smartFHE: Privacy-Preserving Smart Contracts from Fully Homomorphic Encryption

Ravital Solomon[1] and Ghada Almashaqbeh[2]

[1] NuCypher, ravital@nucypher.com
[2] University of Connecticut, ghada.almashaqbeh@uconn.edu

Abstract. Despite the great potential and flexibility of smart contract-enabled blockchains, building privacy-preserving applications using these platforms remains an open question. Existing solutions fall short in achieving this goal since they support a limited operation set, enable private computation on inputs belonging to only one user, or even ask the users themselves to coordinate and perform the computation off-chain.
To address these limitations, we propose *smartFHE*, a framework to support private smart contracts using fully homomorphic encryption (FHE). To the best of our knowledge, smartFHE is the first to use FHE in the blockchain model; it is also the first to allow for building arbitrary smart contracts that operate on multiple users' inputs on-chain while preserving input/output privacy. smartFHE does not overload the user since miners are instead responsible for performing the private computation. This is achieved by employing (single and multi-key) FHE so miners can compute over encrypted data and account balances, along with efficient zero-knowledge proof systems (ZKPs) so users can prove well-formedness of their private inputs. Crucially, our framework is *modular* as any FHE and ZKP scheme can be used so long as they satisfy certain requirements with respect to correctness and security. We formulate a notion for a privacy-preserving smart contract (PPSC) scheme and show a concrete instantiation. We provide formal definitions along with proofs of the correctness and security of our construction. Finally, we include preliminary benchmarks to evaluate the feasibility of our instantiation.

1 Introduction

Cryptocurrency can be traced back to (at least) 1983 when Chaum first proposed the concept of *electronic cash* using blind signatures [17]. Extending Chaum's design, Bitcoin [39] removed the need for a trusted party and introduced the notion of a public distributed ledger called blockchain through which users could exchange currency directly with one another. Unfortunately, Bitcoin provides no privacy for the user as transaction records are fully public on the blockchain [8]; thus, several initiatives emerged to bring privacy to currency transfer [43,40].

Smart contracts and privacy. In parallel to the development of a private cryptocurrency, a very different question about Bitcoin's functionality was asked. Could Bitcoin be extended to support arbitrary user-defined applications? The

answer was *yes* but with major changes to its UTXO-based design. Thus, Ethereum was born, defining an account-based model and a Turing-complete scripting language that permit users to deploy arbitrary programs called smart contracts [46]. Although Ethereum offers a highly expressive functionality, it provides no privacy out of the box.

Over the last few years, several attempts have been made to support private computations for a single user's inputs over a blockchain. Some constructions built upon the paradigm used for private currency transfer—operating directly on additively homomorphic encryptions with ZKPs used to prove that certain relations hold on the inputs [14]. Unfortunately, additive homomorphisms enable only a limited set of applications with input/output (I/O) privacy. Other constructions offload *all* work to the user to do offline [45,12]. Users perform the intended computations on plaintext data, encrypt inputs and outputs, and create a ZKP certifying correctness of computation with respect to these encryptions. Blockchain miners only verify correctness of the ZKP. We call this approach the *pure ZKP approach* as it relies on the power of ZKPs to perform computations with I/O privacy. Even worse, this paradigm cannot scale to enable privacy-preserving computation on multi-user inputs without utilizing (usually) highly-interactive MPC protocols in which the users would be responsible for coordinating the entire computation off-chain themselves.[3]

1.1 Our Contributions

To address these limitations, we propose smartFHE, a framework for building smart contracts that preserve I/O privacy for an arbitrary number of users. Operating directly on encrypted values has proven invaluable across numerous applications [16,22]. Existing private smart contract schemes may have chosen to abandon this approach as it did not yield practical results in the short-term; however, such a viewpoint may also be short-sighted. Supporting additive and multiplicative homomorphisms on ciphertexts leads us to the holy grail of fully homomorphic encryption (FHE). Using FHE, users could supply encrypted inputs along with a simple ZKP showing well-formedness of the initial ciphertexts and that certain relations on the plaintexts are satisfied. Miners check the proofs and then perform the requested computations directly on the encrypted inputs. No need for users to remain online or perform any off-chain coordination for the private computation. Also no need for the users to provide complex ZKPs attesting to the correctness of the entire computation.

We observe that combining FHE with blockchain represents a *harmonious union*. Blockchain addresses the pain point of verifying correctness of homomorphic computation. In FHE, the evaluation party is *different* from the (encrypted) input owner and there is no immediate way for the input owner to

[3] Several works dealt with MPC in a non-interactive setting; users submit their inputs to an output-producing party (that is available all the time), then this party computes the intended functionality output. However, these works leak the residual function [31], require a setup assumption such as pre-dealt correlated randomness [25] or a PKI [31], or even rely on indistinguishability obfuscation [30].

validate correctness of the computation (without repeating the entire computation). Although there are solutions to this problem, the added cost can be prohibitive [27]. A blockchain offers a simpler solution through consensus—that is, the assumption that the majority of the mining power is honest provides guarantees with regards to correctness [36]. Moreover, a blockchain solves another problem for computation outsourcing (as in the FHE setting)—namely, the need for an always-available evaluation party. Miners can perform paid computations for users (as in Ethereum) which could be the FHE computations needed in private smart contracts.

We take a foundational approach to realizing smart contracts with I/O privacy employing FHE and ZKPs. Using FHE, we investigate two flavours. The *single-key* FHE approach readily supports private computations on inputs belonging to the same user, whereas the *multi-key* FHE approach addresses the multi-user input problem. To the best of our knowledge, smartFHE is the first to use FHE in the blockchain model; it is also the first to support building arbitrary smart contracts that operate on multiple users' inputs *on-chain* while preserving input/output privacy. We elaborate on our contributions in what follows.

A notion for privacy-preserving smart contracts. We define a notion for privacy-preserving smart contracts (PPSCs) capturing the support of arbitrary computation with multi-user I/O privacy. Furthermore, we extend existing definitions of correctness and security [14,43], in terms of privacy/ledger indistinguishability and overdraft safety/balance, to provide formal guarantees for a PPSC scheme. We believe that our PPSC definition is of independent interest as it is general enough to be used in other private smart contract constructions.

smartFHE framework. We propose smartFHE, a framework to support smart contracts with I/O privacy along with payments that hide the users' balances and transfer amount, via FHE and ZKPs. smartFHE preserves privacy under the same decentralization, availability, and work model of general-purpose (public) smart-contract systems. smartFHE does not overload end users as miners are responsible for executing the required computations. To allow for operations on encrypted account states, smartFHE introduces a locking mechanism (reminiscent of a mutex) to solve resulting concurrency issues. It also protects against front-running and replay attacks.

smartFHE is highly flexible with respect to functionality. First, it offers two modes of operation—public and private—that users can switch between. Private accounts and their data are stored encrypted on the blockchain and users supplement any additional encrypted inputs with proper ZKPs attesting to well-formedness. FHE allows miners to operate directly on these private inputs, produce private outputs, and update the blockchain state accordingly. When using a multi-key FHE scheme, these computations can be performed over multi-user inputs, making our framework the first to offer such capability on-chain. Second, our framework is *modular* since it is not bound to particular FHE and ZKP schemes, allowing us to exploit future developments in these areas.

smartFHE is highly versatile with respect to applications. When using a multi-key FHE scheme, one can implement any functionality, with any number of users, as a private smart contract. Even the simpler (and more efficient) single-key FHE approach allows us to realize private payments and arbitrary computations over a single user's inputs. However, with additional smart contract logic (as shown in Appendix D), it can even realize some important applications operating on multi-user inputs such as multi-item sealed-bid auctions and inventory tracking.

smartFHE instantiation. We provide two instantiations of our framework; one using a single-key FHE scheme and another with a multi-key FHE scheme. Combining FHE with ZKPs is non-trivial if we require efficiency and fairly small proofs sizes. The most obvious path forward to proving the lattice-based relations of FHE is via lattice-based ZKPs. However, state-of-the-art lattice-based ZKPs [10,9] tend to be hundreds of kilobytes in size and are not nearly as efficient as recently proposed elliptic curve-based ZKPs [15,19]. To address this challenge, we utilize a recent elliptic-curve based ZKP system [42] that allows for proving certain lattice-based relations. To take advantage of this ZKP, we must choose an FHE scheme whose security is based on the hardness of Ring Learning with Errors (RLWE), such as the BGV or BFV scheme, for optimal performance. As [42] proves relations with respect to a Pedersen commitment, we can employ another elliptic-curve based ZKP—Bulletproofs [15]—to prove further relations over private inputs quite efficiently. Separately, we are able to re-use randomness when encrypting values under the sender and receiver's key by observing that this is indeed secure via the generalized leftover hash lemma for improved performance [23]. Our multi-key instantiation uses the recent multi-key FHE scheme of [38] since it is compatible with the discrete log proofs scheme [42], albeit with a loss in efficiency since its hardness is based on Learning with Errors (LWE).

To show feasibility, we provide preliminary benchmarks to evaluate the performance of our single-key instantiation. Perhaps surprisingly, these benchmarks indicate that private transactions and private smart contracts can achieve superior performance (in terms of execution time) on the user's end compared to current state-of-the-art schemes including Zexe [12] and Zkay [45].

1.2 Related Work

Several works have explored privacy in the context of blockchain. We focus on those peer-reviewed works that provide I/O privacy for arbitrary computation (rather than customized solutions for specific use cases).

Hawk [36] was one of the first works to construct a private smart contract scheme using ZKPs. Hawk requires a semi-trusted manager—trusted with protecting the privacy of the users' inputs but not for correctness of computation. Ekiden [18] replaces a semi-trusted manager with trusted hardware. Subsequent works avoid such (semi-)trusted parties or hardware. Among them, Zether [14] targets smart contract privacy for Ethereum. Its reliance on additively homomorphic encryption (that operates on single user inputs) restricts its functionality to private currency transfer and a limited class of private smart contracts. Although

Zether supports anonymity, this feature cannot be implemented on Ethereum as the cost exceeds the gas limit per block [14]. Extending smartFHE to provide anonymity is a direction for our future work.

Zkay [45] takes a different approach and proposes a compiler for private smart contracts. It defines a language to write smart contracts with specific syntax to define private data, which can then be compiled to smart contracts with the required code to handle private I/O. Zkay follows the pure ZKP approach described earlier, so it overloads end users and requires off-chain coordination to handle multi-user inputs. Concerningly, it does not address concurrency issues related to operating on private (encrypted) accounts. Nonetheless, we view Zkay's compiler idea as compatible with our smartFHE framework; we can use it to implement automatic conversion of smart contract code into public or private format based on the types of accounts referenced in the code.

Zexe [12] takes privacy further by also preserving function privacy (i.e. hiding the computation itself). Following the pure ZKP approach outlined previously, Zexe operates in the UTXO-based model which restricts the supported functionality to extending Zerocash scripts used to spend currency. Thus, it does not truly support private smart contracts. Furthermore, it will scale poorly if the ZKP is used to attest to changes in the contract state. Kachina [34] seeks to solves this problem by introducing state oracles to reduce the ledger state size involved in a ZKP. Following the pure ZKP approach, it formally defines and models private smart contracts.

2 Defining a PPSC Scheme

In this section, we define a notion for a privacy-preserving smart contract (PPSC) scheme and formulate its correctness and security.

Notation. We use λ to represent the security parameter, and pp to denote the system's public parameters. To refer to parameter x inside pp, we write $\mathsf{pp.x}$. The public and secret keys of an account are denoted pk and sk, respectively, with the account owner in superscript and the account type (public or private) in subscript. Lastly, PPT means probabilistic polynomial time.

PPSC definition. We envision a PPSC scheme applied on top of a public smart contract-enabled cryptocurrency (such as Ethereum). It can be viewed as the extensions needed to support privacy-preserving execution of smart contracts and payments on an account-based ledger. Hence, a PPSC scheme inherits all the public functionality and data structures found in the underlying public system. This includes the append-only ledger $\mathcal{L}$ that stores states for accounts (e.g. their balances and contract code if applicable). Users have access to this ledger at any time and can initiate basic currency transfer transactions or deploy arbitrary smart contracts. Processing transactions and performing computations (the code portions of smart contracts) change the state of the ledger, where such changes are applied when a new block is mined. Thus, issuing any transaction or implementing any code relies on the latest ledger state (i.e. the latest changes

reflected by the most recent mined block). In our definition below, we focus on the new modules needed to support private transactions and smart contract execution with private inputs and outputs.

Definition 1 (PPSC Scheme). *A PPSC scheme Π is a tuple of* PPT *algorithms (*Setup, CreateAccount, CreateTransaction, VerifyTransaction, Compute, UpdateState*) defined as follows:*

- Setup*: Takes as input a security parameter λ. Outputs public parameters* pp.
- CreateAccount*: Takes as inputs public parameters* pp *and a privacy mode (private or public). It generates a key pair* $(\mathsf{sk}, \mathsf{pk})$ *and an address* addr *(derived from* pk*) with a postfix indicating if it is for a private or public account. It also initializes the account state consisting of a balance* $\mathsf{Bal}[\mathsf{pk}] = 0$ *associated with the account, and a lock entry* $\mathsf{Lk}[\mathsf{pk}] = \perp$ *indicating the address to which the account is locked ($\perp$ means the account is unlocked). Finally,* CreateAccount *outputs the key pair, address, and state.*
- CreateTransaction*: Takes as inputs public parameters* pp*, transaction semantics, syntax, and information. Outputs a transaction* tx *of one of the following types:*
  - $\mathsf{tx}_{\mathsf{shield}}$: *Transfers currency from a public account to a private account. The transfer amount is public.*
  - $\mathsf{tx}_{\mathsf{deshield}}$: *Transfers currency from a private account to a public account. The transfer amount is public.*
  - $\mathsf{tx}_{\mathsf{privtransf}}$: *Transfers currency from one private account to another private account. The transfer amount is private.*
  - $\mathsf{tx}_{\mathsf{lock}}$: *Locks a private account to some other account, thereby transferring account ownership to the recipient and preventing the locked account balance from being altered until unlocked.*
  - $\mathsf{tx}_{\mathsf{unlock}}$: *Unlocks a private account, returning control back to its owner. The transaction is only successful if it is issued by the same account to which the private account was locked.*
- VerifyTransaction*: Takes as inputs public parameters* pp*, transaction* tx*, and the transaction's syntax/semantics for the types mentioned above. Outputs 1 if* tx *is valid and 0 otherwise.*
- Compute*: Takes as inputs public parameters* pp*, a circuit C representing the requested computation, and inputs $x_1, ..., x_n$. If $x_1, ...x_n$ are public, then apply C as is on these inputs. If $x_1, ..., x_n$ are private, transform C into a functionality-equivalent circuit C' operating on private inputs and producing private outputs,*[4] *then apply C' to $x_1, ..., x_n$. If computation is successful, output 1. Otherwise, output 0.*
- UpdateState*: Takes as inputs public parameters* pp*, current ledger state $\mathcal{L}$, and a list of pending operations* $\mathsf{Ops} = \{\mathsf{op}_i\}$ *such that* op_i *can be a transaction* tx_i *or a computation* $\mathsf{Compute}(\mathsf{pp}, C_i, \{x_{i,1}, \ldots, x_{i,n}\})$ *as described above.* UpdateState *proceeds in blocks and epochs (an epoch is k consecutive blocks). Changes induced by all operations are reflected at the end of a*

[4] So for any public input x and its private version x', we have $C(x) = C'(x')$.

block except for $\mathsf{tx}_{\mathsf{shield}}$ *or* $\mathsf{tx}_{\mathsf{privtransf}}$*, which are processed at the end of the epoch (i.e. the last block in an epoch). Incoming transactions to a locked account will not be processed until the next epoch after which the account is unlocked. At the end of a block, output an updated state* $\mathcal{L}'$*.*

Correctness and Security. Due to space limitations, we provide only an overview of our correctness and security notions (which are inspired by [43,14]); formal definitions can be found in Appendix C. Intuitively, correctness of a PPSC scheme requires that if we start with a valid ledger state and apply an arbitrary sequence of valid (or honestly generated) operations, the resulting state will also be valid. Towards this end, we define what constitutes a valid operation list, ledger state evolution, and an incorrectness game, INCORR, in which a challenger $\mathcal{C}$ and a ledger sampler $\mathcal{S}$ interact with each other. A PPSC scheme is correct so long as the advantage of $\mathcal{S}$ in winning the INCORR game is negligible.

With respect to security, we define two requirements for a PPSC scheme: ledger indistinguishability and overdraft safety. To this end, we define a common security game between a challenger $\mathcal{C}$, representing the honest users, and an adversary $\mathcal{A}$. Both interact with the PPSC oracle $\mathcal{O}_{\mathsf{PPSC}}$. The adversary can ask $\mathcal{C}$ to perform various user algorithms. $\mathcal{A}$ can also submit his own operations to $\mathcal{O}_{\mathsf{PPSC}}$ for processing and request arbitrary subsets of pending operations to be processed. Informally, ledger indistinguishability ensures that the ledger produced by a PPSC scheme Π does not reveal additional information beyond what was publicly revealed, whereas overdraft safety ensures that a PPSC scheme Π does not allow an adversary to spend more currency than he owns.

3 smartFHE Framework and Instantiations

In this section, we present the design of smartFHE, a PPSC framework that uses FHE in the blockchain model, along with an overview of concrete instantiations. We begin by outlining the cryptographic building blocks employed, then we describe the smart contract-enabled cryptocurrency architecture that we target, followed by technical details of our framework and instantiations.

Cryptographic building blocks. FHE constructions fall under two categories; single-key FHE allows for arbitrary computation over data encrypted under the same key, whereas multi-key FHE allows for arbitrary computation over data encrypted under different keys. All currently known schemes rely on lattice-based cryptography, thus providing post-quantum security guarantees. Additionally, FHE schemes model computation in one of three ways—as boolean circuits, arithmetic circuits, or floating point arithmetic [32]. Floating point arithmetic will provide only approximate values and, thus, is a poor choice for smartFHE since we need precise balance and transfer amounts. In our instantiations, we use the BGV single-key scheme [13] which models computation as arithmetic circuits and then the Mukherjee-Wichs multi-key scheme [38] which models computation as boolean circuits. In BGV, the message, ciphertext, and the secret key are vectors over the quotient ring $R = \mathbb{Z}_q(x)/(f(x))$ (where $f(x) = x^d + 1$ and d is

a power of 2) whereas the public key takes the form of a matrix over R. The Mukherjee-Wichs scheme follows the same format but instead the vectors and matrices are over $\mathbb{Z}_q$. Both of these FHE schemes are leveled, meaning that only a certain number of homomorphic multiplications can be performed sequentially (although bootstrapping can be used to solve this), often with different keys associated for each level.

We chose the above FHE schemes as they rely on the hardness of Ring-LWE and LWE [37], respectively, so we can use the short discrete log proofs construction [42]. This proof system allows us to fairly efficiently prove knowledge of a short vector $\vec{\mathbf{s}}$ such that $\mathbf{A}\vec{\mathbf{s}} = \vec{\mathbf{t}}$ for public $\mathbf{A}$ and $\vec{\mathbf{t}}$ over the polynomial ring $\mathbb{Z}_q[X]/(g(x))$, where $g(x)$ is a monic, irreducible polynomial of degree d in $\mathbb{Z}[X]$. Such a relation will allow users to attest to the well-formedness of FHE ciphertexts (which are the users' encrypted inputs). Additional details on the building blocks are provided in Appendix A.

3.1 Architecture

Our framework can be viewed as extending a public smart contract-enabled cryptocurrency to support privacy. We require an account-based model, a Turing-complete scripting language, and a virtual machine with a cost (i.e. miners' fees) associated with each smart contract operation. Thus, we consider an Ethereum-like architecture (an overview of Ethereum can be found in Appendix A).

smartFHE supports four services: public payments, public smart contracts, private payments, and private smart contracts. The default operation is the *public* mode—meaning that everything will be logged in the clear on the blockchain and a smart contract code will operate on public inputs/outputs. These are handled in the same manner as in Ethereum. On the other hand, if the smart contract (or a payment transaction) operates on private accounts, then the *private* mode will be used instead. The required operations will be converted into their equivalent privacy-preserving format and will produce private outputs (for simplicity, we refer to these as private smart contracts). smartFHE extends the standard transaction set of Ethereum with new types of transactions and cryptographic capabilities to permit operations on private accounts and user inputs.

Similar to Ethereum, smartFHE has two types of accounts: contract owned and externally (or user) owned. However, we further subdivide externally owned accounts into two types: *public* and *private*. Private accounts will be used to initiate private transactions and participate in private smart contracts. In our scheme, each account (public or private) will maintain its own nonce which must be signed and incremented as part of any transaction this account issues. This approach ensures that valid transactions cannot be replayed and zero-knowledge proofs cannot be maliciously imported into new transactions.

smartFHE operation proceeds in rounds (a round is the time needed to mine a block on the blockchain) and epochs (where an epoch is y contiguous rounds for some integer y that is selected during the system setup phase). The latter is needed to handle concurrency issues related to operating on private accounts,

as will be shown later. If desired, epochs can be eliminated entirely (which we discuss at the end of this section).

3.2 The smartFHE Protocol

Our framework is composed of three components: a global setup phase to deploy the system, a network protocol defining all extensions required to support private payments/smart contracts, and a mechanism for handling concurrency issues resulting from operating on private accounts.

3.2.1 Setup This includes setup related to the system, user, and smart contracts. System setup involves launching the PPSC system—which starts with deploying miners, creating the genesis block of its blockchain, and generating all public parameters pp needed by the cryptographic primitives (such as FHE and ZKP) that we employ in the system. The public parameters will be known to everyone and could either be published in the genesis block or announced and maintained off-chain. Once system setup is complete, users can now join and create their own public and/or private accounts. Smart contract setup is dependent on the creator of its code. This code will specify the sorts of (private or public) inputs the contract functions will accept, along with the operations to be performed on these inputs. Once the creator deploys the contract on the blockchain, users can invoke its functionality and pass in their inputs to be operated on.

3.2.2 Network Protocol Syntax In what follows, we informally present the syntax that smartFHE adds to Ethereum's network protocol to support privacy. We do this using the single-key FHE-based design (full syntax and technical details can be found in Appendix B). Given the modularity of smartFHE's design, this syntax will be re-used for the multi-key FHE-based design with minimal changes that we cover later in the section.[5]

(1) Public operations via public accounts. To create a public account, the user calls $\mathsf{Pub.CreateAccount}(\mathsf{pp})$ to generate the account key pair $(\mathsf{pk}_{\mathsf{pub}}, \mathsf{sk}_{\mathsf{pub}})$. The public key defines the user's account address while the secret key allows her to sign all transactions issued by this account. Each public account also has an unencrypted balance $\mathsf{balance}$ and a nonce $\mathsf{ctr}[\mathsf{pk}_{\mathsf{pub}}]$ associated with it. smartFHE handles public operations (both payments and smart contracts) in the same manner as Ethereum. The algorithm $\mathsf{Transfer}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{pub}}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{pub}}, \mathsf{amnt})$ allows a user to send amnt currency from one public account to another. The syntax for invoking functions in a smart contract is defined by the contract creator. As in Ethereum, invoking a function is done by issuing a transaction that contains all inputs this function needs.

[5] Users must sign all (public and private) transactions they issue and miners must verify these signatures before accepting any of these transactions. We omit repeating this fact and the corresponding syntax in this section.

(2) Private payments via private accounts. Having a private account in smartFHE allows its owner to initiate transactions that hide transfer values and/or users' balances.

To create a private account, the user calls $\mathsf{Priv.CreateAccount}(\mathsf{pp})$ to generate the FHE account key pair $(\mathsf{pk}_{\mathsf{priv}}, \mathsf{sk}_{\mathsf{priv}})$ (which is used to encrypt her inputs) along with a signature scheme key pair $(\mathsf{sigpk}_{\mathsf{priv}}, \mathsf{sigsk}_{\mathsf{priv}})$ to sign outgoing transactions.[6] A private account has an encrypted balance (with respect to $\mathsf{pk}_{\mathsf{priv}}$) and a nonce $\mathsf{ctr}[\mathsf{pk}_{\mathsf{priv}}]$ associated with it. Users can initiate the following private transaction types:

- $\mathsf{tx}_{\mathsf{shield}} \leftarrow \mathsf{Shield}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{pub}}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{amnt})$: Transfers a given amount of currency from a public account to a private one. Thus, the transaction contains the public keys of the sender's public account and the recipient's private account, and the (unencrypted) transfer amount. No ZKP is needed to prove that the sender owns the right transfer amount. The sender uses a public account and can be verified by simply tracking the account's public state on the blockchain.
- $\mathsf{tx}_{\mathsf{privtransf}} \leftarrow \mathsf{PrivTransfer}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{amnt})$: Transfers some undisclosed (encrypted) amount of currency from one private account to another private account. This transaction requires a ZKP that the sender owns the transferred currency, that the same amount has been added to the recipient's account as has been deducted from the sender's account, that the transfer amount is positive, and that the sender's remaining balance is non-negative.
- $\mathsf{tx}_{\mathsf{deshield}} \leftarrow \mathsf{Deshield}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{pub}}, \mathsf{amnt})$: Transfers a given amount of currency from a private account to a public one. This transaction needs a ZKP to prove that the sender's account has a balance equals to at least the transfer value. Note that such an interaction reveals some information about the private account (i.e. sender has a balance larger than or equal to the released value) since the recipient's account is public.

(3) Private smart contracts. Users can write smart contracts with code operating on their private data and private account balances. This code will be translated to an arithmetic or boolean circuit depending on the type of FHE scheme used. Since the code may operate on encrypted values, users participating in the contract need to provide ZKPs showing that their initial ciphertexts are well-formed and satisfy certain conditions (dependent on the application). Miners (of which a majority are trusted for correctness and availability in the blockchain model) will check these ZKPs, perform the requested homomorphic computations directly on the ciphertexts, and update the blockchain state accordingly.

[6] For multi-user input privacy, a multi-key FHE scheme is used to generate the account key pair.

A smart contract will have functions that users can invoke to operate on their inputs. When these inputs are private, operations within a function will be translated in terms of the following homomorphic computations:[7]

- $\mathsf{Priv.HomAdd}(\mathsf{pk}_{\mathsf{priv}}, \mathsf{c}_1, \mathsf{c}_2)$: Adds ciphertexts c_1 and c_2 (which are encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) together to produce the sum c_3 of the two ciphertexts.
- $\mathsf{Priv.HomMult}(\mathsf{pk}_{\mathsf{priv}}, \mathsf{c}_1, \mathsf{c}_2)$: Multiplies two ciphertexts c_1 and c_2 (which are encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) together to obtain the product c_3.

3.2.3 Handling Concurrency Operating on private states (such as encrypted account balances) introduces concurrency issues. In particular, changes in an account state can invalidate all pending ZKPs tied to this account, thus invalidating all private transactions that rely on these ZKPs. Such a situation can be exploited to perform front-running attacks; Bob can front-run Alice by issuing a transfer transaction that changes Alice's account state and, if this transfer is processed before Alice's pending transactions, her transactions will be rejected. We introduce two complementary techniques to address front-running: automatic balance rollovers for private transactions and a private account locking mechanism for private smart contracts.

Automatic Rollovers. Using this technique, which is similar to the one in [14], all incoming transfers to a private account are held in a pending state until an epoch is complete. smartFHE will roll over these pending funds to private account's balance automatically at the end of the epoch (unlike [14] which requires users to trigger the rollover). To guarantee that deshielding and private transfer transactions will be processed by the end of the same epoch, private account users are advised to submit such transactions at the beginning of an epoch. The length of an epoch must be chosen carefully to ensure that a transaction submitted at the start of an epoch is processed before the epoch ends. The sender should view the transaction amount as being deducted from his own account and reflected in his account balance immediately (to avoid double spending).

Private Account Locking. To address multi-epoch concurrency, smartFHE enables private accounts to be *locked* to other accounts (via $\mathsf{tx}_{\mathsf{lock}}$) of *any* type. The locking mechanism allows a user to put her account on hold for as long as needed—preventing any state changes to her private balance while her own private transactions are still pending. The lockee will issue a $\mathsf{tx}_{\mathsf{unlock}}$ transaction to resume acceptance of new state updates, thereby returning complete control of the locked account to the locker.

- $\mathsf{tx}_{\mathsf{lock}} \leftarrow \mathsf{Lock}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{addr}^{\mathsf{to}})$: First, checks that $\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}$ is not already locked by calling $\mathsf{CheckLock}$. If not, it locks the private account corresponding to $\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}$ to the account corresponding to $\mathsf{addr}^{\mathsf{to}}$ (the latter can even be the same account itself). Finally, it outputs $\mathsf{tx}_{\mathsf{lock}} = (\mathsf{addr}^{\mathsf{to}}, \sigma_{\mathsf{lock}})$ where

[7] Please note that in the multi-key instantiation these algorithms can take in ciphertexts encrypted with respect to different keys. To decrypt such ciphertexts, the relevant users will need to participate in a joint decryption process.

$\sigma_{\mathsf{lock}} = \mathsf{Priv.Sign}(\mathsf{sigsk}_{\mathsf{priv}}, (\mathsf{addr}^{\mathsf{to}}, \mathsf{ctr}[\mathsf{pk}_{\mathsf{priv}}]))$. Note that funds sent to $\mathsf{pk}_{\mathsf{priv}}^{\mathsf{from}}$ will not be rolled over onto the account balance until it is unlocked.

- $\mathsf{tx}_{\mathsf{unlock}} \leftarrow \mathsf{Unlock}(\mathsf{pk}_{\mathsf{priv}})$: First, checks $\mathsf{pk}_{\mathsf{priv}}$ is locked by calling $\mathsf{CheckLock}$. If so, it unlocks the private account corresponding to $\mathsf{pk}_{\mathsf{priv}}$ if and only if the address addr that called Unlock is the same one returned by $\mathsf{CheckLock}(\mathsf{pk}_{\mathsf{priv}})$.
- $\mathsf{CheckLock}(\mathsf{pk}_{\mathsf{priv}})$: Checks if the account corresponding to $\mathsf{pk}_{\mathsf{priv}}$ is currently locked. If $\mathsf{pk}_{\mathsf{priv}}$ is locked, returns the address of the account it is locked to. Otherwise, returns $\perp$.

As an optimization of the smartFHE design, epochs can be eliminated entirely using the above locking mechanism.[8] When issuing a deshield or private transfer transaction, Alice will lock her account to itself. However, the network protocol would need to be modified—so that once the ZKP is verified and the transaction is processed, Alice's account will automatically be unlocked.[9]

3.3 Instantiations

As mentioned before, all known FHE schemes are lattice-based. Hence, a naive instanitation of our framework may also require the use of a lattice-based ZKP system. Indeed this provides full post-quantum security for private accounts but compromises efficiency. To remedy this, we use elliptic curve-based ZKP systems when proving properties about FHE ciphertexts and the underlying plaintexts. In particular, for (single-key) FHE we use the BGV scheme [13], for proving well-formedness of FHE ciphertexts we use short discrete log proofs [42], and for proving properties of a private account balance/private inputs we use Bulletproofs [15]. For digital signatures, we use the (lattice-based) Falcon scheme [28] when issuing transactions from private accounts and ECDSA [33] when the issuers are public accounts (as in Ethereum).

To give a taste of our concrete instantiation (full details can be found in Appendix B), we briefly show how to set up the matrix-vector equation needed for $\mathsf{tx}_{\mathsf{privtransf}}$ (as it is the most complex out of the new transaction types). The sender will need to provide the transfer amount encrypted under both her and the receiver's public keys with the same randomness.[10] Let the sender's public key be represented by matrix $\mathbf{A}$; the receiver's public key, by matrix $\mathbf{B}$. Let $\vec{\mathbf{m}}$ contain the transfer amount amnt and randomness. Then we can form the following matrix-vector equation over $R = \mathbb{Z}_q(x)/(f(x))$:

$$\begin{pmatrix} \mathbf{A} \\ \mathbf{B} \end{pmatrix} \cdot \vec{\mathbf{m}} = \begin{pmatrix} \vec{\mathbf{c}} \\ \vec{\mathbf{c}}' \end{pmatrix}$$

[8] By this we mean setting the epoch length to be equal to one block.

[9] Note that we would still keep the $\mathsf{Lock}, \mathsf{Unlock}$ procedures to handle front-running issues in private smart contracts (to transfer ownership of the user account and keep away incoming transactions for an unknown amount of time).

[10] The scheme is still secure with randomness re-use here (to encrypt the transfer amount under the sender and receiver's keys) via the generalized Leftover Hash Lemma [23].

This equation verifies that $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}'}$ do in fact encrypt the same amount amnt with respect to the sender's public key $\mathbf{A}$ and the receiver's public key $\mathbf{B}$. Thus, we will need to show that $\vec{\mathbf{m}}$ satisfies the above equation and that amnt represented in it is non-negative. This can be done using short discrete log proofs. We will also have another proof that the sender's remaining private account balance is non-negative, which is done using Bulletproofs.

For private smart contracts, a user will need to provide her encrypted inputs to the miner and a ZKP showing these encrypted inputs are well-formed (via short log proofs). We have the following equation over R:

$$\mathbf{A}\vec{\mathbf{s}} = \vec{\mathbf{t}}$$

where the user must prove that her plaintext input and randomness (contained in $\vec{\mathbf{s}}$) encrypt to $\vec{\mathbf{t}}$ with respect to her public key $\mathbf{A}$. As we now have a Pedersen commmitment to the coefficients of $\vec{\mathbf{s}}$, Bulletproofs can be used to prove arbitrary statements about the plaintext (as needed by the application).

Our single-key FHE-based instantiation realizes a correct and secure PPSC scheme based on the notions introduced in Section 2. Formal definitions and proofs can be found in Appendix C.

Extending to multi-user inputs. To support computations over multi-user private inputs, we use the Mukherjee-Wichs multi-key FHE scheme [38] to generate private account keys and subsequently encrypt users' inputs in smartFHE. Since the Mukherjee-Wichs scheme relies on the hardness of LWE, we can re-use the short discrete log proofs construction,[11] and as prior, the Pedersen commitment could then be re-used in Bulletproofs.

The use of the Mukherjee-Wichs scheme introduces additional steps. First, before performing a homomorphic operation on the encrypted inputs coming from n users, miners have to expand each of these ciphertexts to be a ciphertext under all n public keys. This is achieved by having each user send auxiliary information, along with her encrypted input, to allow for this expansion. Second, a user cannot decrypt the output ciphertext on her own; revealing the output requires all n users to engage in a one-round distributed decryption protocol. In particular, each party will use her secret key to partially decrypt the output ciphertext, broadcast the partial decryption to the rest of the users, and can fully decrypt the output once she receives $n-1$ partial ciphertexts from the rest of the users.

Unlike the BGV scheme, the Mukherjee-Wichs scheme requires a trusted setup to generate a common random string that all users must use. To avoid placing trust in a single party to execute the setup process, a multiparty computation (MPC) ceremony can be used where several parties execute a suitable

[11] Unfortunately, the construction takes a large efficiency hit when proving LWE relations (instead of Ring-LWE relations). The proof would require d^2 exponentiations where d is the dimension of the matrix $\mathbf{A}$ so that generating a discrete log proof would now take on the order of minutes instead of seconds [42].

MPC protocol representing the setup. Such ceremonies are widely used to generate public parameters for ZKP systems that require a trusted setup (e.g. the one used in Zcash [2]).

Another issue is fairness of the decryption process. A malicious user may quit after receiving all $n-1$ partial decryptions without broadcasting her partial decryption of the output ciphertext. Only this user will be able to decrypt the FHE computation output. Impossibility of fairness is a classical problem in MPC and, thus, can be resolved using similar techniques to those used with MPC. For example, as in [3,7], parties may create collateral on the blockchain before the FHE computation is performed; this collateral will be revoked if a party quits before finishing the decryption process. Another guaranteed fairness approach (either everyone obtains the output or no one does) can be found in [20] which relies on witness encryption and the blockchain model as well.

Finally, the same optimizations are possible for the Mukherjee-Wichs based smartFHE instantiation as the BGV based instantiation; if we assume circular security, users can use the same keys for all levels. Additionally, bootstrapping can be used to support circuits of arbitrary depth [38].

4 Performance Evaluation

We evaluate the computational and storage cost of the various operations in our instantiation. To this end, we focus on the single-key FHE based instantiation and compare its performance with other schemes in the literature. The rest of this section describes our methodology and discusses the significance of the obtained results.

Methodology. To establish our benchmarks, we examine the cryptographic primitives needed in our construction: FHE, lattice-based digital signatures, discrete log proofs, and Bulletproofs. For each of these primitives, we measure their computational and storage overhead, which we then use to estimate the cost of performing private transactions ($\mathsf{tx}_{\mathsf{shield}}, \mathsf{tx}_{\mathsf{deshield}}, \mathsf{tx}_{\mathsf{privtransf}}$) and private smart contract computations.

We use Microsoft's SEAL library [44] to benchmark the BFV scheme [24], an FHE scheme closely related to BGV,[12] as this library provides good benchmarking support and an optimized implementation. For the Falcon signature scheme, we use its reference implementation [28] with NIST Level I security (i.e. 128 bits). For Bulletproofs, we use the Dalek library [1] with 32-bit range proofs. For elliptic curve operations, we use Curve25519. Finally, for short discrete log proofs, the original work provides no implementation but shows how to estimate proof size and times using cycle count [42]. Thus, we follow the same approach in our evaluation. Our experiments were conducted on a machine with a 2.3 GHz Intel i5 processor and 8 GB RAM.

[12] Specifically, BFV only has relinearization as part of the "refresh" procedure. However, a fully optimized BGV implementation will likely outperform the BFV equivalent [21].

Table 1: FHE performance (BFV) with smallest and largest default parameters.

Performed by	**Operation**	$d = 1024$	$d = 32768$
User	KeyGen	0.364 ms	18.963 s
	Priv.Encrypt	0.471 ms	60.608 ms
	Priv.Decrypt	0.063 ms	31.815 ms
Miner	Priv.HomAdd	0.002 ms	0.888 ms
	Priv.HomMult	0.587 ms	318.881 ms

Table 2: Private transaction costs for smartFHE's instantiation.

Performed by	**Operation**	**Time**	**Size**
User	$\mathsf{Shield}(\mathsf{tx}_{\mathsf{shield}})$	0.671 ms	9.83 KB
	$\mathsf{Deshield}(\mathsf{tx}_{\mathsf{deshield}})$	17.766 s	11.78 KB
	$\mathsf{PrivTransfer}(\mathsf{tx}_{\mathsf{privtransf}})$	46.431 s	22.96 KB
Miner	VerifyShield	0.511 ms	N/A
	VerifyDeshield	3.554 s	N/A
	VerifyPrivTransfer	9.290 s	N/A

Results. We begin with the computational cost of FHE operations, shown in Table 1. We look at the smallest and largest default parameters (polynomial modulus degree $d = 1024$ vs. 32768 which, in our notation, is the degree of $f(x)$ in ring R) supported by SEAL for BFV [44]. As shown, even for very large parameters, FHE operations are quite fast for both users and miners.

We estimate the overhead of the main transactions in our system—shield, deshield, and private transfer (we use the results from Table 1 for $d = 1024$). As shown in Table 2, a shield transaction is lightweight; a client can issue 1490 shield transactions per second. The situation is different for deshield and private transfers, due to the proofs incorporated. With respect to time to generate the various transactions, proof generation for short discrete log proofs dominates the transaction cost when used (i.e. for $\mathsf{tx}_{\mathsf{deshield}}$ and $\mathsf{tx}_{\mathsf{privtransf}}$, log proofs contributes 17.757 s and 46.414 s respectively). While the BFV scheme is fast, the FHE ciphertext dominates the various transaction sizes (i.e. for $\mathsf{tx}_{\mathsf{shield}}$, $\mathsf{tx}_{\mathsf{deshield}}$, and $\mathsf{tx}_{\mathsf{privtransf}}$, the FHE ciphertext contributes 9.16 KB, 9.16 KB, and 18.32 KB respectively). Nevertheless, in comparison to Zkay, our scheme exhibits superior performance in terms of user execution time; a private transfer using Zkay takes a user 70 s even with access to more powerful machine of 4.7 GHz, 6 cores, 32 GB RAM [6].[13]

Next, we consider computations offering I/O privacy implemented within a private smart contract. We focus on two private inputs/outputs, although the following logic applies for an arbitrary number of private inputs (multiply the base cost from with n instead of 2). With a large number of private I/O, the transaction size can be relatively large; to combat degradation in transaction

[13] Zkay's compiled contract for private token transfer is over 1.4 GB in size.

throughput, a sidechain [5,35] can be used to store these FHE ciphertexts instead of having them on the main blockchain. The user encrypts these two values ($2 \cdot 0.471$ ms and $2 \cdot 9.16$ KB), provides up to two proofs each using discrete log proofs ($2 \cdot 15.264$ s and $2 \cdot 1.28$ KB) and Bulletproofs ($2 \cdot 8.005$ ms and $2 \cdot 0.675$ KB), and signs the transaction (0.2 ms and 0.666 KB). This gives an estimated user cost of up to 30.55 s and a transaction size of up to 22.896 KB (with $d = 1024$ for BFV). In contrast, to perform a private computation on two private inputs using Zexe (larger numbers are not provided), the user will take over 52 s even with a more powerful machine of 3.0 GHz, 12 cores, and 252 GB of RAM [12]. Perhaps surprisingly, these results show how our FHE-based PPSC approach can outperform the pure ZKP approach.

References

1. Dalek library. `https://github.com/dalek-cryptography/bulletproofs`
2. Zcash parameter generation. `https://z.cash/technology/paramgen/`
3. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on bitcoin. In: 2014 IEEE Symposium on Security and Privacy. pp. 443–458. IEEE (2014)
4. Attema, T., Lyubashevsky, V., Seiler, G.: Practical product proofs for lattice commitments. IACR Cryptol. ePrint Arch. **2020**, 517 (2020)
5. Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J., Wuille, P.: Enabling blockchain innovations with pegged sidechains. URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains (2014)
6. Baumann, N., Steffen, S., Bichsel, B., Tsankov, P., Vechev, M.: zkay v0. 2: Practical data privacy for smart contracts. arXiv preprint arXiv:2009.01020 (2020)
7. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Annual Cryptology Conference. pp. 421–439. Springer (2014)
8. Biryukov, A., Tikhomirov, S.: Deanonymization and linkability of cryptocurrency transactions based on network analysis. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 172–184. IEEE (2019)
9. Bootle, J., Lyubashevsky, V., Nguyen, N.K., Seiler, G.: A non-pcp approach to succinct quantum-safe zero-knowledge. In: Annual International Cryptology Conference. pp. 441–469. Springer (2020)
10. Bootle, J., Lyubashevsky, V., Seiler, G.: Algebraic techniques for short (er) exact lattice-based zero-knowledge proofs. In: Annual International Cryptology Conference. pp. 176–202. Springer (2019)
11. Bootle, J., Lyubashevsky, V., Seiler, G.: Algebraic techniques for short (er) exact lattice-based zero-knowledge proofs. In: Annual International Cryptology Conference. pp. 176–202. Springer (2019)
12. Bowe, S., Chiesa, A., Green, M., Miers, I., Mishra, P., Wu, H.: Zexe: Enabling decentralized private computation. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 820–837 (2018)
13. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277 (2011), `https://eprint.iacr.org/2011/277`

14. Bünz, B., Agrawal, S., Zamani, M., Boneh, D.: Zether: Towards privacy in a smart contract world. In: International Conference on Financial Cryptography and Data Security. pp. 423–443. Springer (2020)
15. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 315–334. IEEE (2018)
16. Cash, D., Jaeger, J., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., Steiner, M.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: NDSS. vol. 14, pp. 23–26. Citeseer (2014)
17. Chaum, D.: Blind signatures for untraceable payments. In: Advances in cryptology. pp. 199–203. Springer (1983)
18. Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., Juels, A., Miller, A., Song, D.: Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 185–200. IEEE (2019)
19. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: Marlin: Preprocessing zksnarks with universal and updatable srs. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 738–768. Springer, Cham (2020)
20. Choudhuri, A.R., Green, M., Jain, A., Kaptchuk, G., Miers, I.: Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 719–728 (2017)
21. Costache, A., Laine, K., Player, R.: Evaluating the effectiveness of heuristic worst-case noise analysis in fhe. In: European Symposium on Research in Computer Security. pp. 546–565. Springer (2020)
22. Dathathri, R., Saarikivi, O., Chen, H., Laine, K., Lauter, K., Maleki, S., Musuvathi, M., Mytkowicz, T.: Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 142–156 (2019)
23. Dodis, Y., Ostrovsky, R., Reyzin, L., Smith, A.: Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. SIAM journal on computing **38**(1), 97–139 (2008)
24. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. **2012**, 144 (2012)
25. Feige, U., Killian, J., Naor, M.: A minimal model for secure computation. In: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing. pp. 554–563 (1994)
26. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986)
27. Fiore, D., Gennaro, R., Pastro, V.: Efficiently verifiable computation on encrypted data. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 844–855 (2014)
28. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-fourier lattice-based compact signatures over ntru. Submission to the NIST's post-quantum cryptography standardization process (2018)
29. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Annual Cryptology Conference. pp. 75–92. Springer (2013)

30. Halevi, S., Ishai, Y., Jain, A., Komargodski, I., Sahai, A., Yogev, E.: Non-interactive multiparty computation without correlated randomness. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 181–211. Springer (2017)
31. Halevi, S., Lindell, Y., Pinkas, B.: Secure computation on the web: Computing without simultaneous interaction. In: Annual Cryptology Conference. pp. 132–150. Springer (2011)
32. Hallman, R.A., Laine, K., Dai, W., Gama, N., Malozemoff, A.J., Polyakov, Y., Carpov, S.: Building applications with homomorphic encryption. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 2160–2162 (2018)
33. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ecdsa). International journal of information security **1**(1), 36–63 (2001)
34. Kerber, T., Kiayias, A., Kohlweiss, M.: Kachina–foundations of private smart contracts. In: 2021 IEEE 34th Computer Security Foundations Symposium (CSF). pp. 1–16. IEEE (2021)
35. Kiayias, A., Zindros, D.: Proof-of-work sidechains. In: International Conference on Financial Cryptography and Data Security. pp. 21–34. Springer (2019)
36. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE symposium on security and privacy (SP). pp. 839–858. IEEE (2016)
37. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 1–23. Springer (2010)
38. Mukherjee, P., Wichs, D.: Two round multiparty computation via multi-key fhe. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 735–763. Springer (2016)
39. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2019)
40. Noether, S.: Ring signature confidential transactions for monero. IACR Cryptol. ePrint Arch. **2015**, 1098 (2015)
41. Parsons, S., Marcinkiewicz, M., Niu, J., Phelps, S.: Everything you wanted to know about double auctions, but were afraid to (bid or) ask. City University of New York: New York2005 (2006)
42. del Pino, R., Lyubashevsky, V., Seiler, G.: Short discrete log proofs for fhe and ring-lwe ciphertexts. In: IACR International Workshop on Public Key Cryptography. pp. 344–373. Springer (2019)
43. Sasson, E.B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy. pp. 459–474. IEEE (2014)
44. Microsoft SEAL (release 3.5). `https://github.com/Microsoft/SEAL` (Apr 2020), microsoft Research, Redmond, WA.
45. Steffen, S., Bichsel, B., Gersbach, M., Melchior, N., Tsankov, P., Vechev, M.: zkay: Specifying and enforcing data privacy in smart contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1759–1776 (2019)
46. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper (2014)

A Preliminaries

In this section, we review the cryptographic building blocks that will be needed in our schemes—namely, fully homomorphic encryption, zero-knowledge proofs, and digital signatures. We also provide a brief overview Ethereum as our framework builds upon its ideas.

Additional Notation. We use $\mathbb{Z}_p$ to represent $\mathbb{Z}/p\mathbb{Z}$, the arrow notation for column vectors (e.g., $\vec{v}$), and capital letters for matrices. For polynomials, we use boldface notation (e.g., $\mathbf{v}$), boldface with arrow notation for a vector of polynomials (e.g. $\vec{\mathbf{v}}$), and boldface capital letter for a matrix of polynomials. Lastly, $\mathsf{negl}(\lambda)$ is meant to denote negligible functions (recall λ represents the security parameter).

A.1 Overview of Ethereum

Ethereum [46] is a smart contract-enabled cryptocurrency that allows users to perform simple currency transfers in its native currency, Ether, as well as deploy complex applications via the creation of user-defined smart contracts. To this end, Ethereum introduces a Turing-complete language and maintains a virtual machine to execute contracts written in this language. Ethereum relies on an account-based model rather than the UTXO model like Bitcoin [39]. Thus, it introduces a more advanced notion of ledger state, which includes the state of all accounts in the system.

Ethereum provides two types of accounts: externally owned accounts (EOAs) that are controlled by users and contract accounts that are controlled by their contract code. The state of an EOA mainly consists of a nonce (to prevent replay attacks) and a balance, whereas that of a contract account also includes contract code and its storage. Both account types can invoke functions from a smart contract's code. However, only an EOA can initiate a transaction or deploy a smart contract.

Miners will execute the code in any smart contract upon request (i.e. when invoked). To prevent DoS attacks, each operation in Ethereum has some associated cost in terms of gas. Additionally, Ethereum's blockchain has a gas limit which constrains the number of operations that can be executed in a single block.

A.2 Fully Homomorphic Encryption

FHE supports computations directly on ciphertexts. All currently known schemes rely on lattice-based cryptography, thus providing post-quantum security guarantees. We use the BGV single-key FHE scheme [13] and the Mukherjee-Wichs multi-key FHE scheme [38] in our instantiations.

Ring-LWE Encryption Scheme. The basic Ring-LWE public key encryption scheme [37] forms the basis of the BGV (fully homomorphic encryption) scheme

[13]. Thus, we present the Ring-LWE encryption scheme first. Let λ be the security parameter. All operations will be performed over the polynomial ring $R_q = \mathbb{Z}_q[x]/(f(x))$ where q is an integer and $f(x) \in \mathbb{Z}[X]$ is a monic, irreducible polynomial of degree d. The original BGV paper chooses the plaintext space to be 2 (such that $p = 2$ in the syntax below). We loosely follow the presentation from short discrete log proofs here [42].

$\mathsf{E.Setup}(1^\lambda, 1^\mu)$: The setup algorithm takes as inputs security parameter λ and positive integer μ. $\mathsf{E.Setup}$ outputs public parameters $\mathsf{e.pp} = (p, q, d, \chi)$ where p is the size of the plaintext space (often chosen to be binary), q is a μ-bit modulus, $d = d(\lambda, \mu)$ is a power of 2 for $R = \mathbb{Z}[x]/f(x)$ where $f(x) = x^d + 1$, and $\chi = \chi(\lambda, \mu)$ is a "small" noise distribution. The parameters are chosen such that the scheme is based on a Ring-LWE instance that achieves 2^λ security against known attacks [13].

$\mathsf{E.SecretKeyGen}(\mathsf{e.pp})$: The secret key generation algorithm $\mathsf{E.SecretKeyGen}$ outputs secret key $\mathsf{e.sk} = \mathbf{s}$ where $\mathbf{s}$ is a polynomial with small, bounded coefficients from the error distribution χ.

$\mathsf{E.PublicKeyGen}(\mathsf{e.pp}, \mathsf{e.sk})$: The public key generation algorithm outputs public key $\mathsf{e.pk} = (\mathbf{a}, \mathbf{t})$ for $\mathbf{a}, \mathbf{t} \in R_q$ where $\mathbf{a}$ is a random polynomial and $\mathbf{t} = \mathbf{as} + \mathbf{e}$ where $\mathbf{e}$ is a polynomial with small, bounded coefficients from the error distribution χ.

$\mathsf{E.Enc}(\mathsf{e.pp}, \mathsf{e.pk}, \mathsf{m})$: To encrypt message $\mathsf{m} = \mathbf{m} \in R_q$, where all the coefficients of $\mathbf{m}$ are in $\mathbb{Z}_p$, we do the following:

1. Sample polynomials $\mathbf{r}, \mathbf{e_1}, \mathbf{e_2}$ with small, bounded coefficients from the error distribution. Let $\vec{\mathbf{m}}^* = \begin{pmatrix} \mathbf{r} \\ \mathbf{e_1} \\ \mathbf{e_2} \\ \mathbf{m} \end{pmatrix}$ consisting of the message and randomness.
2. Form the matrix $\mathbf{A}$ from $\mathsf{e.pk}$ by setting $\mathbf{A} = \begin{pmatrix} p\mathbf{a} & p & 0 & 0 \\ p\mathbf{t} & 0 & p & 1 \end{pmatrix}$.
3. Compute $\mathbf{A} \cdot \vec{\mathbf{m}}^* = \begin{pmatrix} p\mathbf{a} & p & 0 & 0 \\ p\mathbf{t} & 0 & p & 1 \end{pmatrix} \begin{pmatrix} \mathbf{r} \\ \mathbf{e_1} \\ \mathbf{e_2} \\ \mathbf{m} \end{pmatrix} = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix}$.
4. Output ciphertext $\vec{\mathbf{c}} = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix}$.

$\mathsf{E.Dec}(\mathsf{e.pp}, \mathsf{e.sk}, \vec{\mathbf{c}})$: To decrypt ciphertext $\vec{\mathbf{c}} = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix}$, compute $\mathbf{v} - \mathbf{us} \mod p$. This will return the plaintext message $\mathbf{m}$ since $\mathbf{v} - \mathbf{us} = p(\mathbf{er} + \mathbf{e_2} - \mathbf{se_1}) + \mathbf{m}$ and all the coefficients in the above equation were chosen to be small so that no reduction modulo q occurred.

Correctness is straightforward. Semantic security of the encryption scheme is based on the hardness of Ring-LWE for ring R [37]. Recall that the Ring-LWE problem with appropriately chosen parameters can be reduced (via a quantum reduction) to the Shortest Vector Problem over ideal lattices. For details on the reduction, see [37].

BGV Scheme. The BGV scheme [13] is a (single-key) leveled FHE scheme, meaning that only a certain number of homomorphic multiplications can be performed sequentially before reaching a point at which the resulting ciphertext cannot be decrypted. Each time we perform homomorphic operations (especially multiplication), the ciphertext's noise grows. To help manage the noise growth, a refreshing procedure is introduced that can be performed by anyone. Bootstrapping can be also used as an optimization to avoid having to specify the number of levels (i.e. multiplicative depth) in advance. The security of the BGV scheme follows from the security of the basic Ring-LWE encryption scheme [37].

To prevent the noise from growing so large such that decryption fails, a technique called *modulus switching* is used to keep the noise level roughly constant [13]. When we multiply two ciphertexts $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}}'$ together, we get a long resulting ciphertext that is decryptable under a long secret key. Having to work with these long keys and ciphertexts impacts the efficiency of the scheme so BGV utilizes an additional technique called *key switching* that instead allows us to work with a smaller ciphertext and secret key in place of the originals. Both of these techniques—modulus switching and key switching—are encapsulated in the refreshing procedure that can be performed by anyone.

We present a simplified description of the BGV scheme below. For full details, please see [13].

1. $\mathsf{BGV.Setup}(1^\lambda, 1^L)$: The setup algorithm $\mathsf{BGV.Setup}$ takes as inputs the security parameter λ and the number of levels L. It outputs the parameters $\mathsf{bgv.pp_j}$ for each level $j \in \{L, ..., 0\}$—which includes a modulus, noise distribution, and an integer. We also obtain a ladder of decreasing moduli that will be used in the modulus switching procedure in the algorithm $\mathsf{BGV.Refresh}$.
2. $\mathsf{BGV.KeyGen}(\{\mathsf{bgv.pp_j}\})$: The key generation algorithm $\mathsf{BGV.KeyGen}$ takes as inputs the parameters $\{\mathsf{bgv.pp_j}\}$. It outputs a secret key sk which consists of the secret key $\mathbf{s}_j$ for each level j from L down to 0 (obtained by running $\mathsf{E.SecretKeyGen}(\mathsf{e.pp_j})$), a public key pk which consists of public keys $\mathsf{pk_j}$ for each level j (obtained by running $\mathsf{E.PublicKeyGen}(\mathsf{e.pp_j}, \mathbf{s_j})$), and auxiliary information $\{\tau\}$ needed to facilitate the key switching procedure in $\mathsf{BGV.Refresh}$.
3. $\mathsf{BGV.Encrypt}(\mathsf{bgv.pp}, \mathsf{pk}, \mathsf{m})$: The encryption algorithm $\mathsf{BGV.Encrypt}$ takes as inputs the scheme's parameters $\mathsf{bgv.pp}$, the public key pk, and a message m. It runs $\mathsf{E.Enc}(\mathsf{e.pk_L}, \mathsf{m})$ (which is the same as $\mathsf{E.Enc}(\mathbf{A_L}, \mathbf{m})$) and outputs a ciphertext $\vec{\mathbf{c}}$.
4. $\mathsf{BGV.Decrypt}(\mathsf{bgv.pp}, \mathsf{sk}, \vec{\mathbf{c}})$: The decryption algorithm $\mathsf{BGV.Decrypt}$ takes as inputs the scheme's parameters $\mathsf{bgv.pp}$, the appropriate secret key sk for the level, and a ciphertext $\vec{\mathbf{c}}$. It outputs the corresponding plaintext m by running

$\mathsf{E.Decrypt}(\mathbf{s}_j, \vec{\mathbf{c}})$ (assuming the ciphertext was encrypted with respect to level j).

5. $\mathsf{BGV.HomAdd}(\mathsf{pk}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: BGV.HomAdd is used to add two ciphertexts together. It takes as inputs two ciphertexts—$\vec{\mathbf{c}}_1 = (c_{1,0}, c_{1,1}), \vec{\mathbf{c}}_2 = (c_{2,0}, c_{2,1})$—and the public key pk under which they are encrypted. If the ciphertexts are not encrypted with respect to the same level, then run the BGV.Refresh procedure first. We set $\vec{\mathbf{c}}_3 = (c_{1,0} + c_{2,0}, c_{1,1} + c_{2,1})$—the sum of the two ciphertexts $\vec{\mathbf{c}}_1$ and $\vec{\mathbf{c}}_2$ from performing component-wise vector addition. If desired, we can call BGV.Refresh on $\vec{\mathbf{c}}_3$ and output the "refreshed" result [13]. Otherwise, output $\vec{\mathbf{c}}_3$.
6. $\mathsf{BGV.HomMult}(\mathsf{pk}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: BGV.HomMult is used to multiply two ciphertexts together. It takes as inputs two ciphertexts—$\vec{\mathbf{c}}_1 = (c_{1,0}, c_{1,1}), \vec{\mathbf{c}}_2 = (c_{2,0}, c_{2,1})$—and the public key pk under which they are encrypted. If the ciphertexts are not encrypted with respect to the same level, then run the BGV.Refresh procedure first. We obtain $\vec{\mathbf{c}}_3 = (c_{1,0} \cdot c_{2,0}, c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}, c_{1,1} \cdot c_{2,1})$, the "product" of the two ciphertexts. Finally, we call BGV.Refresh on $\vec{\mathbf{c}}_3$ and output this result.
7. $\mathsf{BGV.Refresh}(\vec{\mathbf{c}}, \tau, q_j, q_{j-1})$: BGV.Refresh takes as inputs a ciphertext $\vec{\mathbf{c}}$, auxiliary information τ to facilitate key switching from secret key $\mathbf{s}_j$ to $\mathbf{s}_{j-1}$, the current modulus q_j, and the next modulus q_{j-1}. It then does the following:
   (a) "Expands": Expand the ciphertext into a powers-of-2 representation.
   (b) "Switch Moduli": Scales the ciphertext to prepare it for modulus switching according to the new modulus q_{j-1}.
   (c) "Switch Keys": Performs the key switching procedure resulting in a new ciphertext $\vec{\mathbf{c}}'$ decryptable under key $\mathbf{s}_{j-1}$ for modulus q_{j-1}.

   BGV.Refresh finally outputs ciphertext $\vec{\mathbf{c}}'$.

Mukherjee-Wichs Multi-key FHE Scheme. The multi-key FHE scheme proposed in [38] extends the single-key GSW scheme [29] to support operating on ciphertexts encrypted under different keys. The GSW scheme represents homomorphic addition/multiplication as matrix addition/multiplication respectively [29]. More importantly, it supports a masking scheme that can be used to extend this single-key scheme to a multi-key one. Unlike the BGV scheme, the Mukherjee-Wichs construction models computations as boolean circuits and requires a trusted setup. However, this scheme relies only on the hardness of LWE (Learning With Errors) and has a one round decryption process. Although the Mukherjee-Wichs multi-key FHE scheme is leveled, bootstrapping can be used to avoid having to specify multiplicative depth in advance.

A.3 Zero-Knowledge Proofs

As FHE uses lattice-based cryptography, lattice-based ZKPs would be a natural candidate for proving relations about our plaintexts in our instantiation. There have been recent improvements to lattice-based ZKPs (namely [11], [9], and [4]) but these constructions still do not achieve the desired efficiency level with regards to proof sizes (<100KB).

Perhaps surprisingly, it is possible to use elliptic curve-based ZKPs to prove relations in lattice-based cryptography quite efficiently via the short discrete log proofs construction [42]. We take this approach in our instantiation to obtain small proof sizes (in the single digit kilobyte range). We will then use Bulletproofs [15] to prove properties of the plaintext (such as ensuring that a currency amount is in a particular value range). Both of these ZKP systems provide soundness, completeness, and zero-knowledge guarantees and can be made non-interactive using the Fiat-Shamir transform [26]. Additionally, neither requires a trusted setup.

Bulletproofs. This proof system [15] allows us to efficiently prove that a committed value is in a particular range using an inner product argument. We have chosen Bulletproofs for our smartFHE instantiation as they are universal (i.e. a single reference string can be used to prove any NP statement), transparent (i.e. no trusted setup), and efficient. Bulletproofs are readily compatible with short discrete log proofs [42], relying also on the hardness of the discrete log assumption. Additionally, the Pedersen commitment obtained from short discrete log proofs can be re-used for our range proof [42].

Short Discrete Log Proofs. This proof system [42] allows us to efficiently prove knowledge of a short vector $\vec{\mathbf{s}}$ such that $\mathbf{A}\vec{\mathbf{s}} = \vec{\mathbf{t}}$ for public $\mathbf{A}$ and $\vec{\mathbf{t}}$ over the polynomial ring $R_q = \mathbb{Z}_q[X]/(f(x))$, where $f(x)$ is a monic, irreducible polynomial of degree d in $\mathbb{Z}[X]$.

To do so, we first form a Pedersen commitment to the coefficients of $\vec{\mathbf{s}}$. This commitment is in some group $\mathbb{G}$ of size p such that the discrete log problem is hard. The proofs owe their efficiency to the fact that p is usually much larger than q, particularly in the FHE setting.

Then, to prove the linear relation, a variant of Bulletproofs is used, which differs from the original Bulletproofs construction in that the inner-product proof will be zero-knowledge [42]. Using the initial Pedersen commitment to $\vec{\mathbf{s}}$, we can use Bulletproofs to prove properties of the plaintext—such as a secret value being in a particular range. The soundness of the proofs is based on the discrete log problem, whereas secrecy is based on Ring-LWE, a problem generally considered to be hard even for quantum computers [37].

A.4 Lattice-based Signature Schemes

We require a secure signature scheme to sign transactions that originate from private accounts. In practice, we would like for such a scheme to be fairly efficient and compatible with our lattice-based FHE scheme. We use the lattice-based Falcon signature scheme [28] in our instantiation, one of three finalists for NIST's post-quantum cryptography standardization competition.

$\mathsf{System.Setup}(1^\lambda, 1^L)$: Takes as inputs the security parameter λ and number of levels L to be supported in the leveled BGV scheme. Outputs the public parameters pp for the entire system including:

- $\mathsf{pp.BGV} \leftarrow \mathsf{BGV.Setup}(1^\lambda, 1^L)$
- $\mathsf{pp.NIZK}_{\mathsf{logproofs}} \leftarrow \mathsf{NIZK}_{\mathsf{logproofs}}.\mathsf{Setup}(1^\lambda)$
- $\mathsf{pp.NIZK}_{\mathsf{bulletproofs}} \leftarrow \mathsf{NIZK}_{\mathsf{bulletproofs}}.\mathsf{Setup}(1^\lambda)$
- $\mathsf{pp.sig}_{\mathsf{priv}} \leftarrow \mathsf{PrivSig.Setup}(1^\lambda)$, setup for signature scheme used for private accounts.
- $\mathsf{pp.key}_{\mathsf{pub}} \leftarrow \mathsf{PubKey.Setup}(1^\lambda)$, setup for signature scheme used for public accounts.

Initializes:

- acc, account table.
- $\mathsf{pendOps}$, pending operations table to keep track of pending transactions and computations.
- $\mathsf{lastRollOver}$, table detailing the last epoch at which a private account's balance was rolled over.
- lock, lock table keeping track of which address a private account is locked to.
- $\mathsf{counter}$, counter table keeping track of the counters associated with accounts.

Also outputs:

- MAX, maximum currency amount smartFHE can support. (We require $\mathsf{MAX} << q$, where q is the modulus of the ring R_q, to prevent possible overflow for balance/transfer amounts.)
- E, epoch length.

Fig. 1: System setup.

B Our Instantiation

This section presents the full syntax and technical details of the single-key FHE-based instantiation of a PPSC scheme. This section is an extended version of Section 3, where for completeness and ease of reference, we provide the full details including the parts that already appeared in that section. As mentioned prior, in our instantiation we use the following cryptographic constructions: ECDSA [33], BGV scheme [13], Falcon signature scheme [28], Bulletproofs [15], and short discrete log proofs [42].

B.1 Syntax

We now outline the syntax used in our implementation. Note that all algorithms take as additional inputs the public parameters pp and the state of the system st_h for the current block height h (but we sometimes omit listing it explicitly). Details on the syntax of the BGV scheme are provided in Appendix A.

System Related. To launch the system, we first perform the setup for the entire system by choosing the public parameters of all cryptographic building blocks as well as the initial state of the ledger. Details can be found in Figure 1.

Public Account Related. A public account owner must maintain a key pair $(\mathsf{pk}_{\mathsf{pub}}, \mathsf{sk}_{\mathsf{pub}})$ to sign outgoing $\mathsf{tx}_{\mathsf{transf}}$ and $\mathsf{tx}_{\mathsf{shield}}$ transactions (we use ECDSA here [33] as in Ethereum), an unencrypted balance $\mathsf{balance}$, and a nonce $\mathsf{ctr}[\mathsf{pk}_{\mathsf{pub}}]$.

1. $\mathsf{Pub.CreateAccount}(\mathsf{pp})$: This algorithm creates a public account and outputs its key pair $(\mathsf{pk}_{\mathsf{pub}}, \mathsf{sk}_{\mathsf{pub}})$.
2. $\mathsf{Pub.ReadBalance}(\mathsf{pk}_{\mathsf{pub}})$: Returns the (plaintext) balance $\mathsf{balance}$ belonging to the public account $\mathsf{pk}_{\mathsf{pub}}$. If no such account exists, returns $\perp$.
3. $\mathsf{Pub.Sign}(\mathsf{sk}_{\mathsf{pub}}, \mathsf{m})$: Produces a signature σ_{pub} on message m with secret key $\mathsf{sk}_{\mathsf{pub}}$.
4. $\mathsf{Pub.VerifySig}(\mathsf{m}, \sigma_{\mathsf{pub}}, \mathsf{pk}_{\mathsf{pub}})$: Verifies a signature σ_{pub} on message m using $\mathsf{pk}_{\mathsf{pub}}$.

Private Account Related. A private account owner maintains key pair $(\mathsf{pk}_{\mathsf{priv}}, \mathsf{sk}_{\mathsf{priv}})$, an encrypted balance (with respect to $\mathsf{pk}_{\mathsf{priv}}$), and a nonce $\mathsf{ctr}[\mathsf{pk}_{\mathsf{priv}}]$. Here, the Falcon signature scheme is used to sign outgoing $\mathsf{tx}_{\mathsf{deshield}}$ and $\mathsf{tx}_{\mathsf{privtransf}}$ transactions.

1. $\mathsf{Priv.CreateAccount}(\mathsf{pp})$: This algorithm creates a private account. It outputs the account key pair $(\mathsf{pk}_{\mathsf{priv}}, \mathsf{sk}_{\mathsf{priv}})$, which are the keys for the BGV scheme, along with the keys for the Falcon signature scheme $(\mathsf{sigpk}_{\mathsf{priv}}, \mathsf{sigsk}_{\mathsf{priv}})$. The public key $\mathsf{pk}_{\mathsf{priv}}$ consists of matrix $\mathbf{A}_j$ for each level j, along with auxiliary information $\tau_{s''_{j+1} \to s_j}$ for key switching; the secret key is the set of secret keys for all levels (i.e., $\mathsf{sk}_{\mathsf{priv}} = \{\mathbf{s}_j\}$). If we assume circular security, then the same public and secret key is used for all levels [13].
2. $\mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}_{\mathsf{priv}}, \mathsf{m})$: Calls $\mathsf{BGV.Encrypt}$ on message m, and outputs ciphertext $\vec{\mathbf{c}}$ encrypted with respect to level L.
3. $\mathsf{Priv.Decrypt}(\mathsf{pp}, \mathsf{sk}_{\mathsf{priv}}, \vec{\mathbf{c}})$: Decrypts a ciphertext $\vec{\mathbf{c}}$ encrypted under $\mathsf{pk}_{\mathsf{priv}}$ for level j by running $\mathsf{BGV.Decrypt}(\mathsf{pp}, \mathbf{s}_j, \vec{\mathbf{c}})$. The level j is auxiliary information associated with the ciphertext $\vec{\mathbf{c}}$.
4. $\mathsf{Priv.ReadBalance}(\mathsf{sk}_{\mathsf{priv}})$: Returns the unencrypted balance $\mathsf{balance}$ belonging to a private account $\mathsf{pk}_{\mathsf{priv}}$. If no such account exists, returns $\perp$.
5. $\mathsf{Priv.Sign}(\mathsf{sigsk}_{\mathsf{priv}}, \mathsf{m})$: Produces a signature σ_{priv} on message m using the Falcon signature scheme.
6. $\mathsf{Priv.VerifySig}(\mathsf{m}, \sigma_{\mathsf{priv}}, \mathsf{sigpk}_{\mathsf{priv}})$: Verifies if the signature σ_{priv} on message m is valid using $\mathsf{sigpk}_{\mathsf{priv}}$.
7. $\mathsf{CheckLock}(\mathsf{pk}_{\mathsf{priv}})$: Checks if the account corresponding to $\mathsf{pk}_{\mathsf{priv}}$ is currently locked. If $\mathsf{pk}_{\mathsf{priv}}$ is locked, returns the address of the account it is locked to. Otherwise, returns $\perp$.

Transaction Related. Users can engage in six types of transactions using their key pairs. We have omitted shield, deshield, and private transfer from here as they are discussed in detail in Section B.2.

1. $\mathsf{Transfer}(\mathsf{sk}_{\mathsf{pub}}^{\mathsf{from}}, \mathsf{pk}_{\mathsf{pub}}^{\mathsf{to}}, \mathsf{amnt})$: Used to send currency from one public account to another public account. It outputs $\mathsf{tx}_{\mathsf{transf}}$.
2. $\mathsf{VerifyTransfer}(\mathsf{tx}_{\mathsf{transf}})$: Verifies if all the conditions for $\mathsf{tx}_{\mathsf{transf}}$ have been satisfied. If yes, it outputs 1. Otherwise, it outputs 0.
3. $\mathsf{Lock}(\mathsf{sk}_{\mathsf{priv}}^{\mathsf{from}}, \mathsf{addr}^{\mathsf{to}})$: First, checks that $\mathsf{sk}_{\mathsf{priv}}^{\mathsf{from}}$ is not already locked by calling $\mathsf{CheckLock}$. If not, it locks private account corresponding to $\mathsf{pk}_{\mathsf{priv}}^{\mathsf{from}}$ to account corresponding to $\mathsf{addr}^{\mathsf{to}}$ (the latter can even be the same account itself). Finally, it outputs $\mathsf{tx}_{\mathsf{lock}} = (\mathsf{addr}^{\mathsf{to}}, \sigma_{\mathsf{lock}})$ where
$$\sigma_{\mathsf{lock}} = \mathsf{Priv.Sign}(\mathsf{sigsk}_{\mathsf{priv}}, (\mathsf{addr}^{\mathsf{to}}, \mathsf{ctr}[\mathsf{pk}_{\mathsf{priv}}]))$$
Note that funds sent to $\mathsf{pk}_{\mathsf{priv}}^{\mathsf{from}}$ will not be rolled over onto the account balance until it is unlocked.
4. $\mathsf{Unlock}(\mathsf{pk}_{\mathsf{priv}})$: First, checks that $\mathsf{pk}_{\mathsf{priv}}$ is locked by calling $\mathsf{CheckLock}$. If so, it unlocks the private account corresponding to $\mathsf{pk}_{\mathsf{priv}}$ if and only if the address addr that called Unlock is the same one returned by $\mathsf{CheckLock}(\mathsf{pk}_{\mathsf{priv}})$. It outputs $\mathsf{tx}_{\mathsf{unlock}}$.

Private Smart Contract Related. Operations on inputs belonging to a private account will be translated into homomorphic computations, with the corresponding smart contract code translated to an arithmetic circuit.

1. $\mathsf{Priv.HomAdd}(\mathsf{pk}_{\mathsf{priv}}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: Runs $\mathsf{BGV.HomAdd}$ on the ciphertexts $\vec{\mathbf{c}}_1$ and $\vec{\mathbf{c}}_2$ (which are encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) to produce the sum of the two ciphertexts. Assuming they are encrypted with respect to the same level j, output $\vec{\mathbf{c}}_3 = \vec{\mathbf{c}}_1 + \vec{\mathbf{c}}_2 \mod q_j$. If not, use $\mathsf{Priv.Refresh}$ first to obtain two ciphertexts at the same level.
2. $\mathsf{Priv.HomMult}(\mathsf{pk}_{\mathsf{priv}}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: Runs $\mathsf{BGV.HomMult}$ on the ciphertexts $\vec{\mathbf{c}}_1 = (c_{1,0}, c_{1,1}), \vec{\mathbf{c}}_2 = (c_{2,0}, c_{2,1})$ (which are encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) to obtain the "product" $\vec{\mathbf{c}}_3 = (c_{1,0} \cdot c_{2,0}, c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}, c_{1,1} \cdot c_{2,1})$. We call $\mathsf{Priv.Refresh}$ on $\vec{\mathbf{c}}_3$ and output the result. If the initial ciphertexts are not encrypted with respect to the same level, we use the $\mathsf{Priv.Refresh}$ procedure first to obtain two ciphertexts at the same level.
3. $\mathsf{Priv.Refresh}(\vec{\mathbf{c}}, \tau, q_j, q_{j-1})$: Runs $\mathsf{BGV.Refresh}$ on the ciphertext $\vec{\mathbf{c}}$ (encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) using auxiliary information τ associated with private account $\mathsf{pk}_{\mathsf{priv}}$ to facilitate key switching and modulus switching from q_j to modulus q_{j-1}.

B.2 Instantiating the Payment Mechanism

We discuss our payment scheme in detail; namely, we show how users perform the shield, deshield and private transfer transactions using our instantiation.

Representing Balances and Transfers. Let $R = \mathbb{Z}_q(x)/(f(x))$. We use the Integer Encoder technique (from SEAL [44]) to represent integer value currency amounts for private accounts as follows:

1. Compute the binary expansion of the integer.
2. Use the bits as coefficients to create the polynomial $g(x)$. Negative integers can be represented via the use of 0 and -1 as coefficients.
3. To get back the integer from a polynomial, simply evaluate the polynomial $g(x)$ at $x = 2$.

Thus, the modulus q must be chosen to be large enough so that there is no overflow. Finally, the newly obtained polynomial (that represents some integer amount) is passed into $\mathsf{Priv.Encrypt}$ to obtain an encryption that hides this amount.

Shielding Transaction. A sender with public account $(\mathsf{pk}^{\mathsf{from}}_{\mathsf{pub}}, \mathsf{sk}^{\mathsf{from}}_{\mathsf{pub}})$ and unencrypted balance $\mathsf{balance}^{\mathsf{from}}$ wishes to send some currency amnt to a private account $(\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{sk}^{\mathsf{to}}_{\mathsf{priv}})$ with encrypted balance $\vec{\mathbf{b}}'$. To do so, the sender issues a shielding transaction $\mathsf{tx}_{\mathsf{shield}}$ containing the following information:

- Receiver's public key: $\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}$
- Transfer amount (in plaintext): amnt
- Transfer amount encrypted under the receiver's public key: $\vec{\mathbf{c}}$
- Randomness used for encrypting transfer amount: r

The sender signs the transaction along with his nonce $\mathsf{ctr}[\mathsf{pk}^{\mathsf{from}}_{\mathsf{pub}}]$, producing signature $\sigma^{\mathsf{from}}_{\mathsf{pub}}$. He then broadcasts the transaction $\mathsf{tx}_{\mathsf{shield}}$ to the miners. The miners check that the following conditions are met (perform $\mathsf{VerifyShield}(\mathsf{tx}_{\mathsf{shield}})$) in order to accept this transaction:

- Valid signature from sender
- Receiver's public key exists/is valid
- Ciphertexts are well-formed
- Transfer amount is positive: $\mathsf{amnt} \in [0, \mathsf{MAX}]$
- Encrypted transfer amount matches plaintext amount with published randomness: $\mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{amnt}; \mathsf{r}) \stackrel{?}{=} \vec{\mathbf{c}}$
- Sender's remaining balance is non-negative: $\mathsf{balance}^{\mathsf{from}} - \mathsf{amnt} \in [0, \mathsf{MAX}]$

If all conditions are satisfied, miners update the sender's account balance to $\mathsf{balance}^{\mathsf{from}} - \mathsf{amnt}$ and the receiver's balance to $\vec{\mathbf{b}}' + \vec{\mathbf{c}}$ (i.e. by calling $\mathsf{Priv.HomAdd}(\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \vec{\mathbf{b}}', \vec{\mathbf{c}})$).

Deshielding Transaction. A sender with private account $(\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}})$ and encrypted balance $\vec{\mathbf{b}}$ wishes to send some currency amnt to the receiver who has public account $(\mathsf{pk}^{\mathsf{to}}_{\mathsf{pub}}, \mathsf{sk}^{\mathsf{to}}_{\mathsf{pub}})$ and unencrypted balance $\mathsf{balance}^{\mathsf{to}}$. The sender will issue a deshielding transaction $\mathsf{tx}_{\mathsf{deshield}}$ containing the following information:

- Receiver's public key: $\mathsf{pk}^{\mathsf{to}}_{\mathsf{pub}}$

- Transfer amount (in plaintext): amnt
- Transfer amount encrypted w.r.t. sender's public key: $\vec{\mathbf{c}}$
- Randomness used for encrypting transfer amount: r
- Sender's remaining encrypted balance $\vec{\mathbf{b}}'$ and proof π_{deshield} that sender's remaining balance is non-negative
  (i.e. $\mathsf{Priv.Decrypt}(\mathsf{pp}, \mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \vec{\mathbf{b}}') = \mathsf{balance}^* \in [0, \mathsf{MAX}]$)

The proof follows from a simple, straightforward application of discrete log proofs [42]. The sender signs the transaction along with his nonce $\mathsf{ctr}[\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}]$, producing signature $\sigma^{\mathsf{from}}_{\mathsf{priv}}$. He then broadcasts $\mathsf{tx}_{\mathsf{deshield}}$ to the miners. For the transaction to be valid, and hence $\mathsf{VerifyDeshield}(\mathsf{tx}_{\mathsf{deshield}}) = 1$, miners check that the following conditions are met:

- Sender's account is not currently locked
- Valid signature from sender
- Receiver's public key exists/is valid
- Transfer amount is positive: $\mathsf{amnt} \in [0, \mathsf{MAX}]$
- Encrypted transfer amount matches plaintext amount with published randomness: $\mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{amnt}; \mathsf{r}) \stackrel{?}{=} \vec{\mathbf{c}}$
- Sender's remaining balance is correctly computed: $\vec{\mathbf{b}}' \stackrel{?}{=} \vec{\mathbf{b}} - \vec{\mathbf{c}}$
- Proof π_{deshield} is valid

If the transaction is valid, miners update the sender's encrypted balance to $\vec{\mathbf{b}}' = \vec{\mathbf{b}} - \vec{\mathbf{c}}$ and the receiver's balance to $\mathsf{balance}^{\mathsf{to}} + \mathsf{amnt}$.

Private Transfer Transaction. A sender with private account $(\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}})$ and encrypted balance $\vec{\mathbf{b}}$ wishes to send some amnt of currency to a recipient who is using a private account $(\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{sk}^{\mathsf{to}}_{\mathsf{priv}})$ with encrypted balance $\vec{\mathbf{b}}'$. Thus, this sender will issue a private transaction $\mathsf{tx}_{\mathsf{privtransf}}$ containing the following information:

- Receiver's public key: $\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}$
- Transfer amount encrypted under sender's public key:
  $\vec{\mathbf{c}} = \mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{amnt}; \mathsf{r})$
- Transfer amount encrypted under receiver's public key:
  $\vec{\mathbf{c}}' = \mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{amnt}; \mathsf{r})$
- Proof that $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}}'$ encrypt same transfer amount amnt with same randomness r and that this transfer amount is in $[0, \mathsf{MAX}]$ [14]
- Proof that sender's remaining (encrypted) balance $\vec{\mathbf{b}}^*$ is non-negative (i.e. $\mathsf{Priv.Decrypt}(\mathsf{pp}, \mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \vec{\mathbf{b}}^*) = \mathsf{balance}^* \in [0, \mathsf{MAX}]$)

The sender signs the transaction along with his nonce $\mathsf{ctr}[\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}]$, producing signature $\sigma^{\mathsf{from}}_{\mathsf{priv}}$. He then broadcasts the transaction $\mathsf{tx}_{\mathsf{privtransf}}$ to the miners. In order to accept this transaction, the miners run $\mathsf{VerifyPrivTransfer}(\mathsf{tx}_{\mathsf{privtransf}})$ which checks that the following conditions are satisfied:

[14] The scheme is still secure with randomness re-use here (to encrypt the transfer amount under the sender and receiver's keys) via the generalized Leftover Hash Lemma [23].

– Sender's account is not currently locked
– Valid signature from sender
– Receiver's public key exists/is valid
– Sender's remaining encrypted balance is correctly computed: $\vec{\mathbf{b}}^* \stackrel{?}{=} \vec{\mathbf{b}} - \vec{\mathbf{c}}$
– All proofs are valid

To prove that the two encryptions are to the same positive transfer amount, we set up the following matrix-vector equation. Let the sender's public key be represented by matrix $\mathbf{A}$; the receiver's public key, by matrix $\mathbf{B}$. Let $\vec{\mathbf{m}}$ contain the transfer amount amnt and randomness. Then we can form the equation:

$$\begin{pmatrix} \mathbf{A} \\ \mathbf{B} \end{pmatrix} \cdot \vec{\mathbf{m}} = \begin{pmatrix} \vec{\mathbf{c}} \\ \vec{\mathbf{c}'} \end{pmatrix} \tag{1}$$

This equation verifies that $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}'}$ do in fact encrypt the same amount amnt with respect to the sender's public key $\mathbf{A}$ and the receiver's public key $\mathbf{B}$. Thus, we will need to show that $\vec{\mathbf{m}}$ satisfies the above equation and that the amount amnt represented in it is non-negative. This can be done using discrete log proofs [42]. We will also have another proof that the sender's remaining balance $\mathsf{Priv.Decrypt}(\mathsf{pp}, \mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \vec{\mathbf{b}}^*)$ is non-negative; this proof is identical to the one that will be provided in $\mathsf{tx}_{\mathsf{deshield}}$.

If the transaction is accepted, miners update the sender's encrypted balance to $\vec{\mathbf{b}} - \vec{\mathbf{c}}$ and the receiver's encrypted balance to $\vec{\mathbf{b}'} + \vec{\mathbf{c}'}$.

C Our Definitions and Proofs

In this section, we define notions for correctness and security of a PPSC scheme. We then show how our single key instantiation satisfies these notions.

C.1 Correctness

Intuitively, the correctness of a PPSC scheme requires that if we start with a valid ledger state and apply an arbitrary sequence of operations (transactions or computations), the resulting state is also valid. Recall that a ledger state is composed of account states. Correctness with respect to public state variables is derived from the correctness of the underlying public system. These can be easily verified by inspecting the ledger since public accounts and all operations performed on them are stored in the clear.

On the other hand, private state variables, which are the extensions introduced by a PPSC scheme, store secret values. Although a smart contract's code is public when operating on private inputs, this code is translated into privacy-preserving operations—meaning that the state evolution over time is private. Thus, proving correctness requires validating these private operations. Correctness of a PPSC scheme is derived from the correctness of the cryptographic building blocks used to implement these operations.

For simplicity, since an account balance is also a state variable subject to updates through smart contract code, we only discuss validating account balances after performing a sequence of private operations. This is expressed by requiring that deshielding (i.e. revealing) a private account balance will produce the same amount of currency as if the original account was public (so that the sequence of operations were all public).

Towards this end, we define an incorrectness game INCORR between an honest challenger $\mathcal{C}$ and a ledger sampler $\mathcal{S}$. At a high level, the game starts by having $\mathcal{C}$ perform the setup phase and pass the public parameters pp to $\mathcal{S}$. After that, $\mathcal{S}$ samples a valid initial ledger $\mathcal{L}$, a public account acc_{pub} (representing the reference point) and a private account acc_{priv} such that their initial balances are identical, and an operation transcript Ops that consists of a sequence of instructions covering all basic operations in the system (more details about this shortly). Ops will be applied separately to acc_{pub} (as is) and acc_{priv} (with an equivalent private version of Ops here) starting with $\mathcal{L}$ in each case.

By a private version of Ops, which we refer to as Ops', we mean replacing the operations in Ops with ones that correspond to the same functionality (i.e. produce identical state changes) but instead of dealing only with public inputs/outputs, Ops' can deal with private inputs/outputs. For example, a public transfer transaction between two public accounts could be translated into a private transfer between two private accounts, a shield transaction if the recipient's public account is replaced with a private one, or to a deshield transaction if the sender's public account is replaced with a private account (all with proper lock-/unlock transactions as needed). For $\mathsf{Compute}$, the circuit C will be transformed into a functionality-equivalent version C' that operates on private inputs and produces private outputs.

Applying these two versions of Ops will produce two updated states of the ledger: $\mathcal{L}'_1$ (when working on acc_{pub}) and $\mathcal{L}'_2$ (when working on acc_{priv}). At the end of the game, the balances of both accounts will be revealed (this requires a deshield transaction for acc_{priv}). $\mathcal{S}$ wins the INCORR game if it can produce a scenario in which the balance of acc_{priv} is not equal to the balance of acc_{pub}.

Definition 2 (Correctness of PPSC Scheme). *A PPSC scheme $\Pi = (\mathsf{Setup}, \mathsf{CreateAccount}, \mathsf{CreateTransaction}, \mathsf{VerifyTransaction}, \mathsf{Compute}, \mathsf{UpdateState})$ is correct if no* PPT *ledger sampler $\mathcal{S}$ can win the INCORR game with non-negligible probability. In particular, for every* PPT *$\mathcal{S}$ and sufficiently large security parameter λ, we have*

$$\mathsf{Adv}^{\mathit{INCORR}}_{\Pi,\mathcal{S}} < \mathsf{negl}(\lambda)$$

where $\mathsf{Adv}^{\mathit{INCORR}}_{\Pi,\mathcal{S}} := \Pr[\mathit{INCORR}(\Pi, \mathcal{S}, 1^n) = 1]$ is $\mathcal{S}$'s advantage of winning the incorrectness game.

We now describe the incorrectness experiment—which includes specifications of a valid operation list Ops, the state evolution of a ledger $\mathcal{L}$, and the interaction between $\mathcal{C}$ and $\mathcal{S}$.

Specifications of a Valid Ops**.** Let $\mathsf{Ops} = \{\mathsf{op}_i\}$ be a list of operations sampled by $\mathcal{S}$, where each op_i can be a transaction tx_i or a computation $\mathsf{Compute}(\mathsf{pp}, C_i, \{x_{i,1}, \ldots, x_{i,n}\})$. We say that Ops is valid if it satisfies the following:

- All account addresses, keys, and states are generated using $\mathsf{CreateAccount}$.
- Each op_i is either a transaction defined in a PPSC scheme (cf. Definition 1), a public transaction as defined in the underlying public smart contract-enabled system, or a $\mathsf{Compute}$ operation with some arbitrary circuit C and a set of inputs $\{x_i\}$.
- If an operation op_i is issued in epoch i, then the ledger state used to produce op_i (if needed) is the one produced by the last block of epoch $i-1$.

The last condition implies that an operation issued in an epoch will be processed in the same epoch, which reflects the assumption of processing delays we have in our system.

Ledger State Evolution. A ledger state is composed of two tables, Bal and Lk, that store the balance amount and lock state for each account. These tables are indexed using the public keys of the accounts (i.e. $\mathsf{Bal}[\mathsf{pk}]$ returns the plaintext amount of currency that the account associated with pk owns, and $\mathsf{Lk}[\mathsf{pk}]$ returns the address to which the account pk is locked or $\perp$ if the account is unlocked). Let the initial ledger state sampled by $\mathcal{S}$ be $\mathcal{L}_0$. Bal and Lk will be initially set to 0 and $\perp$, respectively, for all accounts (including those for acc_{pub} and acc_{priv} sampled by $\mathcal{S}$).

Let $\mathcal{L}_i$ be the i^{th} ledger state defined based on $\mathcal{L}_{i-1}$ and the i^{th} operation op_i. The updates result from processing an op_i is defined as follows:

- $\mathsf{tx}_{\mathsf{shield}} \leftarrow \mathsf{Shield}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{pub}}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{val})$. If the sum of val and $\mathsf{Bal}[\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}]$ is less than MAX and $\mathsf{Lk}[\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}] = \perp$, then increment $\mathsf{Bal}[\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}]$ by val.
- $\mathsf{tx}_{\mathsf{privtransf}} \leftarrow \mathsf{PrivTransfer}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{val})$. If $\mathsf{Lk}[\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}] = \mathsf{Lk}[\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}] = \perp$, then increment $\mathsf{Bal}[\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}]$ by val and decrement $\mathsf{Bal}[\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}]$ by val.
- $\mathsf{tx}_{\mathsf{deshield}} \leftarrow \mathsf{Deshield}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{pub}}, \mathsf{val})$. If $\mathsf{Lk}[\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}] = \perp$, then decrement $\mathsf{Bal}[\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}]$ by val and increment $\mathsf{Bal}[\mathsf{pk}^{\mathsf{to}}_{\mathsf{pub}}]$ by val.
- $\mathsf{tx}_{\mathsf{lock}} \leftarrow \mathsf{Lock}(\mathsf{sk}, \mathsf{addr})$. If $\mathsf{Lk}[\mathsf{pk}] = \perp$ then set $\mathsf{Lk}[\mathsf{pk}] = \mathsf{addr}$ (where pk is the public key associated to sk).
- $\mathsf{tx}_{\mathsf{unlock}} \leftarrow \mathsf{Unlock}(\mathsf{pk})$. If $\mathsf{Lk}[\mathsf{pk}] = \mathsf{tx}_{\mathsf{unlock}}.\mathsf{addr}$, then set $\mathsf{Lk}[\mathsf{pk}] = \perp$ (where $\mathsf{tx}_{\mathsf{unlock}}.\mathsf{addr}$ is the account address that issued $\mathsf{tx}_{\mathsf{unlock}}$).
- $\mathsf{Compute}(\mathsf{pp}, C, \{x_1, \ldots, x_n\})$. Updates depend on the code that C represents. These may include altering storage variables related to the smart contract code and/or account balances.

INCORR Game Definition. The probabilistic experiment INCORR takes as inputs a PPSC scheme Π and a security parameter λ. It defines an interaction between a challenger $\mathcal{C}$ and a ledger sampler $\mathcal{S}$. The game terminates with an output from $\mathcal{C}$—which is 1 if $\mathcal{S}$ succeeds in breaking the correctness of Π and 0 otherwise. The game proceeds as follows:

1. $\mathcal{C}$ runs $\mathsf{System.Setup}(1^\lambda)$ and sends the public parameters pp to $\mathcal{S}$.

2. $\mathcal{S}$ sends back a ledger $\mathcal{L}$, two accounts acc_{pub} and acc_{priv}, and an operation transcript Ops.
3. $\mathcal{C}$ verifies the validity of the transcript (based on the specifications listed above), that the two accounts are recorded on the ledger, and that the state is initialized properly. If any of these checks fail, $\mathcal{C}$ aborts and outputs 0.
4. $\mathcal{C}$ applies Ops to acc_{pub} with ledger state $\mathcal{L}$ and produces an updated ledger state $\mathcal{L}'_1$. Then, it applies an equivalent private version of Ops to acc_{priv} with the same initial ledger state $\mathcal{L}$ and produces an updated ledger state $\mathcal{L}'_2$.
5. $\mathcal{C}$ deshields the balance of acc_{priv} on $\mathcal{L}'_2$ and outputs 1 if the revealed balance is different from the balance of acc_{pub} as recorded on $\mathcal{L}'_1$—meaning that $\mathcal{S}$ won the game. Otherwise, it outputs 0.

The advantage of $\mathcal{S}$ in wining the INCORR game is defined as the probability that $\mathcal{C}$ outputs 1.

Correctness of our instantiation. Informally, correctness is derived from the correctness of the cryptographic building blocks. Every operation, whether a valid transaction or a circuit computation, will be processed successfully in our instantiation and leads to a verifiable ledger update. This can easily be seen for each transaction type in the system. By relying on the completeness of the ZKP systems for Bulletproofs [15] and short discrete log proofs [42], the correctness of the BGV fully homomorphic encryption scheme [13], the locking process (to lock account states to avoid invalidating any pending ZKPs), and the rolling over process at the end of each epoch, it can be shown that valid transactions will update the ledger state as expected. The same is true for $\mathsf{Compute}$ requests. For computations on private inputs, the correctness of the results is based on the correctness of the BGV scheme [13].

Accordingly, in the INCORR game, applying Ops to acc_{pub} and applying an equivalent private version Ops' to acc_{priv}, will lead to the same final balance value. Given that all balance values are not allowed to exceed some maximum value MAX determined by the system's setup, the homomorphic operations on account balances will not cause an overflow.

C.2 Security

A PPSC scheme is secure if it satisfies two properties, namely, overdraft safety and ledger indistinguishability, as captured by the following definition.

Definition 3 (Security of a PPSC Scheme). *A PPSC scheme Π = (*Setup*,* $\mathsf{CreateAccount}$*,* $\mathsf{CreateTransaction}$*,* $\mathsf{VerifyTransaction}$*,* $\mathsf{Compute}$*,* $\mathsf{UpdateState}$*) is secure if it satisfies ledger indistinguishability and overdraft safety.*

Our notions for overdraft safety and ledger indistinguishability are similar to those in [43] and [14]. However, we make the appropriate changes to take into account our different account types, transaction types, and algorithms listed in PPSC definition.

We first define the common security game Security-Game that will be used in overdraft safety and ledger indistinguishability. Let $\mathcal{A}$ represent the adversary; $\mathcal{C}$, the challenger (who represents honest users in our system); $\mathcal{O}_{\mathsf{PPSC}}$, the oracle for our PPSC scheme. Both $\mathcal{C}$ and $\mathcal{A}$ have access to this oracle.

All parties receive the security parameter λ as input. $\mathcal{O}_{\mathsf{PPSC}}$ maintains the public parameters pp and the state of the system. We define PK to be the set of public keys generated by $\mathcal{C}$ at $\mathcal{A}$'s request. Since these belong to $\mathcal{C}$, $\mathcal{A}$ does not have the corresponding secret keys for them. $\mathcal{C}$ can request the current state or previous state from $\mathcal{O}_{\mathsf{PPSC}}$ at any time. $\mathcal{O}_{\mathsf{PPSC}}$ will answer queries from adversary $\mathcal{A}$ proxied by $\mathcal{C}$. Any time a query requires a secret key belonging to $\mathcal{C}$ as input, we allow $\mathcal{A}$ to specify the corresponding public key (which is the set PK).

When $\mathcal{O}_{\mathsf{PPSC}}$ receives a well-formed transaction or computation from either $\mathcal{C}$ or $\mathcal{A}$, it will be added to the list of pending transactions and computations denoted as Ops. $\mathcal{A}$ will also be allowed to directly insert his own well-formed transactions and computations via an Insert query and ask these to be processed immediately via UpdateState.

$\mathcal{A}$ is permitted to make the following query types:

- Request $\mathcal{C}$ to perform any of the user algorithms with certain inputs and send the resulting transaction (if any) to $\mathcal{O}_{\mathsf{PPSC}}$ from an EOA address of $\mathcal{A}$'s choice
  - For CreateAccount, $\mathcal{C}$ will send *only* the resulting EOA address and public key to $\mathcal{A}$
  - For Compute, $\mathcal{C}$ will only agree to perform computations supported by the PPSC system
  - $\mathcal{C}$ will refuse to perform a transaction from a locked account
- Insert, allows $\mathcal{A}$ to send his own well-formed transaction or computation to $\mathcal{O}_{\mathsf{PPSC}}$ which will be held in pending state until processed via UpdateState
- UpdateState, allows $\mathcal{A}$ to ask $\mathcal{O}_{\mathsf{PPSC}}$ to process an arbitrary subset of pending operations and update the state (i.e. add a new block to the blockchain)

For UpdateState, note that the usual conditions around when certain transactions to private accounts are processed still apply. As $\mathcal{C}$ represents the honest parties in the system, $\mathcal{C}$ will use the state of the previous epoch when performing transactions that require it. Lastly, $\mathcal{A}$ can stop the game at any point.

Overdraft Safety. Overdraft safety ensures that a PPSC scheme Π does not allow $\mathcal{A}$ to spend more currency than he owns. To capture this, we define an Overdraft-Safety-Game game in which $\mathcal{C}$ and $\mathcal{A}$ interact in the same manner as they do in Security-Game. $\mathcal{A}$ wins the game and, hence, breaks overdraft safety if he manages to spend currency of a value larger than what he rightfully owns. This is expressed formally in the following definition:

Definition 4 (Overdraft Safety). *A PPSC scheme Π provides overdraft safety if for all* PPT *adversaries $\mathcal{A}$, the probability that*

$$\mathsf{val}_{\mathcal{A}\to\mathsf{PK}} + \mathsf{val}_{\mathsf{Insert}} > \mathsf{val}_{\mathsf{PK}\to\mathcal{A}} + \mathsf{val}_{\mathsf{deposit}} \tag{2}$$

in the Overdraft-Safety-Game is $\mathsf{negl}(\lambda)$ *where the probability is taken over the coin tosses of* $\mathcal{A}$ *and* $\mathcal{C}$ *and:*

- $\mathsf{val}_{\mathcal{A}\to\mathsf{PK}}$ *is the total value of payments confirmed from* $\mathcal{A}$ *to users with addresses in* PK
- $\mathsf{val}_{\mathsf{Insert}}$ *is the total value of payments placed by* $\mathcal{A}$ *on the ledger*
- $\mathsf{val}_{\mathsf{PK}\to\mathcal{A}}$ *is the total value of payments confirmed from users with addresses in* PK *to* $\mathcal{A}$
- $\mathsf{val}_{\mathsf{deposit}}$ *is the initial amount of currency in accounts owned by* $\mathcal{A}$*.*

$\mathcal{A}$ has two ways in which he can win the game—by inserting his own transactions into the ledger (handled by $\mathsf{val}_{\mathsf{Insert}}$) or by asking honest parties represented by $\mathcal{C}$ to create transactions for him (handled by $\mathsf{val}_{\mathcal{A}\to\mathsf{PK}}$).

Overdraft safety of our instantiation. We now show how our instantiation provides overdraft safety. We will look at $\mathsf{Transfer}, \mathsf{Shield}, \mathsf{Deshield}, \mathsf{PrivTransfer}$ and show that none of these transaction algorithms can be used to send more currency than a user rightfully owns with non-negligible probability. As in [14], the nonce associated with each account will enforce order on the pending transactions and prevent $\mathcal{A}$ from double-spending. Additionally, the instantiation satisfies correctness (as seen prior) so that private computations cannot be used to falsely increase a user's account balance. Ultimately, operations on private balances and transfer amounts will be captured in transactions.

Proof Sketch. In $\mathsf{Transfer}$, all account and transaction details are associated with public accounts so are publicly verifiable information (e.g. sender/receiver's balances, transfer amount). Thus, if the sender attempts to send more currency than he rightfully owns, $\mathsf{VerifyTransfer}$ would output 0 and the transaction would be rejected.

In Shield, the state of the sender's account can be publicly tracked and verified. The encrypted transfer amount will be checked to ensure that it matches the published plaintext transfer amount with randomness and that the sender's remaining balance is non-negative. If the sender attempts to send more currency than he rightfully owns, $\mathsf{VerifyShield}$ will output 0.

In $\mathsf{Deshield}$, the state of the sender's account is private. The encrypted transfer amount will be checked to ensure it matches the published non-negative plaintext transfer amount with corresponding randomness. The zero-knowledge proof showing that the sender has enough currency in his private account to perform this transfer will also be checked as part of $\mathsf{VerifyDeshield}$. Thus, if the sender is able to send more currency than he rightfully owns (i.e. $\mathsf{VerifyDeshield}(\mathsf{tx}_{\mathsf{deshield}}) = 1$), he has violated the soundness of the ZKP systems of Bulletproofs or short discrete log proofs (which happens with at most negligible probability).

In $\mathsf{PrivTransfer}$, the state of the sender and receiver's accounts are private. As part of $\mathsf{VerifyPrivTransfer}$, ZKPs will be checked showing that the sender has enough currency in his account to perform the transaction and that the transfer amount encrypted under the sender and receiver's public key matches and is non-negative. Thus, if the sender is able to send more currency than he rightfully

owns (i.e. $\mathsf{VerifyPrivTransfer}(\mathsf{tx}_{\mathsf{privtransf}}) = 1$), he has violated the soundness of the ZKP systems of Bulletproofs or short discrete log proofs (which happens with at most negligible probability).

Ledger Indistinguishability. Ledger indistinguishability ensures that the ledger produced by the PPSC scheme Π does not reveal additional information about private data beyond what can be inferred from what is publicly revealed. We define a $\mathsf{Ledger\text{-}Indistinguishability\text{-}Game}$ to capture this. It is the same as $\mathsf{Security\text{-}Game}$ except that at some point in the game, $\mathcal{A}$ will send two publicly consistent instructions instead of one (we define publicly consistent instructions below, which is needed to rule out trivial wins of $\mathcal{A}$). $\mathcal{C}$ will execute one of these instructions based on bit b that is hidden from $\mathcal{A}$ which is chosen at random and in advance. $\mathcal{A}$ will have to guess which instruction $\mathcal{C}$ performed at the end of the game. Let b' be $\mathcal{A}$'s guess.

We first define the notion of public consistency of two instructions.

Definition 5 (Public Consistency). *Two instructions are publicly consistent if:*

- *They refer to the same user algorithm with the same public key/address.*
- *All transactions are associated with the same sender and recipient.*
- *For transactions including a public EOA, the transfer amount must be the same.*
- *For transactions between private EOAs, if the recipient is corrupt then the transfer amount must be the same.*
- *If computations are requested, they must be the same computations on the same inputs.*
- Lock *must be associated with the same account and address for the locker and lockee.*
- Unlock *must be associated with the same account.*
- *Same balance value returned when querying an account's balance.*

Based on the above, we formally define the ledger indistinguishability property.

Definition 6 (Ledger Indistinguishability). *A PPSC scheme Π satisfies ledger indistinguishability if for all* PPT *adversaries $\mathcal{A}$, the probability the $b' = b$ in the* $\mathsf{Ledger\text{-}Indistinguishability\text{-}Game}$ *is* $1/2 + \mathsf{negl}(\lambda)$ *where the probability is taken over the coin tosses of $\mathcal{A}$ and $\mathcal{C}$.*

Ledger indistinguishability of our instantiation. We show that our instantiation satisfies the ledger indistinguishability property.

Proof Sketch. We have defined public consistency to rule out trivial wins by the adversary. This leaves us with the following cases to consider:

- A deshielding transaction.

– A private transfer transaction.

For two consistent deshielding transactions, $\mathcal{A}$ has a negligible advantage of winning the game due to the zero-knowledge property of the proof systems employed.

The same argument holds for two consistent private transactions. $\mathcal{A}$ has a negligible advantage of winning the game due to the zero-knowledge property of the underlying ZKP systems. Additionally, note that the ciphertexts of the transfer amounts are computationally indistinguishable from random assuming the BGV scheme is semantically secure. Thus, with overwhelming probability, they will not reveal any additional information that may help $\mathcal{A}$ in guessing b correctly.

Based on the above, we have the following theorem.

Theorem 1. *Our instantiation as described in Appendix B realizes a correct (cf. Definition 2) and secure (cf. Definition 3) PPSC scheme (cf. Definition 1).*

D Applications

In this section, we demonstrate how a single key FHE-based instantiation of smartFHE can be used to support some multi-user applications (with additional contract code).

D.1 Sealed-bid Auctions on Multiple Items

Bidding on *multiple* items of a good is of interest in many financial and trading services. The stock exchange, for example, allows potential buyers to bid on multiple shares of a stock using auctions [41]. These auctions allow buyers and sellers to specify not only the per-item price, but also the quantity (or number of shares) they are willing to buy or sell. In what follows, we show how a single-key instantiation of smartFHE can be used to implement this multi-item sealed-bid auction while avoiding a serious DoS attack.

A smart contract, representing a simplified stock exchange, can be deployed to allow buyers and sellers to post bids and offers respectively. In its simplest form, each seller can publicly specify the maximum number of shares of a given stock she is willing to sell. Buyers can submit their sealed bids; each of which is composed of a private per-item price and a private quantity value (encrypted with respect to their private accounts). The auction proceeds in two phases: a bidding phase during which bidders post private bids along with proofs of their correctness and a matching phase during which those bidders reveal their bids to allow settlement.[15] To settle the auction, the auctioneer (which will be a smart

[15] We require bidders to lock their private accounts to the smart contract account as part of the bidding process. At the end of the auction, the smart contract will unlock all bidders' accounts.

contract in our case) needs to compute the market clearing price (which could be the highest per-item price among all bids), match buyers with sellers, and enforce currency transfer from the buyer to the seller. The last condition requires multiplying together the ciphertexts of the per-item price and the quantity of items in the matched bid—a homomorphic multiplication operation.

For example, Bob may post an offer to sell *up to* 32 shares of Noether's stock. Alice wants to buy n shares of this stock at price p per share. Alice maintains a private account in smartFHE with key pair $(\mathsf{pk}_{\mathsf{priv}}, \mathsf{sk}_{\mathsf{priv}})$ and encrypted balance b'_{Alice}. To construct a sealed bid, Alice encrypts the values (n, p) under $\mathsf{pk}_{\mathsf{priv}}$ to get (n', p') and submits the output to the exchange smart contract. She also needs to submit ZKPs attesting to the well-formedness of the ciphertexts, that the number of shares she wants to buy is within the range that Bob is offering, and that she has enough currency in her account to make the bid. All of this can be done via the proof systems in smartFHE's single-key instantiation, consisting of short discrete log proofs [42] and Bulletproofs [15].

In the reveal phase, all bids that were not rejected (due to invalid ZKPs) will be given a timeout to be revealed. Alice, the winner of the auction, will then create a private transfer transaction of the total amount—namely, the plaintext value $bid_{total} = np$—and present it to the smart contract. Alice's balance after the private transaction will be updated to $b'_{Alice} - (n'p')$. Bob's balance will be updated to $b'_{Bob} + bid'_{total}$ (which is the sum of Bob's private account balance and the winning bid amount encrypted under Bob's key).

To see why homomorphic multiplication is needed, note that proving that Alice's balance can cover the total bid value requires multiplying the ciphertexts together as $n'p'$. Alice is able to provide ZKPs proving properties of the individual ciphertexts (e.g. n' encrypts a value n such that $0 < n \leq 2^5$ where 2^5 is the total number of shares offered by Bob), as well as a ZKP over the homomorphically multiplied ciphertext that will be computed later. This multiplication capability is also needed to prevent other serious attacks.

To clarify, we consider an alternative bidding approach that does not require homomorphic multiplication. One may suggest computing the total bid value $bid_{total} = np$ locally and then submitting an encryption of the output, along with encryptions of the per-item price and quantity, n' and p', respectively. Next, a ZKP could be computed attesting that the buyer's balance can cover bid_{total}. In the reveal phase, the bidder reveals all values (n, p, and bid_{total}); anyone can verify that np equals to bid_{total}.

However, such an approach exposes the system to a DoS attack. A malicious bidder can provide a valid ZKP proving that they can cover bid_{total}, but with invalid n and p values such that $np \neq bid_{total}$. This will be detected in the reveal phase if the bidder reveals the bid. At this stage, the exchange smart contract will reject such a fraudulent bid but *after* performing all computations needed to verify the attached ZKPs. Thus, an attacker may exploit this vulnerability and submit a large number of fraudulent bids, making the exchange unavailable to honest users. Although other means can be used here, such as punishing a malicious party financially via a penalty deposit, it may potentially be infeasible

to compute a lower bound for this financial punishment (which would require knowing the utility gain of the attackers). Supporting homomorphic multiplication removes the need for additional countermeasures and makes our system secure against all efficient adversaries, rather than only rational and efficient ones.

D.2 Private Inventory Tracking

In certain trading scenarios, buyers and sellers may agree to trade a quantity of items that have yet to be produced. In such a scenario, there is a chance that the seller may not produce the agreed upon amount within the specified timeline so that the buyer will contact several sellers to increase the chance of finalizing the deal on time. Thus, a binding trading contract between the buyer and seller is needed (which automatically settles the trade once the seller produces the items and financially punishes the seller if he does not meet the agreed-upon timeline). In what follows, we show how the single-key FHE based instantiation of smartFHE can be used to implement such a binding trading contract.

In particular, Alice, the buyer, can create a smart contract to track the inventory of m products. For each of these m products, the contract will store a private per-item price, denoted as p'_i, and a private counter tracking the number of items produced so far, denoted as n'_i for $i \in \{1, \ldots, m\}$. The trading process is composed of two stages: deal term negotiation and item production. In the negotiation period, Alice negotiates the per-item price, quantity, and the timeline with Bob, the seller. This stage concludes with Alice registering Bob as the seller for a product in the list, and Bob recording the per-item price and quantity they agreed on. The latter is done by encrypting these two values under Alice's public key and storing them on the smart contract. Furthermore, Bob records the production deadline which is simply the index of some future block on the blockchain.

After finalizing the deal terms, both Alice and Bob have to create penalty deposits by sending currency to the trading smart contract. These deposits will be used to financially punish the parties if they do not execute the trading terms. In contrast to sealed-bid auctions, this is feasible here since the utility gain of both parties can be computed (which could be set as a proper compensation for the losses).

The production stage will start once the penalty deposits are in place and continues until the agreed-upon deadline. At that time, Alice will be given a period to dispute the produced quantity (e.g. by revealing that the agreed upon quantity and the quantity produced by Bob are not equal). If there is a mismatch, Bob's penalty deposit will be given to Alice as compensation. Otherwise, the trading contract will compute a ciphertext of the total payment value as $p'_1 n'_1$, assuming Bob's product is at index 1 in the product array. Alice will then create a private transfer transaction of the total amount—namely, the plaintext $p_1 n_1$—owed to Bob and present it to the smart contract. If no such transaction is issued within a given period, the trading contract will give Alice's penalty

deposit to Bob. Otherwise, the trading contract will refund the parties their deposits and reset the inventory tracking variables to allow them to start another trade (if desired).