i\in[N]}, \mathsf{in}_{\mathcal{M}})$
- assert $\sum_{i\in[N]} \$\mathsf{val}_i = \sum_{i\in[N]} \$\mathsf{val}'_i$
- for $i \in [N]$:
  - assert $\mathsf{cm}_i = \mathsf{Comm}_{s_i}(\$\mathsf{val}_i\|\mathsf{in}_i\|k_i))$
    $\vee(\$\mathsf{val}_i, \mathsf{in}_i, k_i, s_i, \mathsf{cm}_i) = (0, \perp, \perp, \perp, \perp)$
  - assert $\mathsf{ct}_i = \mathsf{SENC}_{k_i}(s'_i\|\$\mathsf{val}'_i)$
  - assert $\mathsf{coin}'_i = \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$

Protocol $\mathsf{UserP}_{\mathbf{hawk}}(\mathcal{P}_{\mathcal{M}}, \{\mathcal{P}_i\}_{i\in[N]}, T_1, T_2, \phi_{\mathbf{priv}}, \phi_{\mathbf{pub}})$

Init: Call $\mathsf{UserP}_{\mathrm{cash}}$.**Init**.

Protocol for a party $\mathcal{P} \in \{\mathcal{P}_i\}_{i\in[N]}$:

Freeze: On input $(\texttt{freeze}, \$\mathsf{val}, \mathsf{in})$ as party $\mathcal{P}$:
- assert current time $T < T_1$
- assert this is the first `freeze` input
- let MT be a merkle tree over $\mathsf{Blockchain}_{\mathrm{cash}}$.Coins
- assert that some entry $(s, \$\mathsf{val}, \mathsf{coin}) \in$ Wallet for some (s, coin)
- remove one $(s, \$\mathsf{val}, \mathsf{coin})$ from Wallet
- $\mathsf{sn} := \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(\mathcal{P}\|\mathsf{coin})$
- let branch be the branch of $(\mathcal{P}, \mathsf{coin})$ in MT
- sample a symmetric encryption key k
- sample a commitment randomness s'
- $\mathsf{cm} := \mathsf{Comm}_{s'}(\$\mathsf{val}\|\mathsf{in}\|k)$
- statement $:= (\mathcal{P}, \mathsf{MT.root}, \mathsf{sn}, \mathsf{cm})$
- witness $:= (\mathsf{coin}, \mathsf{sk}_{\mathtt{prf}}, \mathsf{branch}, s, \$\mathsf{val}, \mathsf{in}, k, s')$
- $\pi := \mathsf{NIZK.Prove}(\mathcal{L}_{\texttt{FREEZE}}, \mathsf{statement}, \mathsf{witness})$
- send $(\texttt{freeze}, \pi, \mathsf{sn}, \mathsf{cm})$ to $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$
- store in, cm, $\$val, s', and k to use later (in `compute`)

Compute: On input $(\texttt{compute})$ as party $\mathcal{P}$:
- assert current time $T_1 \leq T < T_2$
- sample encryption randomness r
- $\mathsf{ct} := \mathsf{ENC}(\mathcal{P}_{\mathcal{M}}.\mathsf{epk}, r, (\$\mathsf{val}\|\mathsf{in}\|k\|s'))$
- $\pi := \mathsf{NIZK.Prove}((\mathcal{P}_{\mathcal{M}}, \mathsf{cm}, \mathsf{ct}), (\$\mathsf{val}, \mathsf{in}, k, s', r))$
- send $(\texttt{compute}, \pi, \mathsf{ct})$ to $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$

Finalize: Receive $(\texttt{finalize}, \mathsf{coin}, \mathsf{ct})$ from $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$:
- decrypt $(s\|\$\mathsf{val}) := \mathsf{SDEC}_k(\mathsf{ct})$
- store $(s, \$\mathsf{val}, \mathsf{coin})$ in Wallet
- output $(\texttt{finalize}, \$\mathsf{val})$

Protocol for manager $\mathcal{P}_{\mathcal{M}}$:

Compute: On receive $(\texttt{compute}, \mathcal{P}_i, \mathsf{ct})$ from $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$:
- decrypt and store $(\$\mathsf{val}_i\|\mathsf{in}_i\|k_i\|s_i) := \mathsf{DEC}(\mathsf{esk}, \mathsf{ct})$
- store $\mathsf{cm}_i := \mathsf{Comm}_{s_i}(\$\mathsf{val}_i\|\mathsf{in}_i\|k_i)$
- output $(\mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$
- If this is the last `compute` received:
  - for $i \in [N]$ such that $\mathcal{P}_i$ has not called `compute`, $(\$\mathsf{val}_i, \mathsf{in}_i, k_i, s_i, \mathsf{cm}_i) := (0, \perp, \perp, \perp, \perp)$
  - $(\{\$\mathsf{val}'_i\}_{i\in[N]}, \mathsf{out}) := \phi_{\mathrm{priv}}(\{\$\mathsf{val}_i, \mathsf{in}_i\}_{i\in[N]}, \mathsf{in}_{\mathcal{M}})$
  - store and output $(\{\$\mathsf{val}'_i\}_{i\in[N]}, \mathsf{out})$

Finalize: On input $(\texttt{finalize}, \mathsf{in}_{\mathcal{M}}, \mathsf{out})$:
- assert current time $T \geq T_2$
- for $i \in [N]$:
  - sample a commitment randomness s'_i
  - $\mathsf{coin}'_i := \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$
  - $\mathsf{ct}_i := \mathsf{SENC}_{k_i}(s'_i\|\$\mathsf{val}'_i)$
- statement $:= (\mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{cm}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$
- witness $:= \{s_i, \$\mathsf{val}_i, \mathsf{in}_i, s'_i, k_i\}_{i\in[N]}$
- $\pi := \mathsf{NIZK.Prove}(\mathsf{statement}, \mathsf{witness})$
- send $(\texttt{finalize}, \pi, \mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{coin}'_i, \mathsf{ct}_i\})$ to $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$

$\mathsf{UserP}_{\mathrm{cash}}$: include $\mathsf{UserP}_{\mathrm{cash}}$.

Fig. 6. $\mathsf{Blockchain}_{\mathrm{hawk}}$ and $\mathsf{UserP}_{\mathrm{hawk}}$ construction.

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

Technical subtleties. Our $\mathsf{Blockchain}_{\mathsf{cash}}$ 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 [11] appears subtly questionable upon scrutiny, which we elaborate on in the online version [37]. 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.

B. Binding Privacy and Programmable Logic

So far, $\mathsf{Blockchain}_{\mathsf{cash}}$, similar to Zerocash [11], only supports *direct* money transfers between users. We allow transactional privacy and programmable logic simutaneously.

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 $\mathcal{P}$ chooses a private coin $(\mathcal{P}, \mathsf{coin}) \in \mathsf{Coins}$, where $\mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val})$. Using its secret key, $\mathcal{P}$ computes the serial number sn for coin – to be disclosed with the `freeze` operation to prevent double-spending.
- The user $\mathcal{P}$ computes a commitment $(\mathsf{val}||\mathsf{in}||k)$ 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 V.
- The user $\mathcal{P}$ 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 Coins, the sn is correctly constructed, and that the val committed to the contract equals the value of the coin being spent. See $\mathcal{L}_{\text{FREEZE}}$ in Figure 6 for details of the NP statement being proven.

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

$$\mathsf{ct} := \mathsf{ENC}(\mathcal{P}_{\mathcal{M}}.\mathsf{epk}, r, (\$\mathsf{val}\|\mathsf{in}\|k\|s'))$$

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 $\{\mathsf{val}'_i\}_{i\in[n]}$ and public output out by applying the private contract ϕ_{priv}. The manager also constructs a zero-knowledge proof attesting to the outcomes.

Finalize. When the manager submits the outcome of ϕ_{priv} and a zero-knowledge proof of correctness to $\mathsf{Blockchain}_{\mathsf{hawk}}$, $\mathsf{Blockchain}_{\mathsf{hawk}}$ verifies the proof and redistributes the frozen money accordingly. Here $\mathsf{Blockchain}_{\mathsf{hawk}}$ also passes the manager's public input $\mathsf{in}_{\mathcal{M}}$ and public output out to the public Hawk contract ϕ_{pub}. The public contract ϕ_{pub} 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* Comm *is perfectly binding and computationally hiding, the* NIZK *scheme is computationally zero-knowledge and simulation sound extractable, the encryption schemes* ENC *and* SENC *are perfectly correct and semantically secure, the PRF scheme* PRF *is secure, then, our protocols in Figures 5 and 6 securely emulates the ideal functionality $\mathcal{F}(\mathsf{IdealP}_{hawk})$ against a malicious adversary in the static corruption model.*

Proof. Deferred to our online version [37]. □

C. Extensions and Discussions

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 $\mathcal{P}$ simply sends the contract a newly constructed coin $(\mathcal{P}, \mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val}))$ and proves in zero-knowledge that its value $\$\mathsf{val}$ 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 contact), 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 $\mathsf{Ideal}\mathsf{P}_{\text{cash}}$, the `pour` operation discloses the recipient's pseudonyms to the adversary. This means that $\mathsf{Ideal}\mathsf{P}_{\text{cash}}$ only retains full privacy if the recipient generates a fresh, new pseudonym every time. In comparison, Zerocash [11] provides an option of anonymously spending to a long-lived pseudonym (in other words, having $\mathsf{Ideal}\mathsf{P}_{\text{cash}}$ not reveal recipients' pseudonyms to the adversary).

It would be straightforward to add this feature to Hawk as well (at the cost of a constant factor blowup in performance); however, in most applications (e.g., a payment made after receiving an invoice), the transfer is subsequent to some interaction between the recipient and sender.

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. Due to SNARK's preprocessing, right now, each contract instance must declare an upper-bound on the number of participants. An enrollment fee can potentially be adopted to prevent a DoS attack where the attacker joins the contract with many pseudonyms thus preventing legitimate users from joining. How to choose the correct fee amount to achieve incentive compatibility is left as an open research challenge. The a priori upper bound on the number of participants can be avoided if we adopt recursively composable SNARKs [18], [25] or alternative proofs that do not require circuit-dependent setup [16].

V. Adopting SNARKs in UC Protocols and Practical Optimizations

A. Using SNARKs in UC Protocols

Succinct Non-interactive ARguments of Knowledge [12], [33], [49] provide succinct proofs for general computation tasks, and have been implemented by several systems [12], [49], [56]. We would like to use SNARKs to instantiate the NIZK proofs in our protocols — unfortunately, SNARK's security is too weak to be directly employed in UC protocols. Specifically, SNARK's knowledge extractor is non-blackbox and cannot be used by the UC simulator to extract witnesses from statements sent by the adversary and environment — doing so would require that the extractor be aware of the environment's algorithm, which is inherently incompatible with UC security.

UC protocols often require the NIZKs to have simulation extractability. Although SNARKs do not satisfy simulation extractability, Kosba et al. show that it is possible to apply efficient SNARK-lifting transformations to construct simulation extractable proofs from SNARKs [38]. Our implementations thus adopt the efficient SNARK-lifting transformations proposed by Kosba et al. [38].

B. Practical Considerations

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], [49]. 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. 2) Building customized circuit generators to produce SNARK-friendly implementations instead of relying on compilers to translate higher level implementation.

The main cryptographic building blocks in our system are: collision-resistant hash function for the Merkle trees, pseudorandom function, commitment, and encryption. Our implementation supports both 80-bit and 112-bit security levels. To instantiate the CRH efficiently, we use an Ajtai-based SNARK-friendly collision-resistant hash function that is similar to the one used by Ben-Sasson et al. [14]. In our implementation, the modulus q is set to be the underlying SNARK implementation 254-bit field prime, and the dimension d is set to 3 for the 80-bit security level, and to 4 for the 112-bit security level based on the analysis in [38]. For PRFs and commitments, we use a hand-optimized implementation of SHA-256. Furthermore, we adopt the SNARK-friendly primitives for encryption used in the study by Kosba et al. [38], in which an efficient circuit for hybrid encryption in the case of 80-bit security level was proposed. The circuit performs the public key operations in a prime-order subgroup of the Galois field extension $\mathbb{F}_{p^\mu}$, where $\mu = 4$, p is the underlying SNARK field prime (typically 254-bit prime, i.e. p^μ is over 1000-bit), and the prime order of the subgroup used is 398-bit prime. This was originally inspired by Pinocchio coin [26]. The circuit then applies a lightweight cipher like Speck [10] or Chaskey-LTS [47] with a 128-bit key to perform symmetric encryption in the CBC mode, as using the standard AES-128 instead will result in a much higher cost [38]. For the 112-bit security, using the same method for public key operations requires intensive factorization to find suitable parameters, therefore we use a manually optimized RSA-OAEP encryption circuit with a 2048-bit key instead.

In the next section, we will illustrate how using SNARK-friendly implementations can lead to **2.0-3.7**$\times$ savings in the size of the circuits at the 80-bit security level, compared to the case when naive straightforward implementation are used. We will also illustrate that the performance is also practical in the higher security level case.

Optimizations for `finalize`. In addition to the SNARK-friendly optimizations, we 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 **1.5**× as can be inferred from Figure 9 of Section VI, after SNARK-friendly optimizations are applied.

Optimization 2: Minimize public-key encryption in SNARKs. Second, during `finalize`, the manager encrypts each party $\mathcal{P}_i$'s output coins to $\mathcal{P}_i$'s key, resulting in a ciphertext ct_i. The ciphertexts $\{\mathsf{ct}_i\}_{i\in[N]}$ 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 ct_i's, it would result in relatively large SNARK circuit size. Instead, we rely on a symmetric-key encryption scheme denoted SENC in Figure 6. This requires that the manager and each $\mathcal{P}_i$ perform a key exchange to establish a symmetric key k_i. During an `compute`, the user encrypts this k_i to the manager's public key $\mathcal{P}_{\mathcal{M}}$.epk, and prove that the k encrypted is consistent with the k committed to earlier in cm_i. 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 **1.9**× as shown in Figure 9 of Section VI after applying the previous optimizations. Overall, all optimizations will lead to a gain of more than **10**× in the `finalize` circuit.

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 [18], [25] 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 $O(|\mathsf{in}| + |\mathsf{out}|)$ 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 remark that the CRS for protocol $\mathsf{UserP}_{\mathsf{cash}}$ 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].

Finally, in the future when new primitives become sufficiently fast, it is possible to drop-in and replace our SNARKs with other primtives that do not require per-circuit preprocessing. Examples include recursively composed SNARKs [18], [25] or other efficient PCP constructions [16]. The community's efforts at optimizing these constructions are underway.

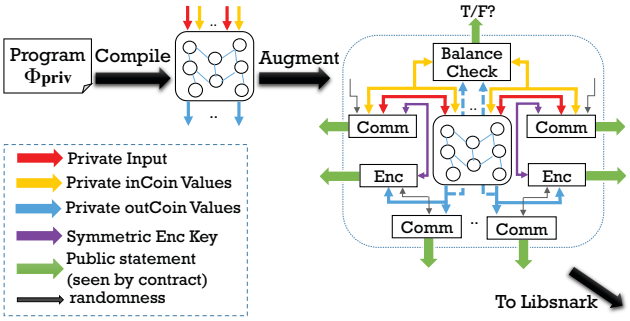

Fig. 7. Compiler overview. Circuit augmentation for `finalize`.

VI. Implementation and Evaluation

A. Compiler Implementation

Our compiler consists of several steps, which we illustrate in Figure 7 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 C language, and is passed as input to the Pinocchio arithmetic circuit compiler [49]. 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 composition of SNARKs.

Circuit Augmentation: After compiling the preprocessed private contract code with Pinocchio, we have an arithmetic circuit representing the input/output relation ϕ_{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.

B. 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 [9]) overcomes the "free-rider problem," allowing a large

TABLE I

Performance of the zk-SNARK circuits for the user-side circuits: `pour`, `freeze` AND `compute` (SAME FOR ALL APPLICATIONS). `MUL` denotes multiple (4) cores, and `ONE` denotes a single core. The `mint` operation does not involve any SNARKs, and can be computed within tens of microseconds. The Proof includes any additional cryptographic material used for the SNARK-lifting transformation.

		80-bit security			112-bit security		
		pour	freeze	compute	pour	freeze	compute
KeyGen(s)	MUL	26.3	18.2	15.9	36.7	30.5	34.6
	ONE	88.2	63.3	54.42	137.2	111.1	131.8
Prove(s)	MUL	12.4	8.4	9.3	18.5	15.7	16.8
	ONE	27.5	20.7	22.5	42.2	40.5	41.7
Verify(ms)		9.7	9.1	10.0	9.9	9.3	9.9
EvalKey(MB)		148	106	90	236	189	224
VerKey(KB)		7.3	4.4	7.8	8.7	5.3	8.4
Proof(KB)		0.68	0.68	0.68	0.71	0.71	0.71
Stmt(KB)		0.48	0.16	0.53	0.57	0.19	0.53

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 crowdfund 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], [17] 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], [17] 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 our online version [37].

C. Performance Evaluation

We evaluated the performance for various examples, using an Amazon EC2 `r3.8xlarge` virtual machine. We assume a maximum of 2^{64} leaves for the Merkle trees, and we present results for both 80-bit and 112-bit security levels. Our benchmarks actually consume at most 27GB of memory and 4 cores in the most expensive case. Tables I and II illustrate the results – we focus on evaluating the zk-SNARK performance since all other computation time is negligible in comparison. We highlight some important observations:

- **On-chain computation** (dominated by zk-SNARK verification time) is very small in all cases, ranging from **9 to 20 milliseconds** The running time of the verification algorithm

TABLE II

Performance of the zk-SNARK circuits for the manager circuit `finalize` for different applications. The manager circuits are the same for both security levels. `MUL` denotes multiple (4) cores, and `ONE` denotes a single core.

		swap	rps	auction		crowdfund	
#Parties		2	2	10	100	10	100
KeyGen(s)	MUL	8.6	8.0	32.3	300.4	32.16	298.1
	ONE	27.8	24.9	124	996.3	124.4	976.5
Prove(s)	MUL	3.2	3.1	15.4	169.3	15.2	169.2
	ONE	7.6	7.4	40.1	384.2	40.3	377.5
Verify(ms)		8.4	8.4	10	19.9	10	19.8
EvalKey(GB)		0.04	0.04	0.21	1.92	0.21	1.91
VerKey(KB)		3.3	2.9	12.9	113.8	12.9	113.8
Proof(KB)		0.28	0.28	0.28	0.28	0.28	0.28
Stmt(KB)		0.22	0.2	1.03	9.47	1.03	9.47

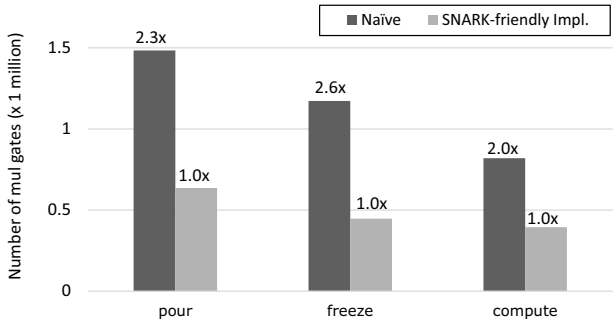

Fig. 8. **Gains of using SNARK-friendly implementation for the user-side circuits:** `pour`, `freeze` and `compute` at 80-bit security.

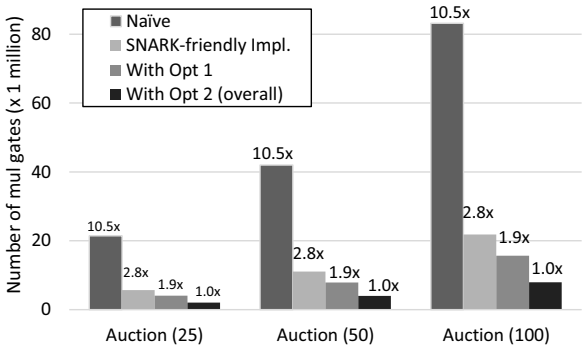

Fig. 9. **Gains after adding each optimization to the `finalize` auction circuit, with 25, 50 and 100 Bidders.** Opt 1 and Opt 2 are two practical optimizations detailed in Section V.

is just linearly dependent on the size of the public statement, which is far smaller than the size of the computation, resulting into small verification time.

- **On-chain public parameters**: As mentioned in Section IV-C, 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 23KB 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 IV-C for more discussions on techniques for performing trusted setup.
- **Manager computation:** Running private auction or crowdfunding protocols with 100 participants requires under 6.5min proof time for the manager on a single core, and under **2.85min** on 4 cores. This translates to under **$0.14** of EC2 time [2].
- **User computation:** Users' proof times for `pour`, `freeze` and `compute` are under one minute, and independent of the number of parties. Additionally, in the worst case, the peak memory usage of the user is less than 4 GB.

Savings from protocol optimizations. Figure 8 illustrates the performance gains attained by using a SNARK-friendly implementation for the user-side circuits, i.e. `pour, freeze` and `compute` w.r.t. the naive implementation at the 80-bit security level. We calculate the naive implementation cost using conservative estimates for the straightforward implementation of standard cryptographic primitives. The figure shows a gain of **2.0-2.6**$\times$ compared to the naive implementation. Furthermore, Figure 9 illustrates the performance gains attained by our protocol optimizations described in Section V The figure considers the sealed-bid auction finalize circuit at different number of bidders. We show that the SNARK-friendly implementation along with our two optimizations combined significantly reduce the SNARK circuit sizes, and achieve a gain of **10**$\times$ relative to a straightforward implementation. The figure also illustrates that the manager's cost is proportional to the number of participants. (By contrast, the user-side costs are independent of the number of participants).

VII. Additional Theoretical Results

Last but not the least, we present additional theoretical results to fruther illustrate the usefulness of our formal blockchain model. In the interest of space, we defer details to the online version [37], and only state the main findings here.

Fair MPC with public deposits in the generic blockchain model. As is well-understood, fairness is in general impossible in plain models of multi-party computation when the majority can be corrupted. This was first observed by Cleve [24] and later extended in subsequent papers [8]. Assuming a blockchain trusted for correctness and availability (but not for privacy), an interesting notion of fairness which we refer to as "financial fairness" can be attained as shown by recent works [7], [17], [40]. In particular, the blockchain can financially penalize aborting parties by confiscating their deposits. Earlier works in this space [7], [17], [40], [50] focus on protocols that retrofit the artifacts of Bitcoin's limited scripting language. Specifically, a few works use Bitcoin's scripting language to construct intermediate abstractions such as "claim-or-refund" [17] or "multi-lock" [40], and build atop these abstractions to construct protocols. Table VII shows that by assuming a generic blockchain model where the blockchain can run Turing-complete programs, we can improve the efficiency of financially fair MPC protocols.

TABLE III
Additional theoretical results for fair MPC with public deposits. The table assumes that N parties wish to securely compute 1 bit of output that will be revealed to all parties at the end. For collateral, we assume that each aborting party must pay all honest parties 1 unit of currency.

	claim-or-refund [17]	multi-lock [40]	**generic blockchain**
On-chain cost	$O(N^2)$	$O(N^2)$	$\mathbf{O(N)}$
# rounds	$O(N)$	$O(1)$	$\mathbf{O(1)}$
Total collateral	$O(N^2)$	$O(N^2)$	$\mathbf{O(N^2)}$

Fair MPC with private deposits. We further illustrate how to perform financially fair MPC using private deposits, i.e., where the amount of deposits cannot be observed by the public. The formal definitions, constructions, and proofs are supplied in the online version [37].

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, CNS-1514261, CNS-1526950, a Sloan Fellowship, three Google Research Awards, Yahoo! Labs through the Faculty Research Engagement Program (FREP) and a NIST award.

References

[1] http://koinify.com.
[2] Amazon ec2 pricing. http://aws.amazon.com/ec2/pricing/.
[3] Augur. http://www.augur.net/.
[4] bitoinj. https://bitcoinj.github.io/.
[5] The rise and rise of bitcoin. Documentary.
[6] Skuchain. http://www.skuchain.com/.
[7] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure Multiparty Computations on Bitcoin. In *S&P*, 2013.
[8] G. Asharov, A. Beimel, N. Makriyannis, and E. Omri. Complete characterization of fairness in secure two-party computation of boolean functions. In *TCC*, 2015.
[9] M. Bagnoli and B. L. Lipman. Provision of public goods: Fully implementing the core through private contributions. *The Review of Economic Studies*, 1989.

[10] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The simon and speck families of lightweight block ciphers. http://ia.cr/2013/404.
[11] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *S&P*, 2014.
[12] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.
[13] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *S&P*, 2015.
[14] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, 2014.
[15] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Security*, 2014.
[16] E. Ben-Sasson and M. Sudan. Short pcps with polylog query complexity. *SIAM J. Comput.*, 2008.
[17] I. Bentov and R. Kumaresan. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO*, 2014.
[18] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *STOC*, 2013.
[19] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*. 2008.
[20] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In S&P, 2015.
[21] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.
[22] R. Canetti. Universally composable signature, certification, and authentication. In *CSF*, 2004.
[23] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC*. 2007.
[24] R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC*, 1986.
[25] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *S & P*, 2015.
[26] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *PETShop*, 2013.
[27] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and mtgox. In *ESORICS*. Springer, 2014.
[28] A. K. R. Dermody and O. Slama. Counterparty announcement. https://bitcointalk.org/index.php?topic=395761.0.
[29] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *FC*, 2014.
[30] C. Fournet, M. Kohlweiss, G. Danezis, and Z. Luo. Zql: A compiler for privacy-preserving data processing. In *USENIX Security*, 2013.
[31] M. Fredrikson and B. Livshits. Zø: An optimizing distributing zero-knowledge compiler. In *USENIX Security*, 2014.
[32] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Eurocrypt*, 2015.
[33] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Eurocrypt*, 2013.
[34] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security*, 2015.
[35] A. Juels, A. Kosba, and E. Shi. The ring of gyges: Using smart contracts for crime. Manuscript, 2015.
[36] A. Kiayias, H.-S. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. http://ia.cr/2015/574.
[37] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. http://ia.cr/2015/675.
[38] A. Kosba, Z. Zhao, A. Miller, H. Chan, C. Papamanthou, R. Pass, abhi shelat, and E. Shi. How to use snarks in universally composable protocols. https://eprint.iacr.org/2015/1093, 2015.
[39] B. Kreuter, B. Mood, A. Shelat, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Security*, 2013.
[40] R. Kumaresan and I. Bentov. How to Use Bitcoin to Incentivize Correct Computations. In *CCS*, 2014.
[41] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *S&P*, 2015.
[42] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *IMC*, 2013.
[43] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *S&P*, 2013.
[44] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *POPL*, 2014.
[45] A. Miller and J. J. LaViola Jr. Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin, 2014.
[46] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. In *FC*, 2001.
[47] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede. Chaskey: An efficient mac algorithm for 32-bit microcontrollers. In *Selected Areas in Cryptography–SAC 2014*, pages 306–323. Springer, 2014.
[48] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf, 2009.
[49] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *S&P*, 2013.
[50] R. Pass and abhi shelat. Micropayments for peer-to-peer currencies. In *CCS*, 2015.
[51] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *S&P*, 2014.
[52] D. Ron and A. Shamir. Quantitative Analysis of the Full Bitcoin Transaction Graph. In *FC*, 2013.
[53] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 1997.
[54] N. van Saberhagen. Cryptonote v 2.0. *https://goo.gl/kfojVZ*, 2013.
[55] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of finance*, 1961.
[56] R. S. Wahby, S. T. V. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.
[57] G. Wood. Ethereum: A secure decentralized transaction ledger. http://gavwood.com/paper.pdf.
[58] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *S&P*, 2003.
[59] G. Zyskind, O. Nathan, and A. Pentland. Enigma: Decentralized computation platform with guaranteed privacy.

APPENDIX A
FREQUENTLY ASKED QUESTIONS

we address frequently asked questions. Some of this content repeats what is already stated earlier, 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.

A. Motivational

"How does Hawk's programming model differ from Ethereum?" Our high-level approach may be superior than Ethereum: Ethereum's language defines the blockchain program, where Hawk allows the programmer to write a single global program, and Hawk auto-generates not only the blockchain program, but also the protocols for users.

"Why not spin off the formal blockchain modeling into a separate paper?" The blockchain formal model 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.

B. Technical

"SNARKs do not offer simulation extractability required for UC." See Section V-A as well as Kosba et al. [38].

SNARK's common reference string. See discussions in Section V-B.

"Why are the recipient pseudonyms $\mathcal{P}_1$ and $\mathcal{P}_2$ revealed to the adversary? And what about Zerocash's persistent addresses feature?" See discussions in Section IV-C.

"Isn't the manager a trusted-third party?" No, our manager is not a trusted third party. As we mention upfront in Sections I-A and I-B, the manager need not be trusted for correctness and input independence. Due to our use of zero-knowledge proofs, if the manager deviates from correct behavior, it will get caught.

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 cryptocurrency as a whole. Therefore, the only thing we trust the manager for is posterior privacy.

As mentioned in Section IV-C we note that one can possibly rely on secure multi-party computation (MPC) to avoid having to trust the manager even for posterier privacy – however such a solution is unlikely to be practical in the near future, especially when a large number of parties are involved. The thereotical formulation of this full-generality MPC-based approach is detailed in the online version [37]. 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 IV-C, 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 prcessor(s). With Hawk's design, the SGX is only very minimally trusted, and is only trusted within the scope of the current contract instance.

APPENDIX B
FORMAL TREATMENT OF PROTOCOLS IN THE BLOCKCHAIN MODEL

We are the first to propose a UC model for the blockchain model of cryptography. 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 the blockchain can 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 blockchain as a party trusted for correctness and availability 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.

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. We allow each identity to generate an arbitrary (polynomial) number of pseudonyms as in Bitcoin and Ethereum.

A. Programs, Functionalities, and Wrappers

To make notations simple for writing ideal functionalities and smart contracts, we make a conscious notational choice of introducing *wrappers*. Wrappers 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 blockchain 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, blockchain and user/manager programs.

Blockchain functionality wrapper $\mathcal{G}$: A blockchain functionality wrapper $\mathcal{G}(\mathsf{B})$ takes in a *blockchain program* denoted B, and produces a blockchain functionality. Our real world protocols will be defined in the $\mathcal{G}(\mathsf{B})$-hybrid world. Our blockchain functionality wrapper is formally presented in Figure 11. We point out the following important facts about the $\mathcal{G}(\cdot)$ wrapper:

- *Trusted for correctness and availability but not privacy.* The bloc kchain functionality wrapper $\mathcal{G}(\cdot)$ stipulates that a blockchain program is trusted for correctness and availability but not for privacy. In particular, the blockchain wrapper exposes the blockchain program'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 $\mathcal{G}(\cdot)$ wrapper captures the following fact. In each round (i.e., block interval), the blockchain 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 blockchain program will then process the messages in this adversary-specified ordering.
- *Rushing adversary.* The blockchain wrapper $\mathcal{G}(\cdot)$ naturally captures a rushing adversary. Specifically, the adversary can first see all messages sent to the blockchain program by honest parties, and then decide its own messages for this round, as well as an ordering in which the blockchain program 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 [11], [27]. For example, in

$\mathcal{F}(\mathsf{idealP})$ **functionality**

Given an ideal program denoted idealP, the $\mathcal{F}(\mathsf{idealP})$ functionality is defined as below:

Init: Upon initialization, perform the following:

Time. Set current time $T := 0$. Set the receive queue $\mathsf{rqueue} := \emptyset$.

Pseudonyms. Set $\mathsf{nyms} := \{(P_1, P_1), \ldots, (P_N, P_N)\}$, i.e., initially every party's true identity is recorded as a default pseudonym for the party.

Ledger. A ledger dictionary structure $\mathsf{ledger}[P]$ 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 $\mathsf{ledger}[]$ to the ideal adversary $\mathcal{S}$.

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

Tick: Upon receiving `tick` from an honest party P: notify $\mathcal{S}$ of (`tick`, P). If the functionality has collected `tick` confirmations from all honest parties since the last clock tick, then

Call the **Timer** procedure of the idealP program.

Apply the adversarial permutation perm to the rqueue to reorder the messages received in the previous round.

For each $(m, \bar{P}) \in \mathsf{rqueue}$ 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 $\mathsf{rqueue} := \emptyset$.

Set $T := T + 1$

Other activations: Upon receiving a message of the form $(m, \bar{P})$ from a party P:

Assert that $(\bar{P}, P) \in \mathsf{nyms}$.

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 $\mathsf{rqueue.add}(m, \bar{P})$.

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

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

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

Ledger operations: *// inner activation*

Transfer: Upon receiving $(\texttt{transfer}, \mathsf{amount}, \bar{P}_r)$ from some pseudonym $\bar{P}_s$:

Notify $(\texttt{transfer}, \mathsf{amount}, \bar{P}_r, \bar{P}_s)$ to the ideal adversary $\mathcal{S}$.

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

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

Expose: On receiving `exposeledger` from a party P, return ledger to the party P.

Fig. 10. The $\mathcal{F}(\mathsf{idealP})$ 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 $\mathcal{F}(\cdot)$ wrapper allows the ideal adversary $\mathcal{S}$ to specify an order 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 $\mathcal{S}$ upon recipient of the message.

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 $\mathcal{F}$: An ideal functionality $\mathcal{F}(\mathsf{idealP})$ takes in an *ideal program* denoted idealP. Specifically, the wrapper $\mathcal{F}(\cdot)$ 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 10.

Protocol wrapper Π: Our protocol wrapper allows us to modularize the presentation of user protocols. Our protocol wrapper is formally presented in Figure 12.

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 $\mathcal{F}(\cdot)$

$\mathcal{G}(\mathsf{B})$ functionality

Given a blockchain program denoted B, the $\mathcal{G}(\mathsf{B})$ functionality is defined as below:

Init: Upon initialization, perform the following:

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

Tick: Upon receiving `tick` from an honest party, if the functionality has collected `tick` confirmations from all honest parties since the last clock tick, then

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

Other activations:

- *Authenticated receive:* Upon receiving a message (`authenticated`, m) from party P:

  Send (m, P) to the adversary $\mathcal{A}$
  Queue the message by calling $\mathsf{rqueue.add}(m, P)$.

- *Pseudonymous receive:* Upon receiving a message of the form $(\texttt{pseudonymous}, m, \bar{P}, \sigma)$ from any party:

  Send $(m, \bar{P}, \sigma)$ to the adversary $\mathcal{A}$
  Parse $\sigma := (\mathsf{nonce}, \sigma')$, and assert $\mathsf{Verify}(\bar{P}.\mathsf{spk}, (\mathsf{nonce}, T, \bar{P}.\mathsf{epk}, m), \sigma') = 1$
  If message $(\texttt{pseudonymous}, m, \bar{P}, \sigma)$ has not been received earlier in this round, queue the message by calling $\mathsf{rqueue.add}(m, \bar{P})$.

- *Anonymous receive:* Upon receiving a message (`anonymous`, m) from party P:

  Send m to the adversary $\mathcal{A}$
  If m has not been seen before in this round, queue the message by calling $\mathsf{rqueue.add}(m)$.

Permute: Upon receiving (`permute`, perm) from the adversary $\mathcal{A}$, record perm.
Expose: On receiving `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 $(\texttt{transfer}, \mathsf{amount}, \bar{P}_r)$ from some pseudonym $\bar{P}_s$:

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

Fig. 11. The $\mathcal{G}(\mathsf{B})$ functionality is parameterized by a blockchain program denoted B. The $\mathcal{G}(\cdot)$ 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 $\mathcal{G}(\cdot)$ wrapper allows the adversary to specify an ordering to the messages received by the blockchain program in one round.

and $\mathcal{G}(\cdot)$), we obtain functionalities with well-defined formal meanings. Programs can also be wrapped by a protocol wrapper Π to obtain a full protocol with formal meanings.

B. Modeling Time

At a high level, we express time in a way that conforms to the Universal Composability framework [21]. In the ideal world execution, time is explicitly encoded by a variable T in an ideal functionality $\mathcal{F}(\mathsf{idealP})$. In the real world execution, time is explicitly encoded by a variable T in our blockchain functionality $\mathcal{G}(\mathsf{B})$. 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 $\mathcal{F}(\mathsf{idealP})$ in the ideal-world execution, or to the $\mathcal{G}(\mathsf{B})$ functionality in the real-world execution. On collecting "tick" messages from all honeset parties, the $\mathcal{F}(\mathsf{idealP})$ or $\mathcal{G}(\mathsf{B})$ functionality would then advance the time $T := T + 1$. The functionality also allows parties to query the current time T.

When multiple messages arrive at the blockchain in a time interval, we allow the adversary 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.

C. 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

$\Pi(\mathsf{UserP})$ **protocol wrapper in the** $\mathcal{G}(\mathsf{B})$**-hybrid world**

Given a party's local program denoted prot, the $\Pi(\mathsf{prot})$ functionality is defined as below:

Pseudonym related:

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

Send: Upon receiving internal call (`send`, m, $\bar{P}$):

If $\bar{P} == P$: send (`authenticated`, m) to $\mathcal{G}(\mathsf{B})$. *// 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 $\mathcal{G}(\mathsf{B})$. Compute $\sigma' := \mathsf{Sign}(\mathsf{ssk}, (\mathsf{nonce}, T, \mathsf{epk}, m))$ where ssk is the recorded secret signing key corresponding to $\bar{P}$, nonce is a freshly generated random string, and epk is the recorded public encryption key corresponding to $\bar{P}$. Let $\sigma := (\mathsf{nonce}, \sigma')$.

Send $(\texttt{pseudonymous}, m, \bar{P}, \sigma)$ to $\mathcal{G}(\mathsf{B})$.

AnonSend: Upon receiving internal call (`anonsend`, m, $\bar{P}$): send (`anonymous`, m) to $\mathcal{G}(\mathsf{B})$.

Timer and ledger transfers:

Transfer: Upon receiving input (`transfer`, \$amount, $\bar{P}_r$, $\bar{P}$) from the environment $\mathcal{E}$:

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

Send $\big(\texttt{transfer}, \$\mathsf{amount}, \bar{P}_r\big)$ to $\mathcal{G}(\mathsf{B})$ as pseudonym $\bar{P}$.

Tick: Upon receiving `tick` from the environment $\mathcal{E}$, forward the message to $\mathcal{G}(\mathsf{B})$.

Other activations:

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

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 $\mathcal{E}$, or any other message from a party: Pass the input/message to the party's local program to process.

Fig. 12. Protocol wrapper.

(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, and the corresponding private key is known by the party who "owns" the pseudonym. The blockchain functionality allows parties to publish authenticated messages that are bound to a pseudonym of their choice. Thus each interaction with the blockchain program 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 blockchain program. 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 blockchain program (e.g., "auction manager"). The alternative is to assign roles on a "first-come first-served" basis (e.g., as the bidders in an auction). To this end, we allow each party to define generate a single "default" pseudonym which is publicly-bound to their real identity. We allow applications to make use of this through a convenient abuse of notation, by simply using a party identifier as a parameter or hardcoded string. Strictly speaking, the pseudonym string is not determined until the "gennym" subroutine is executed; the formal interpretation is that whenever such an identity is used, the default pseudonym associated with the identity is fetched from the blockchain program. (This approach is effectively the same as taken by Canetti [22], where a functionality $\mathcal{F}_{\mathtt{CA}}$ allows each party to bind their real identity to a single public key of their choice).

Additional appendices are supplied in the online full version [37].