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. Smart contract-enabled blockchains represent a powerful tool in supporting a large variety of applications. Despite their salient features of transparency, decentralization, and expressiveness, 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, support private computation on inputs belonging to only one user, or even ask the users themselves to coordinate and perform the computation off-chain.

In this paper, we propose *smartFHE*, a framework to support private smart contracts that utilizes fully homomorphic encryption (FHE). smartFHE allows users to build arbitrary decentralized applications that preserve input/output privacy for an arbitrary number of users' inputs. This is achieved by employing (single and multi-key) FHE to compute over private (encrypted) data and account balances, along with efficient zero-knowledge proof systems (ZKPs) to prove well-formedness of private transactions. 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. Furthermore, smartFHE reduces the burden on the users; users provide their private inputs but miners are responsible for performing the private computation. To the best of our knowledge, we are the first to use FHE in the blockchain model.

We define notions for a privacy-preserving smart contract (PPSC) scheme along with its correctness and security. Then, we provide an instantiation of a PPSC using smartFHE and evaluate its performance. Despite common beliefs, our benchmarks show that FHE operations are fast even on a modest machine. This is reflected in the performance of our instantiation; private payments and smart contract computations are faster on our modest machine than state-of-the-art schemes using much more powerful machines.

1 Introduction

Cryptocurrency can be traced back to (at least) 1983 when Chaum first proposed the concept of *electronic cash* using blind signatures [1]. Extending Chaum's design, Bitcoin [2] was introduced a few decades later and removed the need for a trusted party. It was an appealing concept: a way to exchange currency in a cryptographically secure way without relying on banks. Since then, hundreds

of cryptocurrencies have been deployed with a total market cap exceeding $1.4 trillion [3]. Moreover, the concept of virtual currency has expanded to support a variety of applications.

Cryptocurrency and Privacy. In Bitcoin [2], Nakamoto introduced the notion of a public distributed ledger called blockchain through which users could exchange currency directly with one another. Although users have addresses that serve as their pseudonymous identities in the system, anyone can trace transactions on the blockchain to see exactly how much was exchanged and by which addresses. There have even been successful attempts to link these addresses with real-world identities [4].

This lack of privacy resulted in several initiatives to bring privacy to cryptocurrency such as Zerocash [5] and Monero [6]. Such schemes usually exploit additive homomorphisms in commitment or encryption schemes and use zero-knowledge proofs (ZKP) to prove certain relations hold on the committed or encrypted values. While these constructions succeed in bringing confidentiality (i.e. hiding the transfer amount) to cryptocurrency, they do not support much more than private currency transfer.

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 could support arbitrary user-defined programs called smart contracts [7]. Using these capabilities, individuals can build applications processing highly sensitive data such as auctions and voting. 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 bring privacy to smart contracts. However, supporting arbitrary computation with privacy for even a *single* user's inputs and outputs has proved to be quite the challenge. Some constructions (such as Zether [8]) built upon the approach used for private currency transfer—operating directly on (ElGamal based) encryptions with ZKPs used to prove that certain relations hold. However, additive homomorphisms can support only a limited set of applications with input/output (I/O) privacy. Other constructions (such as Zexe [9] and Zkay [10]) abandoned this approach since it did not yield immediately practical results in the short-term. Instead, they resort to offloading *all* work to the user to do offline. Users perform the intended computations on plaintext data, encrypt the inputs and the computed outputs, and create a ZKP certifying correctness of computation with respect to these encryptions. The blockchain miner's only role here is to verify correctness of the ZKP. We refer to this approach as the "pure ZKP approach" as it relies on the power of ZKPs to perform computations with I/O privacy.

1.1 Our Contributions

Operating directly on encrypted values has proven invaluable across numerous applications—from searching over encrypted databases [11], to protecting privacy in machine learning applications involving neural networks [12]. Prominent private smart contract schemes may have chosen to abandon this approach as it did not yield practical results in the short-term; we believe 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) which provides I/O privacy for any computation. 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 proof and then perform the requested computations directly on the encrypted inputs. No need for the user to remain online during the computation or provide complex ZKPs attesting to correctness of computation. Unlike the pure ZKP approach, here the private computation is performed on-chain.

FHE's primary (if not sole) drawback is its poor efficiency. However, even this viewpoint is slightly outdated. Industry has been championing more efficient implementations of popular FHE schemes each year; for example, a particular variant of the popular TFHE scheme can perform homomorphic multiplication in less than 50 milliseconds and bootstrapping in less than 20 milliseconds on a 2.6 GHz machine using only a single thread [13]. FHE still has the major drawback of large ciphertext sizes; this problem isn't limited to FHE but still exists for lattice-based cryptography at large.

Most importantly, the pure ZKP approach cannot scale to support privacy-preserving computation on multi-user inputs without utilizing (usually) highly-interactive MPC protocols.[3] Regardless of the particular MPC protocol chosen, users would still be responsible for coordinating the entire computation off-chain themselves, a non-trivial and unnecessary burden.

With FHE, we investigate two flavours. The *single-key* FHE approach, like the pure ZKP approach, readily supports computations with I/O privacy on inputs belonging to the same user. On the other hand, the *multi-key* FHE approach can address the multi-user problem. It follows the same blueprint; users encrypt their own inputs using a multi-key FHE scheme and provide simple ZKPs showing well-formedness of each of their own ciphertexts. Miners can then compute directly on any subset of the users' encrypted inputs. No off-chain coordination needed. No interaction necessary for the computation. No need for the users to even be online.

[3] Several works deal 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 [14], require an additional setup assumption such as pre-dealt correlated randomness [15], assume the existence of a PKI [14], or even rely on a much stronger assumptions, such as indistinguishability obfuscation [16].

In an ambitious forward-looking world, one may entertain the thought of harnessing the power of multi-key FHE to realize these advantages—advantages that cannot be realized using the pure ZKP approach or its MPC-based extensions. Although multi-key FHE is far from being ready for practical deployment, current state-of-the-art provides highly appealing theoretical features—non-interactive circuit computation and one round output decryption [17]—that we can utilize in bringing this dream into reality.

We take a foundational approach to realizing smart contracts with I/O privacy. In particular, we design a privacy-preserving smart contract framework using both single and multi-key FHE along with ZKPs. Crucially, our framework is *modular*, allowing the instantiator to choose the specific FHE and ZKP schemes assuming they satisfy certain security and correctness properties. We elaborate on our contributions in what follows.

New 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 previous definitions of correctness and security (in terms of privacy/ledger indistinguishability and overdraft safety/balance) from Zether [8] and Zerocash [5] 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 future private smart contract constructions.

smartFHE: An FHE-based PPSC Framework. We propose a modular framework to support PPSCs using FHE and ZKPs. We say that our framework is "modular" since it is not bound to particular FHE and ZKP schemes, allowing us to exploit future developments in these areas. To the best of our knowledge, we are the first to use FHE in the blockchain setting.

Specifically, our framework supports smart contracts with I/O privacy, along with payments that hide the users' balances and transfer amount. We offer two modes of operation—public and private—that users can automatically switch between. A user can write any smart contract of his choice. If the contract operates on public accounts, the contract code will operate in the usual way; everything will be public. On the other hand, if the contract operates on private accounts, the system will translate the contract code into operations providing input/output privacy.

Private accounts and their data are stored encrypted on the blockchain. FHE computes over this encrypted data directly, with ZKPs used to prove well-formedness of the initial ciphertexts and conditions on the corresponding plaintext. Since we use FHE, we provide post-quantum security with respect to account privacy [18]. Furthermore, when using a multi-key FHE scheme, we can support arbitrary smart contracts with I/O privacy on inputs belonging to different users. This is in stark contrast to the previous works of Zether [8], Zexe [9], and Zkay [10] which cannot support readily privacy-preserving computation on multi-user inputs.

Operating directly on private—encrypted—accounts results in concurrency issues. That is, any changes on the state of an account renders all pending ZKPs invalid since they are tied to the old state. We resolve this issue by using a locking mechanism reminiscent of a mutex.

We show how the smartFHE framework supports a correct and secure PPSC scheme. Additionally, our framework protects against front-running and replay attacks.

A Harmonious Union. From our work, we observe that FHE and blockchain are well-suited to one another in some regards. Blockchain allows FHE to address the pain point of verifying correctness of homomorphic computation. Users can offload computations to blockchain miners who provide the needed computational power and are financially incentivized to always be online.

A Solution to Verifying Homomorphic Computations. In FHE, the evaluation party is usually *different* from the (encrypted) input owner. Hence, there is no immediate way for the key owner to verify that the produced output is correct (without repeating the entire computation). Despite there being solutions to this problem, the added cost can be prohibitive (verification of even simple homomorphic computations of multiplicative depth 2 can double the cost for the input owner [19]). By using a blockchain, we can solve this problem in a simpler way through consensus. That is, the underlying security assumption of blockchain—namely, that the majority of the mining power is honest—provides guarantees with regards to correctness [20]. Miners re-execute the computation and only accept blocks that agree with what they computed. Thus, the owner of the encrypted data can rest assured that the evaluation party (in this case, the miners) performed the homomorphic computation correctly.

An Always-available and Financially-incentivized Evaluation Party. To successfully outsource computation (as in our FHE-based PPSC setting), we need an evaluation party who is readily available. Permissionless blockchains provide an effective solution; in exchange for monetary incentives, the miners need to be online to handle their mining and consensus duties. Such a role easily extends to performing (paid) computations for users (as in Ethereum [7]). Thus, users can utilize these miners to perform FHE computations which are part of deployed smart contracts.

An Instantiation. We propose an instantiation of a PPSC scheme based on the smartFHE framework. We start with a simpler (and more efficient) construction using single-key FHE. This construction employs current state-of-the-art FHE and ZKP schemes—the lattice-based BGV scheme for FHE [21] and the discrete log proofs construction for ZKP [18]. The latter supports efficient proofs of lattice-based relations using Pedersen commitments which can be used in further range proofs.

Our single-key FHE-based instantiation is trustless—so it does not require any trusted setup process. In addition to private payments, this construction supports arbitrary user-defined smart contracts with I/O privacy for a single user's inputs. We demonstrate how this instantiation can be used to build some popular applications that operate on multi-user inputs using additional logic in the smart contract code.

To remove these limitations, we present another PPSC instantiation using a recent multi-key FHE scheme [17] and discrete log proofs [18]. This construction allows for a larger set of applications with any number of users' inputs. In theory, any arbitrary application can be encoded as a smart contract with proper parameter configuration. None of the specific schemes chosen are binding—we have just used current state-of-the-art. Thanks to the modularity of our framework; more advanced and optimized FHE and ZKP constructions can be used in the future instead.

Applications. With a multi-key FHE based instantiation of smartFHE, one can implement any functionality, with any number of users, as a private smart contract (technical aspects regarding FHE and computation circuit depth are discussed in Section 5). Thus, smartFHE can support any privacy-preserving application that can be cast as a smart contract.

The simpler (and more efficient) single-key FHE based instantiation allows us to realize private payments as well as any functionality that operates on a single user's inputs. However, with additional logic added to the smart contract, this instantiation can even realize some important applications operating on multi-user inputs. As examples, we present two applications—multi-item sealed-bid auctions and private inventory tracking (full details can be found in Appendix A). These applications represent two popular use cases of blockchains—decentralized exchanges that do not involve any trusted party and supply chain management.

Performance Evaluation. We evaluate the performance of our smartFHE instantiation. Perhaps surprisingly, we achieve superior runtime performance on the user's end compared to current state-of-the-art. A private transfer transaction ($\mathsf{tx}_{\mathsf{privtransf}}$) in our scheme takes the user less than 47s on our modest machine (2.3 GHz, 8 GB RAM); the equivalent transaction using Zkay 0.2 [22] takes the user over 70s with a more powerful machine (4.7 GHz, 6 cores, 32 GB RAM). Additionally, a private smart contract computation with 2 private inputs takes the user less than 31 seconds in our system; in Zexe, the equivalent operation takes the user over 52 seconds with a more powerful machine (3.0 GHz, 12 cores, 252 GB RAM) [9]. These results show how our FHE-based PPSC approach can be more efficient than the pure ZKP approach.

1.2 Related Work

Several works have explored privacy in the context of blockchain. We focus on those which have dealt with providing I/O privacy for computation (both in the account and the UTXO model). Hawk [20] was one of the first works to construct

a private smart contract scheme using ZKPs. Hawk requires the involvement of a semi-trusted manager—trusted with protecting the privacy of the users' inputs to the contract, but *not* trusted for execution or correctness. Ekiden [23] replaces a semi-trusted manager with trusted hardware to support private smart contracts. Our scheme does not use any such semi-trusted manager or trusted hardware.

Subsequent works avoid the need for such a semi-trusted manager or trusted hardware. Among them, Zether [8] 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. In contrast, the smartFHE framework can support arbitrary computation with I/O privacy on multi-user inputs. Although Zether supports anonymity, this feature cannot be implemented on Ethereum as the cost exceeds the gas limit per block [8].

Zkay [10] 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. Concerningly, it does not address concurrency issues related to operating on private (encrypted) accounts. Nonetheless, we view Zkay's language and compiler as compatible with our smartFHE framework; these can be exploited to implement automatic conversion of a smart contract code into public or private format based on the types of accounts referenced in the code.

Zexe [9] takes privacy further by also supporting function privacy (i.e., hiding the computation itself). However, Zexe works in the UTXO-based model which restricts the supported functionality. Since it follows the pure ZKP paradigm, Zexe suffers from the limitations previously discussed. Finally, Kachina [24] formally defines and models private smart contracts. Since their protocol follows the pure ZKP approach, it cannot readily support private computations on multi-user inputs.

2 Preliminaries

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

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.

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.

For ZKPs, we use $\{(x, y; z) : f(x, y, z)\}$ to mean that the prover shows knowledge of x, y, z (where x, y are public variables and z is a private variable) such that $f(x, y, z)$ holds. Lastly, PPT means probabilistic polynomial time and $\mathsf{negl}(\lambda)$ is meant to denote negligible functions.

Overview of Ethereum. Ethereum [7] 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 [2]. 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 total number of operations that can be executed in a single block.

2.1 Fully Homomorphic Encryption

FHE supports computations (addition and multiplication) directly on ciphertexts. All currently known schemes rely on lattice-based cryptography, thus providing post-quantum security guarantees. 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. Furthermore, FHE schemes can be either single- or multi-hop, where the latter permits evaluated ciphertexts (i.e., outputs of homomorphic operations) to be used in further homomorphic computations.

FHE schemes model computation in one of three ways—as boolean circuits, modular arithmetic, or floating point arithmetic [25]. Floating point arithmetic will provide only approximate values and, thus, is a poor choice here since we need precise balance and transfer amounts for our ZKPs. Specifically, we will use the BGV scheme [21] and then the Mukherjee-Wichs multi-key scheme [17] in our smartFHE instantiations.

BGV Scheme. The BGV scheme [21] 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 [26]. We use BGV in our single-key FHE based instantiation of smartFHE. Additional details on the BGV scheme can be found in Appendix B.

Mukherjee-Wichs Multi-key FHE Scheme. The multi-key FHE scheme proposed in [17] extends the single-key GSW scheme [27] to support operating on ciphertexts encrypted under different keys. The GSW scheme represents homomorphic addition/multiplication as matrix addition/multiplication respectively [27]. 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.

2.2 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 [28], [29], and [30]) 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 [18]. We take this approach in our instantiation to obtain small proof sizes (in the single digit kilobyte range). We will then use Bulletproofs [31] to prove properties of the plaintext (such as the committed value being in a particular range). Both of these ZKP systems provide soundness, completeness, and zero-knowledge guarantees and can be made non-interactive using the Fiat-Shamir transform [32]. Additionally, neither requires a trusted setup.

Bulletproofs. This proof system [31] 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 [18], 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 [18].

Short Discrete Log Proofs. This proof system [18] 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 [18]. 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 [26].

2.3 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 [33] in our instantiation, one of three finalists for NIST's post-quantum cryptography standardization competition.

3 Defining a PPSC Scheme

In this section, we define a notion for a privacy-preserving smart contract (PPSC) scheme and cover its basic operation, correctness, and security. Correctness and security are inspired by Zerocash [5] and Zether [8].

We envision a PPSC scheme applied on top of a public smart contract-enabled cryptocurrency (such as Ethereum). It can be viewed as the extension 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 level (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 nonce* $\mathsf{ctr}[\mathsf{pk}] = 0$ *that is incremented with each transaction, 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). If the account is private,* $\mathsf{Bal}[\mathsf{pk}]$ *will be secret. 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 secret.*
  - $\mathsf{tx}_{\mathsf{lock}}$: *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, and inputs (public or secret) $x_1, ..., x_n$ for this circuit. If $x_1, ...x_n$ are public, then apply C as is on these inputs. If $x_1, ..., x_n$ are secret, transform C into an equivalent circuit C' operating on secret inputs and producing secret outputs, 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 $\mathcal{L}$ which includes the state of all accounts, 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 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. After applying* Ops *to* $\mathcal{L}$ *based on these rules, output an updated state* $\mathcal{L}'$*.*

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. The goal of $\mathcal{S}$ is to produce an Ops that leads to an inconsistent ledger state. Thus, in this game, after receiving the public parameters pp, $\mathcal{S}$ samples a ledger state, a list of operations Ops, and two accounts; one is public and the other is private. Ops will be applied to each account starting with the same initial ledger state (for the public account, a public version of Ops will be used, but for the private account, an equivalent private version will be used). $\mathcal{S}$ wins the game, if at the end, these accounts contain different balance values (meaning that the private operations in a PPSC scheme do not produce a consistent output with their public version). A PPSC scheme is correct so long as the advantage of $\mathcal{S}$ in winning the INCORR game is negligible (as defined below).

Definition 2 (Correctness of a 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.*

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. We define a $\mathsf{Ledger\text{-}Indistinguishability\text{-}Game}$ to represent the interaction between $\mathcal{C}$ and $\mathcal{A}$. At some point in the game, $\mathcal{C}$ chooses a bit b at random (choosing between two operations) and the task of $\mathcal{A}$ is to guess which operation was selected (i.e. return b'). As in Zerocash [5], we define public consistency to rule out trivial wins by the adversary. Our definition requires taking

into account not only the new transactions produced from CreateTransaction, but also the computations resulting from Compute.

Definition 3 (Ledger Indistinguishability). *A PPSC scheme Π satisfies ledger indistinguishability if for all* PPT *adversaries $\mathcal{A}$, the probability that $b' = b$ in the Ledger-Indistinguishability-Game is $1/2 + \mathsf{negl}(\lambda)$, where the probability is taken over the coin tosses of both $\mathcal{A}$ and $\mathcal{C}$.*

Informally, overdraft safety ensures that a PPSC scheme Π does not allow an adversary to spend more currency than he owns. To capture this, we also define an Overdraft-Safety-Game game between $\mathcal{C}$ and $\mathcal{A}$. $\mathcal{A}$ wins the game if he manages to spend currency of a value larger than what he rightfully owns.

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

$$\mathsf{val}_{\mathcal{A}\to\mathsf{PK}} + \mathsf{val}_{\mathsf{Insert}} > \mathsf{val}_{\mathsf{PK}\to\mathcal{A}} + \mathsf{val}_{\mathsf{deposit}} \tag{1}$$

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 sent 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 sent from users with addresses in* PK *to $\mathcal{A}$*
- $\mathsf{val}_{\mathsf{deposit}}$ *is the initial amount of currency in accounts owned by $\mathcal{A}$.*

Now, we define security of a PPSC scheme with respect to the previous two definitions.

Definition 5 (Security of a PPSC Scheme). *A PPSC scheme Π = (*Setup*,* CreateAccount*,* CreateTransaction*,* VerifyTransaction*,* Compute*,* UpdateState*) is secure if it satisfies ledger indistinguishability and overdraft safety.*

A detailed version of the formal definitions (for both correctness and security) can be found in Appendix C.

4 The smartFHE Framework

In this section, we present the design of smartFHE, the first PPSC framework to employ FHE in the blockchain model. We begin by defining the smart contract-enabled cryptocurrency architecture that we target, then we outline the functionalities that our framework supports while showing how it handles concurrency issues. Finally, we provide a high-level description of how correctness and security are satisfied given FHE and ZKP. Concrete instantiations of this framework will be discussed in the next section.

4.1 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. fees charged by the miners) associated for each smart contract operation. In this context, we consider an Ethereum-like architecture.

The default operation of smartFHE is the *public* mode—meaning that everything will be logged in the clear on the blockchain (including the account balances, smart contract code, and all transactions) and the smart contract code will operate on public inputs/outputs. 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 supported by Ethereum's network protocol with new types of transactions and cryptographic capabilities to permit operations on private accounts.

As in 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. To differentiate between these two types (since both are represented by public addresses that could be hashes of the actual public keys), a prefix is added to an account address. Namely, we can let 00 indicate a public account address, whereas 11 will indicate a private one.

smartFHE 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).

It should be noted that we are concerned with the privacy of the accounts and user's data rather than the executed functionality (or function privacy as often called in the literature). The smart contract code, even if working on private accounts, is public. Also, in the current version of smartFHE, we do not support anonymity of the users. The users' addresses are public information, explicitly referenced in any transaction performed in the system. Extending smartFHE to support anonymity is a direction for our future work.

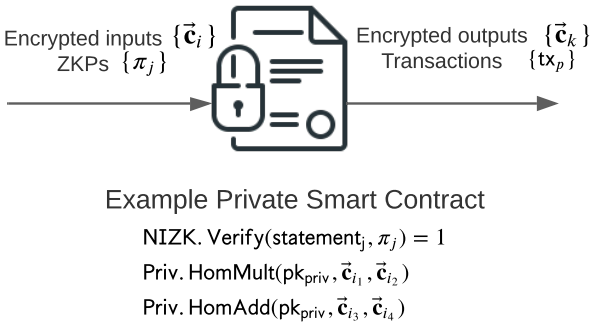

Fig. 1: A high-level description of private smart contract service in smartFHE.

4.2 Supported Functionality

Our framework supports four services: public payments, public smart contracts, private payments, and private smart contracts.

Public Operations. Both public transactions and public smart contracts offer no privacy—with all inputs and outputs provided in the clear. smartFHE handles public operations in the same manner as Ethereum.

Private Payments. For private payments, smartFHE allows users to issue transactions (specifically $\mathsf{tx_{shield}}, \mathsf{tx_{deshield}}, \mathsf{tx_{privtransf}}$) that hide the transfer amount and/or users' balances.

To hide the balance of a private account, we use an FHE scheme to encrypt the balance. Thus, examining the blockchain does not reveal the total amount of currency an account owns. Furthermore, for any private currency transfer transaction, the transfer amount will also be encrypted using this FHE scheme. Doing so allows us to update the account balances of the transaction sender and receiver using the homomorphic addition operation, where the encrypted transfer amount is added to the recipient's encrypted account balance. Since his balance and transfer amount are hidden, the sender will need to provide ZKPs to show that certain conditions are met (i.e. he has enough currency in his account, the transfer amount is non-negative, and the ciphertext is well-formed).

Private Smart Contracts. smartFHE supports arbitrary computations on private inputs belonging to the same user if single-key FHE is used in the framework. If multi-key FHE is used, then smartFHE can support arbitrary computations on private inputs belonging to different users. In both cases, the output produced is private (i.e., encrypted). These functionalities are encapsulated in the $\mathsf{Compute}$ algorithm from Section 3.

Users write smart contracts with code operating on their private data and private account balances (see Figure 1 for a high level pictorial representation).

Since the code may operate on encrypted values, the 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). Operations (aka circuits) over private accounts' data will be translated into homomorphic computations over these private inputs.

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.

4.3 Framework Operations

In what follows, we discuss the setup process of smartFHE and how to handle concurrency issues resulting from distributed computation on private values.

Setup. Setup includes system-related setup, user-related setup, and smart contract-related setup.

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. For a public account, a user generates a key pair (specified by $(\mathsf{pk}_{\mathsf{pub}}, \mathsf{sk}_{\mathsf{pub}})$) to be used for signatures. The public account can be used to initiate public transactions and participate in public smart contracts. For a private account, a user generates a key set for the given FHE scheme (specified by $(\mathsf{pk}_{\mathsf{priv}}, \mathsf{sk}_{\mathsf{priv}})$) along with (signing and verification) keys of the (lattice-based) signature scheme. Private accounts can be used to initiate private transactions and participate in private smart contracts. Note that both account types can be used to deploy smart contracts (public or private).

Smart contract setup is dependent on whoever creates the smart contract code. This code will specify the sorts of inputs the contract functions will accept (which may be encrypted or in the clear), 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 their inputs (public or private) to be operated on.

Handling Concurrency. A challenge in designing privacy-preserving mechanisms for smart contract-enabled systems is how to handle both privacy and concurrency for accounts [24].

Suppose Alice submits a private transaction to be processed by the miners, which involve ZKPs with respect to the current state of her account. While this transaction is still pending, Bob submits his own transaction in which he transfers currency to Alice. If Bob's transaction is processed *first* by the miners,

then Alice's transaction will be invalidated since her account's state has changed and the ZKP is no longer valid. We refer to this issue as "front-running;" here, Bob has front-run Alice's transaction [8].

We introduce two different techniques to address front-running: automatic balance rollovers to handle private transactions and a private account locking mechanism to handle concurrency conflicts resulting from private smart contracts.

Automatic Rollovers. Using this technique, which is similar to the one introduced in [8], 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 [8] 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. Public accounts do not have to worry about such issues; additionally, incoming funds to public accounts are not subject to a waiting period since no ZKPs are involved.

When Alice and Bob submit their own respective transactions, miners will verify the transaction against the sender's current balance at that point in time. 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. smartFHE supports more than just private currency transfer so the automatic rollover mechanism does not suffice for handling front-running in private applications spanning multiple (or even an undetermined number of) epochs.

To address multi-epoch concurrency, smartFHE enables private accounts to be *locked* to other accounts (via $\mathsf{tx}_{\mathsf{lock}}$). A user can lock her account to another account of *any* type, including contract accounts. We refer to the owner of this locked account as the "locker;" the account to which it is locked is the "lockee." 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.

As an optimization of the smartFHE design, epochs can be eliminated entirely using the above locking mechanism. 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.[4]

4.4 Security and Correctness

Our smartFHE framework realizes a correct and secure PPSC scheme based on the notions introduced in Section 3. Full details can be found in Appendix C.

Theorem 1. *Assuming (a) the signature scheme is correct, (b) the (single or multi-key) fully homomorphic encryption scheme is correct and semantically secure, and (c) the ZKP proof system satisfies soundness, completeness, and zero-knowledge properties, then smartFHE realizes a correct (cf. Definition 2) and secure (cf. Definition 5) PPSC scheme (cf. Definition 1).*

5 Our Instantiation

We provide an instantiation of a PPSC scheme using the smartFHE framework. For our single-key instantiation, we use the BGV scheme [21], Bulletproofs [31], and short discrete log proofs [18]. For the multi-key instantiation (Section 5.3), we use the Mukherjee-Wichs multi-key FHE scheme [17] along with discrete log proofs.

5.1 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 B.

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 2.

Public Account Related. A public account owner maintains 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 [34] 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 $\perp$.

[4] 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).

$\mathsf{System.Setup}(1^\lambda, 1^L)$: 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:

- $\mathsf{pp.BGV} \leftarrow \mathsf{BGV.Setup}(1^\lambda, 1^L)$
- $\mathsf{pp.NIZK_{logproofs}} \leftarrow \mathsf{NIZK_{logproofs}.Setup}(1^\lambda)$
- $\mathsf{pp.NIZK_{bulletproofs}} \leftarrow \mathsf{NIZK_{bulletproofs}.Setup}(1^\lambda)$
- $\mathsf{pp.sig_{priv}} \leftarrow \mathsf{PrivSig.Setup}(1^\lambda)$, setup for signature scheme used for private accounts.
- $\mathsf{pp.key_{pub}} \leftarrow \mathsf{PubKey.Setup}(1^\lambda)$, 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. 2: System setup.

3. $\mathsf{Pub.Sign}(\mathsf{sk_{pub}}, \mathsf{m})$: Produces a signature σ_{pub} on message m with secret key $\mathsf{sk_{pub}}$.
4. $\mathsf{Pub.VerifySig}(\mathsf{m}, \sigma_{\mathsf{pub}}, \mathsf{pk_{pub}})$: Verifies a signature σ_{pub} on message m using $\mathsf{pk_{pub}}$.

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

1. $\mathsf{Priv.CreateAccount}(\mathsf{pp})$: This algorithm creates a private account. It outputs the account key pair $(\mathsf{pk_{priv}}, \mathsf{sk_{priv}})$, which are the keys for the BGV scheme, along with the keys for the Falcon signature scheme $(\mathsf{sigpk_{priv}}, \mathsf{sigsk_{priv}})$. The public key $\mathsf{pk_{priv}}$ 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 [21].

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 5.2.

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), and 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.

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}.

5.2 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 [35]) 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 (i.e., 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. The 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 [18]. 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: $\vec{\mathbf{b}}' \stackrel{?}{=} \vec{\mathbf{b}} - \vec{\mathbf{c}}$
– Range 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 $[0, \mathsf{MAX}]$ [5]
- 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:

- 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:

$$\begin{pmatrix} \mathbf{A} \\ \mathbf{B} \end{pmatrix} \cdot \vec{\mathbf{m}} = \begin{pmatrix} \vec{\mathbf{c}} \\ \vec{\mathbf{c}}' \end{pmatrix} \tag{2}$$

This equation verifies that $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}'}$ do in fact encrypt the same amount ammt 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 [18]. 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 $\vec{\mathbf{b}}' + \vec{\mathbf{c}}'$.

5.3 Supporting Multi-user Inputs using Multi-key FHE

Given the modularity of smartFHE's design, all syntax from Section 5.1 will be re-used for the multi-key instantiation (with the exception of $\mathsf{Priv.Refresh}$ which

[5] 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 [36].

will be replaced by an "expanding" procedure). Instead of generating keys using the BGV scheme, users generate keys from the Mukherjee-Wichs scheme [17] when calling Priv.CreateAccount and use these to encrypt their private account balances and inputs. Private payments and private smart contracts use the same syntax and semantics as in a single-key FHE based instantiation. However, now private (smart contract) operations can take in data encrypted under different keys too.

Since the Mukherjee-Wichs scheme relies on the hardness of LWE, we can even re-use the same proof systems (i.e. short discrete log proofs and Bulletproofs). Unfortunately, the construction takes a large efficiency hit when proving LWE relations (instead of Ring-LWE relations) with short discrete log proofs. The proof would require d^2 exponentiations where d is the dimension of the public matrix $\mathbf{A}$ from Section 2.2—so that discrete log proofs would now take on the order of minutes instead of seconds to generate [18]. The Pedersen commitment obtained from short discrete log proofs would then be re-used for the Bulletproofs commitment.

If homomorphic computations were performed on inputs belonging to multiple users, these users would need participate in a one round decryption process. Finally, the same optimizations are possible for the Mukherjee-Wichs based smartFHE instantiation; if we assume circular security, users can use the same keys for all levels. Additionally, bootstrapping can be used to avoid having to specify the number of levels in advance [17].

6 Performance Evaluation

We evaluate the computational and storage cost of the various operations in smartFHE to show feasibility of system deployment. To this end, we focus on the single-key FHE based instantiation from Section 5 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 a private smart contract computation.

For FHE, we use Microsoft's SEAL library [35] to benchmark the BFV scheme [37], an FHE scheme closely related to BGV but much simpler to work with.[6] For the Falcon signature scheme, we use its reference implementation [33] with NIST Level I security (i.e. 128 bits). For Bulletproofs, we use the Dalek library [39] with 32-bit range proofs. For elliptic curve operations, we use Curve25519.

[6] Specifically, BFV only has relinearization as part of the "refresh" procedure. However, a fully optimized BGV implementation will likely outperform the BFV equivalent [38].

Table 1: FHE performance for BFV scheme using the smallest and largest default parameters.

Operation	$d = 1024$	$d = 32768$
KeyGen	0.364 ms	18.963 s
Priv.Encrypt	0.471 ms	60.608 ms
Priv.Decrypt	0.063 ms	31.815 ms
Priv.HomAdd	0.002 ms	0.888 ms
Priv.HomMult	0.587 ms	318.881 ms
Priv.Refresh	Not supported	196.835 ms

Table 2: Private transaction costs in smartFHE.

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

Finally, for short discrete log proofs, the original work provides no implementation but shows how to estimate proof size and times using cycle count [18]. 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.

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) supported by SEAL for BFV [35]. As shown, even for very large parameters, FHE operations are quite fast for both users and miners. We use these results to compute the cost of private transactions and smart contracts.

We measure the overhead of the main transactions in our system—shield, deshield, and private transfer (we use the results for $d = 1024$ from Table 1). 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. Nevertheless, in comparison to Zkay, our scheme is superior. As reported in [22] (using a more powerful machine of 4.7 GHz, 6 cores, 32 GB RAM), a private transfer takes 70s.

For private smart contracts, the user provides encrypted inputs (via FHE) and ZKPs showing these encrypted inputs are well-formed (via log proofs) and in the correct range for the application (via Bulletproofs).

We consider a computation offering I/O privacy implemented within a (private) smart contract. We consider a computation with two private inputs, although the following logic applies for an arbitrary number of private inputs; the

user encrypts these values (0.942 ms), provides up to two proofs each using discrete log proofs (30.528 s) and Bulletproofs (16.01 ms), and signs the transaction (0.2 ms). This gives an estimated cost of 30.55s for the user (using $d = 1024$ for BFV). In contrast, to perform a private computation on two private inputs using Zexe, the user will take over 52s (represented as Execute in their work) on a more powerful machine of 3.0 GHz, 12 cores, and 252 GB of RAM [9]. These results show how our FHE-based PPSC approach can be more efficient than the pure ZKP approach.

7 Conclusion

In this paper, we defined a notion for a PPSC scheme and introduced smartFHE as a modular framework for realizing secure and correct PPSCs. smartFHE is the first work to investigate the utility of FHE in the blockchain model. Users can ask miners to execute arbitrary computations providing I/O privacy, where such computations can even operate on inputs belonging to different users. Our framework supports both public and private modes with respect to payments and smart contracts. In comparison with pure ZKP based approaches, our benchmarks show that an instantiation of smartFHE can perform private payments and computations at a faster rate. Such results demonstrate the potential viability of FHE-based PPSCs.

References

1. D. Chaum, "Blind signatures for untraceable payments," in *Advances in cryptology*. Springer, 1983, pp. 199–203.
2. S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2019.
3. "Coin market capital," https://coinmarketcap.com/.
4. A. Biryukov and S. Tikhomirov, "Deanonymization and linkability of cryptocurrency transactions based on network analysis," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 172–184.
5. E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 459–474.
6. S. Noether, "Ring signature confidential transactions for monero." *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 1098, 2015.
7. G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014.
8. B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, "Zether: Towards privacy in a smart contract world," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 423–443.
9. S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 820–837.
10. S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. Vechev, "zkay: Specifying and enforcing data privacy in smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1759–1776.

11. D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: data structures and implementation." in *NDSS*, vol. 14. Citeseer, 2014, pp. 23–26.
12. R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 142–156.
13. I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe."
14. S. Halevi, Y. Lindell, and B. Pinkas, "Secure computation on the web: Computing without simultaneous interaction," in *Annual Cryptology Conference*. Springer, 2011, pp. 132–150.
15. U. Feige, J. Killian, and M. Naor, "A minimal model for secure computation," in *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, 1994, pp. 554–563.
16. S. Halevi, Y. Ishai, A. Jain, I. Komargodski, A. Sahai, and E. Yogev, "Non-interactive multiparty computation without correlated randomness," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 181–211.
17. P. Mukherjee and D. Wichs, "Two round multiparty computation via multi-key fhe," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016, pp. 735–763.
18. R. del Pino, V. Lyubashevsky, and G. Seiler, "Short discrete log proofs for fhe and ring-lwe ciphertexts," in *IACR International Workshop on Public Key Cryptography*. Springer, 2019, pp. 344–373.
19. D. Fiore, R. Gennaro, and V. Pastro, "Efficiently verifiable computation on encrypted data," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 844–855.
20. A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.
21. Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," Cryptology ePrint Archive, Report 2011/277, 2011, https://eprint.iacr.org/2011/277.
22. N. Baumann, S. Steffen, B. Bichsel, P. Tsankov, and M. Vechev, "zkay v0. 2: Practical data privacy for smart contracts," *arXiv preprint arXiv:2009.01020*, 2020.
23. R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.
24. T. Kerber, A. Kiayias, and M. Kohlweiss, "Kachina-foundations of private smart contracts." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 543, 2020.
25. R. A. Hallman, K. Laine, W. Dai, N. Gama, A. J. Malozemoff, Y. Polyakov, and S. Carpov, "Building applications with homomorphic encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2160–2162.
26. V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 1–23.

27. C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*. Springer, 2013, pp. 75–92.
28. J. Bootle, V. Lyubashevsky, and G. Seiler, "Algebraic techniques for short (er) exact lattice-based zero-knowledge proofs," in *Annual International Cryptology Conference*. Springer, 2019, pp. 176–202.
29. J. Bootle, V. Lyubashevsky, N. K. Nguyen, and G. Seiler, "A non-pcp approach to succinct quantum-safe zero-knowledge," in *Annual International Cryptology Conference*. Springer, 2020, pp. 441–469.
30. T. Attema, V. Lyubashevsky, and G. Seiler, "Practical product proofs for lattice commitments." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 517, 2020.
31. B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 315–334.
32. A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the theory and application of cryptographic techniques*. Springer, 1986, pp. 186–194.
33. P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based compact signatures over ntru," *Submission to the NIST's post-quantum cryptography standardization process*, 2018.
34. D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.
35. "Microsoft SEAL (release 3.5)," https://github.com/Microsoft/SEAL, Apr. 2020, microsoft Research, Redmond, WA.
36. Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," *SIAM journal on computing*, vol. 38, no. 1, pp. 97–139, 2008.
37. J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.
38. A. Costache, K. Laine, and R. Player, "Evaluating the effectiveness of heuristic worst-case noise analysis in fhe," in *European Symposium on Research in Computer Security*. Springer, 2020, pp. 546–565.
39. "Dalek library," https://github.com/dalek-cryptography/bulletproofs.
40. S. Parsons, M. Marcinkiewicz, J. Niu, and S. Phelps, "Everything you wanted to know about double auctions, but were afraid to (bid or) ask," *City University of New York: New York2005*, 2006.

A Applications

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

A.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 [40]. 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.[7] To settle the auction, the auctioneer (which will be a smart 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 system in smartFHE's single-key instantiation, consisting of short discrete log proofs [18] and Bulletproofs [31].

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 homomor-

[7] 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.

phically 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.

A.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).

B FHE Definitions

B.1 Basic Ring-LWE Encryption Scheme

The basic Ring-LWE public key encryption scheme [26] forms the basis of the BGV (fully homomorphic encryption) scheme [21].

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 [18].

$\mathsf{E.Setup}(1^\lambda, 1^\mu)$: 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 [21].

$\mathsf{E.SecretKeyGen}(\mathsf{e.pp})$: 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 [26]. 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 [26].

B.2 BGV Scheme

We present a simplified description of the BGV scheme below. For full details, please see [21].

Recall that each time we perform a homomorphic operation, the noise associated with the ciphertext grows. 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 [21]. 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.

1. $\mathsf{BGV.Setup}(1^\lambda, 1^L)$: The setup algorithm 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 BGV.Refresh.
2. $\mathsf{BGV.KeyGen}(\{\mathsf{bgv.pp_j}\})$: The key generation algorithm 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 BGV.Refresh.
3. $\mathsf{BGV.Encrypt}(\mathsf{bgv.pp}, \mathsf{pk}, \mathsf{m})$: The encryption algorithm BGV.Encrypt takes as inputs the scheme's parameters 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 BGV.Decrypt takes as inputs the scheme's parameters 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 [21]. 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}.

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

C Our Definitions and Proofs

In this section, we define notions for correctness and security of a PPSC scheme. We then show how smartFHE satisfies these notions. As mentioned earlier, our definitions are inspired by those in Zerocash [5] and Zether [8].

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 (i.e. operations inherited from a public smart contract scheme which is Ethereum for smartFHE) 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 private ones such that Ops and Ops' correspond to the same functionality (i.e. produce identical state changes). 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 an 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 6 (Definition 2 revisited). *A PPSC scheme Π = (*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*

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

where $\mathsf{Adv}^{\mathsf{INCORR}}_{\Pi,\mathcal{S}} := \Pr[\mathsf{INCORR}(\Pi, \mathcal{S}, 1^n) = 1]$ *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 the persistent storage variables of the smart contract account (the smart contract containing C's code).

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 in the ledger state, 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 smartFHE. 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 smartFHE 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 system, the

correctness of the FHE scheme, 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 Compute requests. For computations on private inputs, the correctness of the results is based on the correctness of the chosen FHE scheme.

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. Based on this discussion, we have the following lemma (we omit the formal proof which follows from the logic above):

Lemma 1. *Assuming the signature scheme is correct, the (single or multi-key) fully homomorphic encryption scheme is correct, and the proof system satisfies the completeness property, then smartFHE satisfies correctness (cf. Definition 2).*

C.2 Security

Our security definitions for overdraft safety and ledger indistinguishability are similar to those in Zerocash [5] and Zether [8]. However, we make the appropriate changes to take into account our different account types, transaction types, and the additional Compute functionality we defined for PPSC.

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 the oracle; however, $\mathcal{A}$ has full view of the oracle $\mathcal{O}_{\mathsf{PPSC}}$.

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 $\mathsf{CreateAccount}$, $\mathcal{C}$ will send *only* the resulting EOA address and public key to $\mathcal{A}$
  - For $\mathsf{Compute}$, $\mathcal{C}$ 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 $\mathsf{UpdateState}$
- $\mathsf{UpdateState}$, 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 $\mathsf{UpdateState}$, 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 $\mathsf{Overdraft\text{-}Safety\text{-}Game}$ game in which $\mathcal{C}$ and $\mathcal{A}$ interact in the same manner as they do in $\mathsf{Security\text{-}Game}$. $\mathcal{A}$ 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 7 (Definition 4 revisited). *A PPSC scheme Π provides overdraft safety if for all* PPT *adversaries $\mathcal{A}$, the probability that*

$$\mathsf{val}_{\mathcal{A}\to\mathsf{PK}} + \mathsf{val}_{\mathsf{Insert}} > \mathsf{val}_{\mathsf{PK}\to\mathcal{A}} + \mathsf{val}_{\mathsf{deposit}} \tag{3}$$

in the $\mathsf{Overdraft\text{-}Safety\text{-}Game}$ 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 sent 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 sent 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 smartFHE. We now show how the smartFHE framework 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. The nonce associated with each account will enforce order on the pending transactions and prevent $\mathcal{A}$ from double-spending. Additionally, the smartFHE framework 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.

Lemma 2. *Assuming the proof system satisfies soundness and smartFHE satisfies correctness, then smartFHE provides overdraft safety (cf. Definition 4).*

In 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, 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, VerifyShield will output 0.

In 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 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 system (which happens with at most negligible probability).

In PrivTransfer, the state of the sender and receiver's accounts are private. As part of 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 system (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 beyond what was publicly revealed. We define a Ledger-Indistinguishability-Game to capture this. It is the same as Security-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 8 (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, recipient, and nonce value.*

– *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 9 (Definition 3 revisited). *A PPSC scheme Π satisfies ledger indistinguishability if for all* PPT *adversaries $\mathcal{A}$, the probability the $b' = b$ in the Ledger-Indistinguishability-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 smartFHE. The following lemma states that smartFHE satisfies the ledger indistinguishability property.

Lemma 3. *Assuming the proof system satisfies the zero-knowledge property and the (single or multi-key) fully homomorphic encryption scheme is semantically secure, then smartFHE satisfies ledger indistinguishability (cf. Definition 3).*

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 ZKP system employed in smartFHE.

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 system. Additionally, note that the ciphertexts of the transfer amounts are computationally indistinguishable from random assuming the FHE scheme is semantically secure. Thus, with overwhelming probability, they will not reveal any additional information that may help $\mathcal{A}$ in guessing b correctly.

C.3 Proof of Theorem 1 — Correctness and Security of smartFHE

Follows from Lemmas 1, 2, and 3.