smartFHE: Privacy-Preserving Smart Contracts from Fully Homomorphic Encryption

Ravital Solomon[1], Rick Weber[1] and Ghada Almashaqbeh[2]

[1] Sunscreen, {ravital, rick}@sunscreen.tech
[2] University of Connecticut, ghada@uconn.edu

Abstract. Despite the great potential and flexibility of smart contract-enabled blockchains, building privacy-preserving applications using these platforms remains an open question. Existing solutions fall short since they ask end users to coordinate and perform the computation off-chain themselves. While such an approach reduces the burden of the miners of the system, it largely limits the ability of lightweight users to enjoy privacy since performing the actual computation on their own and attesting to its correctness is expensive even with state-of-the-art proof systems.
To address this limitation, we propose *smartFHE*, a framework to support private smart contracts using fully homomorphic encryption (FHE). To the best of our knowledge, smartFHE is the first to use FHE in the blockchain model; moreover, it is the first to support arbitrary privacy-preserving applications for lightweight users under the same computation-on-demand model pioneered by Ethereum. smartFHE does not overload the user since miners are instead responsible for performing the private computation. This is achieved by employing FHE so miners can compute over encrypted data and account balances. Users are only responsible for proving well-formedness of their private inputs using efficient zero-knowledge proof systems (ZKPs). We formulate a notion for a privacy-preserving smart contract (PPSC) scheme and show a concrete instantiation of our smartFHE framework. We address challenges resulting from using FHE in the blockchain setting—including concurrency and dealing with leveled schemes. We also show how to choose suitable FHE and ZKP schemes to instantiate our framework, since naively choosing these will lead to poor performance in practice. We formally prove correctness and security of our construction.
Finally, we conduct experiments to evaluate its efficiency, including comparisons with a state-of-the-art scheme and testing several private smart contract applications. We have open-sourced our (highly optimized) ZKP library, which could be of independent interest.

1 Introduction

Cryptocurrency can be traced back to (at least) 1983 when Chaum first proposed the concept of *electronic cash* using blind signatures [25]. Extending Chaum's design, Bitcoin [60] removed the need for a trusted party and introduced the notion of a public distributed ledger called blockchain through which users could exchange currency directly with one another. Unfortunately, Bitcoin provides no privacy for the user as transaction records are fully public on the blockchain [14]; thus, several initiatives emerged to bring privacy to currency transfer [62, 65].

Smart contracts and privacy. In parallel, a very different question about Bitcoin's functionality was asked. Could Bitcoin be extended to support arbitrary user-defined applications? The answer was *yes* but with major changes to its UTXO-based design. Thus, Ethereum was born, defining an account-based model and a Turing-complete scripting language that permit users to deploy arbitrary programs called smart contracts [70]. Although Ethereum offers a highly expressive functionality, it provides no privacy out of the box.

Over the last few years, several attempts have been made to support private computation for a single user's inputs over a blockchain. Some constructions built upon the paradigm used for private currency transfer—operating directly on additively homomorphic encryptions with zero knowledge proofs (ZKPs) to prove that inputs satisfy certain conditions [22]. Unfortunately, additive homomorphisms enable only a limited set of applications with input/output (I/O) privacy. Other constructions offload *all* work to the end user to do offline (or off-chain) [19, 68]. Users perform the intended computations on plaintext data, encrypt inputs and outputs, and create a ZKP certifying correctness of computation with respect to these encryptions. Blockchain miners only verify correctness of the ZKP. This is referred to as the *ZKP-based approach* as it relies on the power of ZKPs to perform computations with I/O privacy [8].

The ZKP-based approach is not suited to lightweight users [8]. This is due to the fact that generating a proof to certify computation correctness, for even simple computations, is incredibly time and memory intensive even with state-of-the-art constructions. For example, using the highly optimized implementation of Zexe [19] with a non-universal proof system [44], the user needs over 50 s to generate the proof showing correctness of off-chain computation on 2 inputs. Furthermore, even if a very efficient ZKP system is used, operating on private inputs coming from different users (alternatively known as operating on foreign data [67]) requires these users to use a privacy-preserving function evaluation protocol (e.g. multiparty computation [52]). This in turn requires interaction and coordination between the users, along with performing a distributed protocol to generate the computation correctness proof (perhaps embedded inside the private computation offline protocol).

Both industry and academia have become increasingly focused on supporting lightweight users in the blockchain setting. Reducing the user workload while computing over her private inputs has led to considering fully homomorphic encryption (FHE) based approaches. Despite general belief that FHE is quite inefficient, FHE has been used to successfully support lightweight clients attempting to retrieve relevant transactions [54, 57] (the latter, Spiral [57], is deployed in practice [6]). Extending this line of work, we ask *if FHE might be used to support private computation in the blockchain model for lightweight users.* We focus on building a privacy-preserving version of Ethereum's computing-on-demand model.

1.1 Our Contributions

We propose smartFHE, a framework for building smart contracts that supports lightweight users. Operating directly on encrypted values has proven invaluable across numerous applications [24, 33]. Using FHE, users could supply encrypted inputs along with a simple ZKP showing their well-formedness and that certain relations on the plaintexts are satisfied. Miners check the proofs and perform the requested computations directly on the encrypted inputs. No need for users to provide complex ZKPs attesting to the correctness of the entire computation.

Combining FHE with blockchain represents a *harmonious union.* Blockchain addresses the pain point of verifying correctness of homomorphic computation. In FHE, the evaluation party is *different* from the (encrypted) input owner and there is no immediate way for the input owner to validate correctness of the computation (without repeating the entire computation). Although solutions exist for this problem, the added cost can be prohibitive [39]. A blockchain offers a simpler solution through consensus; the assumption that the majority of the mining power is honest provides guarantees with regards to correctness [50]. Moreover, a blockchain solves another problem for computation outsourcing—namely, the need for an always-available evaluation party. Miners perform paid computations for users (as in Ethereum) which could be the FHE computations needed in private smart contracts.

We take a foundational approach to realizing smart contracts with I/O privacy employing FHE and ZKPs. To the best of our knowledge, smartFHE is the first to use FHE in the blockchain model; it is moreover the first to support lightweight users in preserving input/output privacy across arbitrary decentralized applications. Although combining FHE with ZKPs to support private computation appears natural, in the context of large scale distributed systems like blockchain, this solution introduces several challenges (including efficiency and concurrency) that we address in our framework. We elaborate on our contributions in what follows.

A notion for privacy-preserving smart contracts. We define a notion for privacy-preserving smart contracts (PPSCs) capturing the support of arbitrary computation with I/O privacy. Furthermore, we extend existing definitions of correctness and security [22, 65], in terms of privacy/ledger indistinguishability and overdraft safety/balance, to provide formal guarantees for a PPSC scheme. We believe that our PPSC definition is of independent interest as it is general enough to be used in other private smart contract constructions.

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

smartFHE is highly flexible with respect to functionality. First, it offers two modes of operation—public and private—that users can switch between. Private accounts and their data are stored encrypted on the blockchain and users supplement any additional encrypted inputs with proper ZKPs attesting to well-formedness. FHE allows miners to operate directly on these private inputs, produce private outputs, and update the blockchain state accordingly. Second, our framework is *modular* since it is not bound to particular FHE and ZKP schemes, allowing us to exploit future improvements in these areas. Third, our framework and its security notions establish rigorous foundations based on which other FHE flavors can be used to support private smart contracts, such as multi-party and multi-key FHE [15, 58, 59].

smartFHE is highly versatile with respect to applications as it supports operation on foreign values. Our instantiation of smartFHE allows us to realize private payments and arbitrary computations over a single user's inputs, e.g. automated market makers (AMMs) that protect users against front-running attacks. Furthermore, with additional smart contract logic, it can even realize some important applications operating on multi-user inputs such as statistical data analysis for financial purposes (details can be found in Appendix B).

smartFHE instantiation. We provide an instantiation of our framework and formally prove its correctness and security based on our PPSC definition. Working with FHE in the blockchain setting is non-trivial if we require efficiency. Selection of an FHE scheme must be done carefully with consideration of the need for exact computation, fast integer arithmetic, and high levels of precision. Additionally, we encounter concurrency issues and must get around bootstrapping as the best suited FHE schemes for the blockchain setting do not offer fast bootstrapping.[3]

The most obvious path forward to proving the lattice-based relations of FHE is via lattice-based ZKPs. However, state-of-the-art lattice-based ZKPs [16,17] tend to be hundreds of kilobytes to single digit megabytes in size and are not nearly as space efficient as recently proposed elliptic curve-based ZKPs [23, 28]. To address this challenge, we utilize a recent elliptic-curve based ZKP system [34] that allows for proving certain lattice-based relations. As [34] proves relations with respect to a Pedersen commitment, we can employ another elliptic-curve based ZKP—Bulletproofs [23]—to prove further relations over private inputs quite efficiently.

To demonstrate feasibility, we evaluate the performance of our instantiation to show how our approach indeed supports lightweight users. In terms of private payments, smartFHE allows a user to issue payments at a rate that is 1.16x−7.79x (depending on parameter choice) faster than Veri-zexe, a state-of-the-art scheme. We conduct experiments testing our instantiation in supporting private smart contracts for statistical data analysis and automated market makers (AMMs). Transaction generation time for these applications is smaller than that of Veri-zexe across the board. Our implementation includes developing the first library for short discrete log proofs. Unfortunately, verification time in our system is quite large but can be improved upon significantly with further work into a GPU-accelerated version of short discrete log proofs targeting OpenCL (our implementation targets Apple Metal). We open source our library, which may be of independent interest as it advances the current state-of-the-art on privacy-preserving computing.

1.2 Related Work

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

Hawk [50] is one of the first works to construct a private smart contract scheme using ZKPs. Hawk requires a semi-trusted manager—trusted with protecting the privacy of the users' inputs but not for correctness of computation. Ekiden [26] replaces a semi-trusted manager with trusted hardware, while Arbitrum [48] relies on a full quorum of trusted parties. Subsequent works avoid such (semi-)trusted parties or hardware. Among them, several works [10, 11, 13] (including Appendix G in [50]) improve on

[3] Bootstrapping is needed to move from a leveled FHE scheme that can perform only a certain number of homomorphic computations to an FHE scheme that can perform an unlimited number of homomorphic computations on encrypted data [20].

Hawk by implementing the manager functionality using a multiparty computation protocol with various efficiency optimizations. While Zether [22] targets trustless smart contract privacy for Ethereum, its reliance on additively homomorphic encryption restricts functionality to private currency transfer and a limited class of private smart contracts. Although Zether supports anonymity, this feature cannot be implemented on Ethereum as the cost exceeds the gas limit per block [22].

Zkay [68] and its extension ZeeStar [67] (which targets operation on foreign data) suggest that supporting privacy for smart contracts requires knowledge of advanced cryptographic primitives that the average developer may not possess. They develop a compiler that takes as input a public smart contract and produces a functionality-equivalent one that can operate on private inputs. Zkay follows the ZKP-based approach described earlier, so it overloads end users and requires off-chain coordination to handle multi-user inputs [8]. Zeestar, on the other hand, supports only additive homomorphisms, and it implements a non-universal ZKP system [43] that requires a new setup for each computation or application.[4] Both works do not address concurrency issues related to operating on private (encrypted) accounts. Nonetheless, we view the compiler idea as compatible with the smartFHE framework; a compiler could be used to help implement automatic conversion of smart contract code into public or private format based on the types of accounts used in the code.

Zexe [19] takes privacy further by also supporting function privacy (i.e. hiding the computation itself). Following the ZKP-based approach outlined previously, Zexe operates in the UTXO-based model which restricts the supported functionality to extending Zerocash [65] scripts used to spend currency. Thus, it does not truly support private smart contracts. Furthermore, Zexe uses a non-universal ZKP system [43, 44], meaning that a new setup is needed for each new computation circuit.[5] Zexe will scale poorly if the ZKP is used to attest to changes in the contract state. Kachina [49] seeks to solve this problem by introducing state oracles to reduce the ledger state size involved in a ZKP, in addition to introducing a formal model for private smart contracts. Nonetheless, it still follows the ZKP-based approach; thus, it is not suitable for lightweight users.

Combining FHE with blockchain applications has recently been investigated thanks to optimized implementations of FHE schemes. Oblivious message retrieval [54] allows a client to retrieve all private transactions relevant to her. By utilizing FHE, this can be done in an efficient way; miners will produce a concise response (without knowing the client address or which transactions are of interest) so the client can detect and then retrieve these transactions privately. Spiral [57] utilizes FHE to build a highly efficient single-server private information retrieval (PIR) protocol. This allows a client, interacting with a trustless server holding a private database, to retrieve private records without revealing the access pattern.[6]

While the space has explored other paradigms such as private (enterprise or permissioned) blockchains, where privacy is controlled based on which parties are permitted to join certain application or view certain blockchain logs, our work focuses on cryptographic approaches to preserve privacy in permissionless blockchains. Another paradigm is utilizing differential privacy, which in general adds noise to the user data so that the privacy loss is bounded by some factor [31, 46]. A major challenge here is the privacy-accuracy tradeoff—more noise implies higher privacy level but less accurate results when operating on the data. This limits the kind of applications this approach can support, e.g., it cannot be used for private accounts that require exact computation over their balances.

2 Defining a PPSC Scheme

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

[4] Both Zeestar and Zether rely on ElGamal encryption. Thus, both support only *short* plaintexts (i.e. of length up to 32 bits), meaning that account balances and transferred values cannot exceed this limit. smartFHE, as it relies on FHE, does not have this limitation and can operate on longer plaintext values, e.g. 64 bits.

[5] Veri-zexe [72] addresses this problem by replacing the non-universal ZKP system that Zexe uses with a universal one, namely Plonk [41], allowing for one setup in the system.

[6] PESCA [32] proposes using threshold FHE to support private smart contract state and computation with the secret key shared among the committee handling the consensus. However, it introduces high level ideas rather than a detailed protocol design and analysis.

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

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

Our definition below captures 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*,* $\mathsf{CreateAccount}$*,* $\mathsf{CreateTransaction}$*,* $\mathsf{VerifyTransaction}$*,* $\mathsf{Compute}$*,* $\mathsf{UpdateState}$*) defined as follows:*

- Setup*: Takes as input a security parameter λ and outputs system public parameters* pp*.*
- $\mathsf{CreateAccount}$*: Takes as inputs* pp *and a privacy mode (0 for public and 1 for private). It generates a key pair* $(\mathsf{sk}, \mathsf{pk})$ *and an address* addr *(derived from* pk*),*[7] *and it initializes the account state, balance* $\mathsf{Bal}[\mathsf{pk}] = 0$ *and locking entry* $\mathsf{Lk}[\mathsf{pk}] = \perp$ *($\perp$ means the account is initially unlocked).* $\mathsf{CreateAccount}$ *outputs the key pair, address, and state.*
- $\mathsf{CreateTransaction}$*: Takes as inputs* pp *and transaction related information. Outputs a transaction* tx *of one of the following types:*
  - $\mathsf{tx}_{\mathsf{shield}}$: *Transfers currency from a public account to a private one. Currency amount is public.*
  - $\mathsf{tx}_{\mathsf{deshield}}$: *Transfers currency from a private account to a public one. Currency amount is public.*
  - $\mathsf{tx}_{\mathsf{privtransf}}$: *Transfers currency between private accounts. Currency amount is private.*
  - $\mathsf{tx}_{\mathsf{lock}}$: *Locks a private account to some other account, thereby preventing the locked account balance from being altered until unlocked.*
  - $\mathsf{tx}_{\mathsf{unlock}}$: *Unlocks a private account, returning control back to its owner. Only the account to which the private account was locked can issue this transaction.*
- $\mathsf{VerifyTransaction}$*: Takes as inputs* pp*, transaction* tx*, and the transaction's syntax/semantics for the types mentioned above. Outputs 1 if* tx *is valid based on these syntax/semantics; 0, otherwise.*
- $\mathsf{Compute}$*: Takes as inputs* pp*, a circuit C representing the code to be executed, and circuit inputs $x_1, \dots, x_n$:*
  - *If $x_1, \dots, x_n$ are public, then apply C as is on these inputs.*
  - *If $x_1, \dots, x_n$ are private, transform C into a functionality-equivalent circuit C' operating on private inputs and producing private outputs,*[8] *then apply C' to $x_1, ..., x_n$.*

  Output 1 if computation is successful, and 0 otherwise.[9]
- $\mathsf{UpdateState}$*: Takes as inputs* pp*, current ledger state $\mathcal{L}$, and a list of pending operations* $\mathsf{Ops} = \{\mathsf{op}_i\}$*.*[10] *Changes induced by all operations are reflected at the end of a block except:*
  - $\mathsf{tx}_{\mathsf{shield}}$ *or* $\mathsf{tx}_{\mathsf{privtransf}}$ *are processed at the end of the epoch (i.e. in the last block of the epoch).*
  - *Incoming transactions to a locked account will not be processed until the next epoch after which the account is unlocked.*

  At the end of a block, $\mathsf{UpdateState}$ *outputs an updated ledger state $\mathcal{L}'$.*

Now, we define notions for correctness and security of a PPSC scheme, which are inspired by [22,65]. We make the appropriate changes to take into account our different account types, transaction types, private computation (rather just private payments as in these prior works) and algorithms listed in the PPSC definition.

[7] This address will have a postfix indicating if it is for a private or public account.

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

[9] Note that in practice a compute request is packaged as a transaction containing the target smart contract address, the name of a function inside this contract (the one that C represents), and the inputs.

[10] Note that op_i can be a transaction tx_i or a computation $\mathsf{Compute}(\mathsf{pp}, C_i, \{x_{i,1}, \dots, x_{i,n}\})$ as described above.

2.1 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, the resulting state is also valid. Correctness with respect to public state variables is derived from the correctness of the underlying public system. It 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 accounts store secret values. Although a smart contract's code is public, it is translated into privacy-preserving operations before operating on private accounts so it produces private outputs. Thus, proving correctness requires validating these private operations.

We define an incorrectness game $\mathsf{INCGame}$ between a challenger $\mathcal{C}$ and a ledger sampler $\mathcal{S}$. At a high level, after performing the setup phase by $\mathcal{C}$, $\mathcal{S}$ 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 cover all basic operations in the system. Ops will be applied separately to acc_{pub} (as is) and acc_{priv} (with an equivalent private version of Ops denoted as Ops') starting with $\mathcal{L}$ in each case. Here Ops' corresponds to the same functionality of Ops—produces identical state changes—but it deals with private inputs/outputs instead of public ones (e.g., $\mathsf{tx}_{\mathsf{transf}}$ is replaced with $\mathsf{tx}_{\mathsf{privtransf}}$ and C is transformed into C' mentioned before).

Applying Ops and Ops' will produce two updated ledger states: $\mathcal{L}'_1$ (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 $\mathsf{INCGame}$ game if it can produce a scenario in which the balance of acc_{priv} is not equal to the balance of acc_{pub}.[11]

Definition 2 (Correctness of PPSC Scheme). *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* $\mathsf{INCGame}$ *game with non-negligible probability. That is, for every* PPT *$\mathcal{S}$ and sufficiently large λ, we have:*

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

where $\mathsf{Adv}_{\Pi,\mathcal{S}}^{\mathsf{INCGame}} := \Pr[\mathsf{INCGame}(\Pi, \mathcal{S}, 1^\lambda) = 1]$ *is $\mathcal{S}$'s advantage of winning the incorrectness game, and the probability is taken over all randomness of $\mathcal{C}$ and $\mathcal{S}$.*

We now define the specifications of a valid Ops and the ledger state evolution needed to define the $\mathsf{INCGame}$ game referenced above.

Definition 3 (Specifications of a valid Ops). *Let $\mathsf{Ops} = \{\mathsf{op}_i\}$ be a list of operations sampled by $\mathcal{S}$. 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 on processing delays in our system.

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. 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 $\mathcal{S}$).

Definition 4 (Ledger state evolution). *Let $\mathcal{L}_{i-1}$ be the current ledger state and op_i be the next operation to be processed to produce the i^{th} ledger state $\mathcal{L}_i$. The updates resulting from processing op_i are 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* $\mathsf{Lk}[\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}] = \perp$ *and* $\mathsf{val} + \mathsf{Bal}[\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}] < \mathsf{MAX}$*, then increment* $\mathsf{Bal}[\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}]$ *by* val.

[11] Without loss of generality, to simplify the notions, we focus on the balance value when dealing with states.

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

INCGame Game Definition. The probabilistic experiment INCGame 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 $\mathcal{S}$ 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 Ops (cf. Definition 3), that the two accounts have properly-initialized states that are recorded on the ledger. 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 the private version of Ops (Ops' mentioned earlier) to acc_{priv} with the same initial ledger state $\mathcal{L}$ and produces an updated ledger state $\mathcal{L}'_2$. Ledger state evolution rules are per Definition 4.
5. $\mathcal{C}$ deshields the balance of acc_{priv} on $\mathcal{L}'_2$, denoted as b'. Let b be the balance of acc_{pub} based on $\mathcal{L}'_1$. If $b \neq b'$, $\mathcal{C}$ outputs 1 (meaning that $\mathcal{S}$ won the game), otherwise, it outputs 0.

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

2.2 Security

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

Definition 5 (Security of a PPSC Scheme). *A PPSC scheme Π = (*Setup*,* $\mathsf{CreateAccount}$*,* $\mathsf{CreateTransaction}$*,* $\mathsf{VerifyTransaction}$*,* $\mathsf{Compute}$*,* $\mathsf{UpdateState}$*) is secure if it satisfies overdraft safety (cf. Definition 6) and ledger indistinguishability (cf. Definition 8).*

We define the security games that capture overdraft safety and ledger indistinguishability. Let $\mathcal{A}$ be the adversary; $\mathcal{C}$, the challenger; and $\mathcal{O}_{\mathsf{PPSC}}$, the oracle for a PPSC scheme. All parties receive the security parameter λ as input and are given oracle access to $\mathcal{O}_{\mathsf{PPSC}}$. $\mathcal{O}_{\mathsf{PPSC}}$ maintains the public parameters pp, the system state, and all public keys generated in the system PK (the latter is 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. Any time a query requires a secret key belonging to $\mathcal{C}$ as input, we allow $\mathcal{A}$ to specify the corresponding public key in PK. $\mathcal{O}_{\mathsf{PPSC}}$ supports the following query types:

- $(\mathsf{setup}, 1^\lambda)$: Takes the security parameter λ as input and sets up the system accordingly. This includes creating the ledger and the public parameters needed by all parties/cryptographic building blocks. This query can be called only once by $\mathcal{C}$.
- $(\mathsf{request}, \mathsf{op}, \mathsf{aux})$: Allows $\mathcal{A}$ to request executing any of the user algorithms defined in Definition 1 (represented by op) with certain inputs and any account address of $\mathcal{A}$'s choice (represented by the auxiliary input aux) through $\mathcal{C}$:
  - For CreateAccount, $\mathcal{A}$ will receive *only* the account address and its public key.
  - For Compute, only computations supported by the PPSC system will be performed.
  - Any transaction from a locked account will be rejected.
- $(\mathsf{insert}, \mathsf{op})$: Allows $\mathcal{A}$ to insert its own well-formed transaction or computation request. These (and anything via $\mathsf{request}$) will be held in pending state (or pending operations denoted as Ops) until processed.
- $(\mathsf{execute}, \mathsf{op})$: Allows $\mathcal{A}$ to ask $\mathcal{O}_{\mathsf{PPSC}}$ to process an arbitrary subset of pending operations $\mathsf{op} \subset \mathsf{Ops}$ and update the ledger state.

- $(\mathsf{corrupt}, \mathsf{pk})$: Allows $\mathcal{A}$ to corrupt an account with public key pk, meaning that $\mathcal{A}$ will get the corresponding secret key sk. This command can be invoked at anytime and as many times as $\mathcal{A}$ wishes.

OSGame game. Overdraft safety ensures that a PPSC scheme Π does not allow $\mathcal{A}$ to spend more currency than it owns. To capture this, we define an OSGame game. $\mathcal{A}$ wins the game and, hence, breaks overdraft safety if it manages to spend currency of a value larger than what it rightfully owns. OSGame proceeds as follows:

1. $(\mathsf{setup}, 1^\lambda)$
2. $\mathcal{A}^{\mathcal{O}_{\mathsf{PPSC}}}(1^\lambda)$
3. At the end, $\mathcal{C}$ queries the ledger state through $\mathcal{O}_{\mathsf{PPSC}}$ and outputs 1 (meaning that $\mathcal{A}$ won the game) if:

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

   such that:
   - $\mathsf{val}_{\mathcal{A}\to\mathsf{PK}}$ is the total value of payments confirmed from $\mathcal{A}$ to users with addresses in PK (this results from $\mathcal{A}$ asking honest parties represented by $\mathcal{C}$ to create transactions for it).
   - $\mathsf{val}_{\mathsf{Insert}}$ is the total value of payments placed by $\mathcal{A}$ on the ledger.
   - $\mathsf{val}_{\mathsf{PK}\to\mathcal{A}}$ is the total value of payments confirmed from users with addresses in PK to $\mathcal{A}$.
   - $\mathsf{val}_{\mathsf{deposit}}$ is the initial amount of currency in accounts owned by $\mathcal{A}$.

   Otherwise, $\mathcal{C}$ outputs 0.

Definition 6 (Overdraft Safety). *A PPSC scheme Π provides overdraft safety if no* PPT *adversary $\mathcal{A}$ can win the OSGame game with non-negligible probability. That is, for every* PPT *$\mathcal{A}$ and sufficiently large λ, we have:*

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

where $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathit{OSGame}} := \Pr[\mathit{OSGame}(\Pi, \mathcal{A}, 1^\lambda) = 1]$ is $\mathcal{A}$'s advantage of winning the overdraft safety game, and the probability is taken over all randomness of $\mathcal{C}$ and $\mathcal{A}$.

LINDGame game. Ledger indistinguishability ensures that the ledger produced by a PPSC scheme Π does not reveal additional information about private account state beyond what can be inferred from what is publicly revealed. We define a LINDGame to capture this. It is very similar to OSGame except that at some point in the game, $\mathcal{A}$ will send two publicly consistent operations (a transaction or a compute request) instead of one. $\mathcal{C}$ will randomly choose one of these operations to execute. $\mathcal{A}$ wins the game if it manages to correctly guess which operation was chosen.

We first define the notion of public consistency of two operations, which is needed to rule out trivial wins of $\mathcal{A}$ in the LINDGame game.

Definition 7 (Public Consistency). *Two operations are publicly consistent if:*

- *They refer to the same user algorithm, or computation, with the same public keys and addresses.*
- *For currency transfer between private accounts, only one of the sender or recipient can be corrupt, not both. If either is corrupt, then the transfer amount must be the same.*
- 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.*
- *For private computation requests, the computation in both must be the same and behave similarly on the supplemented inputs, with private output variables that are not owned by $\mathcal{A}$.*[12]

Accordingly, the LINDGame proceeds as follows:

1. $b \xleftarrow{\$} \{0,1\}$
2. $(\mathsf{setup}, 1^\lambda)$
3. $(\mathsf{op}_0, \mathsf{aux}_0, \mathsf{op}_1, \mathsf{aux}_1) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{PPSC}}}(1^\lambda)$
4. If op_0 and op_1 are publicly consistent, then $(\mathsf{execute}, \mathsf{op}_b, \mathsf{aux}_b)$
5. $\mathcal{A}$ accesses the updated ledger state through $\mathcal{O}_{\mathsf{PPSC}}$
6. $\mathcal{A}$ outputs b', if $b' = b$ then return 1 (meaning that $\mathcal{A}$ won), otherwise, return 0.

[12] Public computations are always trivial—they must have same input values, making op_0 and op_1 identical.

Definition 8 (Ledger Indistinguishability). *A PPSC scheme Π supports ledger indistinguishability if no* PPT *adversary $\mathcal{A}$ can win the* $\mathit{LINDGame}$ *game with non-negligible probability. That is, for every* PPT $\mathcal{A}$ *and sufficiently large λ, we have:*

$$\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathit{LINDGame}} < \frac{1}{2} + \mathsf{negl}(\lambda)$$

where $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathit{LINDGame}} := \Pr[\mathit{LINDGame}(\Pi, \mathcal{A}, 1^{\lambda}) = 1]$ *is $\mathcal{A}$'s advantage of winning* $\mathit{LINDGame}$*, and the probability is taken over all randomness of $\mathcal{C}$ and $\mathcal{A}$.*

3 Background and Challenges

In this section, we provide a brief background of the cryptographic primitives employed in smartFHE, along with a brief overview of Ethereum as our framework builds upon its ideas.

Overview of Ethereum. Ethereum [70] 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 [60]. 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 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 execute the code in any smart contract upon request (i.e. when invoked). To prevent DoS attacks, each operation has some associated cost in terms of gas. Additionally, Ethereum's blockchain has a gas limit which constrains the number of operations that can be executed in a single block.

Fully Homomorphic Encryption. A fully homomorphic encryption (FHE) scheme consists of 3 efficient algorithms: KeyGen for generating public/private keys and public parameters needed in the scheme, $\mathsf{Encrypt}$ for encrypting a message m to produce a ciphertext ct, and $\mathsf{Decrypt}$ for decrypting a ciphertext ct to return the plaintext message m. Moreover, an FHE scheme supports addition and multiplication on ciphertexts, thereby allowing for arbitrary computation on encrypted data. If ct_1 is a ciphertext of m_1 and ct_2 is a ciphertext of m_2, then $ct_1 + ct_2 = ct_3$ is a ciphertext of $m_1 + m_2$. Also, $ct_1 \cdot ct_2 = ct_3$ is a ciphertext of $m_1 \cdot m_2$. All known FHE constructions rely on lattice-based cryptography so provide post-quantum security guarantees. Correctness of FHE means that decryption produces the original plaintext that was encrypted, and that homomorphic addition and multiplication produce ciphertexts for valid results. For security, we require the conventional semantic security (against CPA attackers).

Challenges in working with FHE. FHE schemes model computation in one of three paradigms—as boolean, exact arithmetic, or floating point arithmetic [45]. Arithmetic-based schemes (such as BFV [36], BGV [21], CKKS [27]) are almost always used in the leveled format, meaning that only a certain number of operations can be performed on encrypted data. Ciphertexts produced directly from encryption are "fresh." A noise budget is associated with each fresh ciphertext and over the course of computations on this ciphertext is depleted. Once this budget reaches 0, the ciphertext can no longer be decrypted successfully.

Although bootstrapping can used to support truly arbitrary computation on encrypted data, it tends to be very slow for arithmetic-based schemes, often taking a few minutes [69]. Binary schemes offer very fast bootstrapping so can (at face value) support truly arbitrary computation. When using binary schemes (like TFHE [29] or Concrete [30]), bootstrapping is used to realize each operation [69]. For the blockchain setting, we anticipate users needing to perform 32 bit computation to represent account balances and transfer amounts. TFHE and its variants generally struggle to support more than 20 bits of precision [53]; in practice, bootstrapping with even around 20 bits of precision can take a few seconds, making boolean schemes practically inefficient for the blockchain setting. Floating point arithmetic, on the other hand, will provide only approximate values and, thus, is a poor choice for smartFHE since we need precise balance and transfer amounts. While any of the computation paradigms may be used in the smartFHE protocol, each one presents a unique set of tradeoffs that system designers must account for.

Zero Knowledge Proofs. A zero knowledge proof (ZKP) system allows a prover to convince a verifier that it knows a witness w for some statement x without revealing anything about the witness itself beyond what can be inferred by x on its own. A ZKP system consists of 3 algorithms: Setup for generating public parameters, Prove for producing a proof π proving correctness of x given w, and Verify for verifying if a proof π is valid for statement x. A secure ZKP system must satisfy certain properties with respect to soundness, completeness, and zero knowledge. Informally, soundness guarantees a prover cannot convince a verifier of false statements; completeness guarantees that any honestly generated proof will be accepted by the verifier; zero knowledge guarantees that the proof of a statement does not reveal anything about the witness.

Challenges in combining ZKPs with FHE. As mentioned before, all known FHE schemes are lattice-based. Hence, a naive instanitation of our framework may also require the use of a lattice-based ZKP system. Indeed this provides full post-quantum security for private accounts but harms efficiency in terms of storage overhead. Lattice-based ZKPs, although quite fast, tend to result in proofs that are hundreds of kilobytes to single digit megabytes in size [55,61], making them challenging to work with in the blockchain model. An elliptic curve or hash-based proof system likely offers space savings but at the cost of longer proof generation times.

4 smartFHE Framework

In this section, we present the design of smartFHE, a PPSC framework that uses FHE in the blockchain model. We begin by outlining the cryptographic building blocks employed, then we describe the smart contract-enabled cryptocurrency architecture that we target, followed by technical details of our framework. In presenting the framework, we address challenges that result from working with FHE in the blockchain setting, such as dealing with leveled FHE schemes and concurrency.

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. miners' fees) associated with each smart contract operation. Thus, we consider an Ethereum-like architecture.

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

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

smartFHE operation proceeds in rounds (a round is the time needed to mine a block on the blockchain) and epochs (where an epoch is y contiguous rounds for some integer y that is selected during the system setup phase). The latter is needed to handle concurrency issues related to operating on private accounts, as will be shown later. If desired, epochs can be eliminated entirely (which we discuss towards the end of this section).

4.2 The smartFHE Protocol

Our framework is composed of three components: a setup phase to deploy the system, a network protocol defining all extensions required to support private payments/smart contracts, and mechanisms for handling concurrency issues (resulting from operating on private accounts) and dealing with leveled FHE schemes.

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

4.2.2 Network Protocol Syntax In what follows, we informally present the syntax that smartFHE adds to Ethereum's network protocol to support privacy. Full technical details can be found in Section 5).[13]

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

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

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

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

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

Note that $\mathsf{tx}_{\mathsf{shield}}$ and $\mathsf{tx}_{\mathsf{deshield}}$ reveal some information about the private account (i.e. sender/recipient has a balance larger than or equal to the released value) since the transaction amount is public.

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

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

- $\mathsf{c}_3 = \mathsf{Priv.HomAdd}(\mathsf{pk}_{\mathsf{priv}}, \mathsf{c}_1, \mathsf{c}_2)$: Adds ciphertexts c_1 and c_2 (which are encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) together to produce the sum c_3 of the two ciphertexts.
- $\mathsf{c}_3 = \mathsf{Priv.HomMult}(\mathsf{pk}_{\mathsf{priv}}, \mathsf{c}_1, \mathsf{c}_2)$: Multiplies two ciphertexts c_1 and c_2 (which are encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) together to obtain the product c_3.

4.2.3 Handling Concurrency Operating on private states (such as encrypted account balances) introduces concurrency issues. In particular, changes in an account state can invalidate all pending ZKPs tied to this account, thus invalidating all private transactions that rely on these ZKPs. Such a situation can be exploited to perform front-running attacks; Bob can front-run Alice by issuing a transfer transaction that changes Alice's account state and, if this transfer is processed before Alice's pending transactions, her transactions will be rejected. We introduce two complementary techniques to address front-running: (1) automatic balance rollovers for private transactions and (2) a private account locking mechanism for private smart contracts.

Automatic Rollovers. Using this technique, which is similar to the one in [22], 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 [22] which requires users to trigger the rollover). To guarantee that deshielding and private transfer transactions will be processed by the end of the same epoch, private account users are advised to submit such transactions at the beginning of an epoch. The length of an epoch must be chosen carefully to ensure that a transaction submitted at the start of an epoch is processed before the epoch ends. The sender should view the transaction amount as being deducted from his own account balance immediately (to avoid double spending).

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

- $\mathsf{tx}_{\mathsf{lock}} \leftarrow \mathsf{Lock}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{addr}^{\mathsf{to}})$: Checks that $\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}$ is not already locked, and if not, it locks the private account tied to $\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}$ to the account tied to $\mathsf{addr}^{\mathsf{to}}$ (the latter can even be the same account itself). Finally, it outputs $\mathsf{tx}_{\mathsf{lock}} = (\mathsf{addr}^{\mathsf{to}}, \sigma_{\mathsf{lock}})$ where $\sigma_{\mathsf{lock}} = \mathsf{Priv.Sign}(\mathsf{sigsk}_{\mathsf{priv}},\ (\mathsf{addr}^{\mathsf{to}}, \mathsf{ctr}[\mathsf{pk}_{\mathsf{priv}}]))$. Note that funds sent to $\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}$ will not be rolled over onto the account balance until it is unlocked.
- $\mathsf{tx}_{\mathsf{unlock}} \leftarrow \mathsf{Unlock}(\mathsf{pk}_{\mathsf{priv}})$: First, checks the account tied to $\mathsf{pk}_{\mathsf{priv}}$ is locked. If so, it unlocks the private account corresponding to $\mathsf{pk}_{\mathsf{priv}}$ if and only if the address addr that called Unlock is the same one returned by $\mathsf{CheckLock}(\mathsf{pk}_{\mathsf{priv}})$.
- $\mathsf{CheckLock}(\mathsf{pk}_{\mathsf{priv}})$: Checks if the account tied $\mathsf{pk}_{\mathsf{priv}}$ is currently locked. If locked, returns the address of the account it is locked to. Otherwise, returns $\perp$.

As an optimization of smartFHE's design, epochs can be eliminated entirely using the above locking mechanism.[14] 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.[15]

4.2.4 Handling Leveled Schemes

Arithmetic-based FHE schemes are almost always used in the leveled format. It is highly unlikely that a user would know a priori how many transactions she anticipates receiving or how many smart contracts she will interact with. Even with careful selection of FHE scheme parameters, it is entirely possible that a user may run out of noise budget and eventually be unable to decrypt her private account balance. In practice, users must track the noise associated with their private account balance to ensure they do not run out of noise budget. Upon reaching some pre-determined noise threshold, bootstrapping could be used. Below, we present an alternative technique to reset the noise budget that is likely to offer better concrete efficiency than bootstrapping due to advancements in ZKP constructions.

- $\mathsf{b}', \pi \leftarrow \mathsf{Priv.RefreshBal}(\mathsf{sk_{priv}}, \mathsf{b})$: Takes in encrypted account balance b to produce freshly encrypted account balance b'. User decrypts her private account balance b to get plaintext value bal and then freshly encrypts bal to produce b'. User will need to provide a ZKP to show that the underlying plaintext of b' is equal to the one encrypted in b.[16]
- $\mathsf{CheckPriv.RefreshBal}(\mathsf{b}', \pi)$: Updates the user private account balance (whose account corresponds to $\mathsf{pk_{priv}}$) from b to b' if the proof π is valid.

5 Our Instantiation

We now present the full syntax and technical details of our instantiation of the smartFHE framework. We first provide additional notation needed in this section and then an overview of the cryptographic constructions used to instantiate the building blocks smartFHE needs.

Additional notation. We use $\mathbb{Z}_p$ to represent $\mathbb{Z}/p\mathbb{Z}$, the arrow notation for column vectors (e.g., $\vec{v}$), and capital letters for matrices. For polynomials, we use boldface notation (e.g., $\mathbf{v}$), boldface with arrow notation for a vector of polynomials (e.g. $\vec{\mathbf{v}}$), and boldface capital letter for a matrix of polynomials.

Cryptographic primitives. For FHE, we use the leveled BFV scheme [36]. The BFV scheme models computation as arithmetic circuits and supports exact arithmetic. In BFV, the message, ciphertext, and the secret key are vectors over the quotient ring $R = \mathbb{Z}_q(x)/(f(x))$ (where $f(x) = x^d + 1$ and d is a power of 2) whereas the public key takes the form of a matrix over R. We have chosen to use this scheme as it offers incredibly fast integer arithmetic (crucial for private transactions) and easily supports 64-bit computation.

Importantly, BFV relies on the hardness of Ring-LWE [56], so we can use the short discrete log proofs construction [34]. This proof system is elliptic curve based and allows us to fairly efficiently prove knowledge of a short vector $\vec{\mathbf{s}}$ such that $\mathbf{A}\vec{\mathbf{s}} = \vec{\mathbf{t}}$ for public $\mathbf{A}$ and $\vec{\mathbf{t}}$ over the polynomial ring $\mathbb{Z}_q[X]/(g(x))$, where $g(x)$ is a monic, irreducible polynomial of degree d in $\mathbb{Z}[X]$. Such a relation will allow users to attest to the well-formedness of FHE ciphertexts (which are the users' encrypted inputs). In practice, using this proof system, we obtain proofs on the order of single to double digit kilobytes and reasonable proving/verification times. As part of the short discrete log proof construction, we obtain a Pedersen commitment which can then easily be re-used for Bulletproofs [23] which may be needed for private computations. Bulletproofs allows us to prove arbitrary properties of the user's private account balance or private inputs in a fairly efficient manner.

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

[15] Note that we would still keep the Lock, 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).

[16] For floating point or certain binary-based FHE schemes, the relation will be less than or equal. This is due to precision loss in homomorphic computations under these paradigms.

$\mathsf{Setup}(1^\lambda)$: Takes as inputs the security parameter λ. Outputs the system public parameters pp, including:

- $\mathsf{pp.BFV} \leftarrow \mathsf{BFV.Setup}(1^\lambda)$
- $\mathsf{pp.NIZK}_{\mathsf{logproofs}} \leftarrow \mathsf{NIZK}_{\mathsf{logproofs}}.\mathsf{Setup}(1^\lambda)$
- $\mathsf{pp.NIZK}_{\mathsf{bulletproofs}} \leftarrow \mathsf{NIZK}_{\mathsf{bulletproofs}}.\mathsf{Setup}(1^\lambda)$
- $\mathsf{pp.sig}_{\mathsf{priv}} \leftarrow \mathsf{PrivSig.Setup}(1^\lambda)$, setup for signature scheme used for private accounts.
- $\mathsf{pp.sig}_{\mathsf{pub}} \leftarrow \mathsf{PubSig.Setup}(1^\lambda)$, 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 of 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 the system can support. (We require $\mathsf{MAX} << q$, where q is the modulus of the ring R_q, to prevent possible overflow for balance/transfer amounts.)
- E, epoch length.

Fig. 1: System setup.

For digital signatures, we use the (lattice-based) Falcon scheme [40] when issuing transactions from private accounts (for post-quantum protection), and ECDSA [47] when the issuers are public accounts (to be compatible with existing smart contract-enabled blockchains as in Ethereum). If post-quantum security is not a concern, one can use ECDSA for both (so long as different keys are used for private and public accounts) Additional details on all the previous primitives are provided in Appendix A.

5.1 Syntax

We now define the syntax used in our instantiation. Note that all algorithms take as additional inputs the public parameters pp and the system state st_h for the current block height h (but we omit listing it explicitly).

Setup Related. This is the setup for the entire system, which involves choosing the public parameters of all cryptographic building blocks and the initial state of the ledger (details are in Figure 1).

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

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

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

1. $\mathsf{Priv.CreateAccount}(\mathsf{pp})$: This algorithm creates a private account. It outputs the account key pair $(\mathsf{pk}_{\mathsf{priv}}, \mathsf{sk}_{\mathsf{priv}})$, which are the keys for the BFV scheme, along with the keys for the Falcon signa-

ture scheme or ECDSA $(\mathsf{sigpk}_{\mathsf{priv}}, \mathsf{sigsk}_{\mathsf{priv}})$. The public key $\mathsf{pk}_{\mathsf{pub}}$ consists of matrix $\mathbf{A}$ and auxiliary information τ for key switching; the secret key is $\mathsf{sk}_{\mathsf{priv}} = \mathbf{s}$.

2. $\mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}_{\mathsf{priv}}, \mathsf{m})$: Calls $\mathsf{BFV.Encrypt}$ on message m, and outputs ciphertext $\vec{\mathbf{c}}$.
3. $\mathsf{Priv.Decrypt}(\mathsf{pp}, \mathsf{sk}_{\mathsf{priv}}, \vec{\mathbf{c}})$: Decrypts a ciphertext $\vec{\mathbf{c}}$ encrypted under $\mathsf{pk}_{\mathsf{priv}}$ by running $\mathsf{BFV.Decrypt}(\mathsf{pp}, \mathbf{s}, \vec{\mathbf{c}})$.
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 Falcon or ECDSA schemes.
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 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. $\mathsf{Transfer}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{pub}}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{pub}}, \mathsf{amnt})$: 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. $\mathsf{Lock}(\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{addr}^{\mathsf{to}})$: First, checks that the account corresponding to $\mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}$ is not already locked by calling $\mathsf{CheckLock}(\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}})$. If not, locks it to the account corresponding to $\mathsf{addr}^{\mathsf{to}}$ (the latter can even be the same account itself). Finally, outputs $\mathsf{tx}_{\mathsf{lock}} = (\mathsf{addr}^{\mathsf{to}}, \sigma_{\mathsf{lock}})$ where

$$\sigma_{\mathsf{lock}} = \mathsf{Priv.Sign}(\mathsf{sigsk}_{\mathsf{priv}}, (\mathsf{addr}^{\mathsf{to}}, \mathsf{ctr}[\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}]))$$

   Note that funds sent to $\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}$ will not be rolled over onto the account balance until it is unlocked.
4. $\mathsf{Unlock}(\mathsf{pk}_{\mathsf{priv}})$: First, checks that the account corresponding to $\mathsf{pk}_{\mathsf{priv}}$ is locked by calling $\mathsf{CheckLock}(\mathsf{pk}_{\mathsf{priv}})$. If so, unlocks this account if and only if the address addr that called Unlock is the same one returned by $\mathsf{CheckLock}$. Outputs $\mathsf{tx}_{\mathsf{unlock}}$.

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

1. $\mathsf{Priv.HomAdd}(\mathsf{pk}_{\mathsf{priv}}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: Runs $\mathsf{BFV.HomAdd}$ on the ciphertexts $\vec{\mathbf{c}}_1$ and $\vec{\mathbf{c}}_2$ (which are encrypted under $\mathsf{pk}_{\mathsf{priv}}$) to produce the sum $\vec{\mathbf{c}}_3 = \vec{\mathbf{c}}_1 + \vec{\mathbf{c}}_2 \mod q$.
2. $\mathsf{Priv.HomMult}(\mathsf{pk}_{\mathsf{priv}}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: Runs $\mathsf{BFV.HomMult}$ 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 under $\mathsf{pk}_{\mathsf{priv}}$) to obtain the product $\vec{\mathbf{c}}_3$ after performing the appropriate rounding operation on $(c_{1,0} \cdot c_{2,0}, c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}, c_{1,1} \cdot c_{2,1})$. We call $\mathsf{Priv.Refresh}$ on $\vec{\mathbf{c}}_3$ and output the result.
3. $\mathsf{Priv.Refresh}(\vec{\mathbf{c}}, \tau)$: Runs $\mathsf{BFV.Refresh}$ on the ciphertext $\vec{\mathbf{c}}$ (encrypted under $\mathsf{pk}_{\mathsf{priv}}$) using auxiliary information τ associated with private account $\mathsf{pk}_{\mathsf{priv}}$ to facilitate key switching.

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 [66]) to represent integer value currency amounts for private accounts as follows:

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

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

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

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

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

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

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

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

- Receiver's public key: $\mathsf{pk}^{\mathsf{to}}_{\mathsf{pub}}$
- Transfer amount (in plaintext): amnt
- amnt encrypted under 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 this 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 π_{deshield} is produced using discrete log proofs. 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:

- Valid signature from sender
- Sender's account is not currently locked
- 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: $\mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{amnt}; \mathsf{r}) \overset{?}{=} \vec{\mathbf{c}}$
- Sender's remaining balance is correctly computed: $\vec{\mathbf{b}}' \overset{?}{=} \vec{\mathbf{b}} - \vec{\mathbf{c}}$
- Proof π_{deshield} is valid

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

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

- Receiver's public key: $\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}$
- Transfer amount encrypted under sender's public key: $\vec{\mathbf{c}} = \mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{amnt}; \mathsf{r})$
- Transfer amount encrypted under receiver's public key: $\vec{\mathbf{c}}' = \mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{amnt}; \mathsf{r})$
- Proof that $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}}'$ encrypt same transfer amount amnt with same randomness r and that this transfer amount is in $[0, \mathsf{MAX}]$ [17]
- 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: $\vec{\mathbf{b}}^* \overset{?}{=} \vec{\mathbf{b}} - \vec{\mathbf{c}}$
- All proofs are valid

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

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

This equation verifies that $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}}'$ do in fact encrypt the same amount amnt under the sender's public key $\mathbf{A}$ and the receiver's public key $\mathbf{B}$, respectively. Thus, we will need to show that $\vec{\mathbf{m}}$ satisfies the above equation and that amnt represented in it is non-negative. This can be done using [34] 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 in $\mathsf{tx}_{\mathsf{deshield}}$ as discussed earlier.

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 Instantiating Private Computation

The exact computation to be performed is based on the content of a smart contract that a user deploys. Requesting a computation to be executed (over public or private inputs) is done by sending transactions containing calls to the functions defined in such a smart contract. The code of a function is represented by a circuit $\mathcal{C}$ that $\mathsf{Compute}$ takes as input. It is up to the contract creator to select the computation and specify which conditions must be satisfied by the user-provided encrypted inputs x_{j}.

Based on the instantiation of smartFHE, computation is limited to what can be presented by the selected FHE scheme—binary or arithmetic circuits depending on the type of scheme chosen. The user provides her encrypted inputs (encrypted with respect to her FHE public key $\mathsf{pk}_{\mathsf{priv}}$) and the necessary ZKPs π_i showing that her inputs satisfy the relevant conditions. Similar to private payments discussed in the previous section, this is done via short discrete log proofs and bulletproofs in which we can re-use the Pedersen commitment created as part of short discrete log proofs for the Bulletproofs commitment.

[17] 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 [35].

If the proof is accepted, the requested computation is then performed directly on the encrypted inputs, and the ledger state will be updated accordingly. As discussed earlier in the Introduction, miners do not need to provide correctness proofs to verify that $\mathcal{C}$ has been performed correctly over the user inputs, i.e. ledger state updates are valid. This is enforced by the security assumption that the majority of the underlying mining power is honest. Thus, only state changes resulted from valid computation will be accepted.

Depending on the computation being performed (if a reciprocal action must happen on a different user's account), the user may also need to encrypt certain inputs with respect to the reciprocal account's public key and show that the two encryptions indeed contain the same plaintext value; this proof is similar in flavor to what was presented as a private transfer transaction (i.e. $\mathsf{tx}_{\mathsf{privtransf}}$).

5.4 Security

Our instantiation realizes a correct and secure PPSC scheme based on the notions introduced in Section 2. In Appendix C, we prove the following theorem:

Theorem 1. *Our smartFHE instantiation realizes a correct (cf. Definition 2) and secure (cf. Definition 5) PPSC scheme (cf. Definition 1).*

We note that smartFHE is applied on top of a (secure) public smart contract-enabled blockchain. We do not change the consensus, liveness, availability, or the public payment/smart contract operation of the underlying system; these invariants are preserved. Thus, proving security entails proving that smartFHE satisfies the additional privacy properties it offers (outlined in Definition 1), which is detailed in Appendix C.

6 Performance Evaluation

We evaluate the computational and storage cost of our instantiation, highlighting how our work supports lightweight users. Specifically, we provide the cost of private transactions in our system as well as three different representative applications. We compare our work to the state-of-the-art Veri-zexe [72], which builds upon and improves Zexe. In evaluating our system, we provide the *first* implementation of short discrete log proofs and open source our library. We describe our methodology, discuss the obtained results, and address potential extensions to further improve execution time and storage costs.

6.1 Methodology

For the BFV scheme [36], we use Microsoft's SEAL library [66] to prototype the various FHE computations needed as this library provides a highly optimized implementation. For Bulletproofs, we use Dalek [3] for 32-bit range proofs as it is one of the fastest implementations of Bulletproofs. For short discrete log proofs, we provide the first implementation (a library with 4102 LOC) with Apple Metal GPU-accelerated code; we significantly improve upon the authors' performance estimates in [34] by speeding up scalar multiplication and scalar inversion. In our library, we use Curve25519 for compatibility with the Dalek Bulletproofs library. We believe our library is of independent interest as this proof system provides reasonable time/space tradeoffs for proving well-formedness of FHE ciphertexts; thus, we have open-sourced our work for the community. For digital signatures, we use ECDSA with curve secp256k1 (as used by Ethereum) from OpenSSL. We conducted our experiments on an Apple M2 Max with 32GB RAM, chosen to represent a lightweight user.

We benchmark our work against the state-of-the-art Veri-zexe [72] which improves upon Zexe by removing per-application trusted setup and significantly cutting down on memory usage but at the cost of increasing transaction size by an order of magnitude. As Zexe was the most performant system implementing the ZKP-based approach, comparing our system against its successor Veri-zexe seems most appropriate.[18] Both Zexe and Veri-zexe work in the UTXO model so costs are given based on the number of inputs and outputs as per the methodology used in their own evaluations.

[18] Zexe's code (the one used in their paper [19]) is no longer maintained. For this reason, Veri-zexe chose to benchmark their work against snarkVM testnet-2 (created by the company Aleo [1]) which instantiates the same model as Zexe.

Table 1: Setup times (one time cost)

Performed by	**Operation**	$d = 1024$	$d = 2048$	$d = 4096$
User	KeyGen	0.216 ms	0.375 ms	36.5 ms
System	ZKP setup	0.8 s	2.06 s	5.7 s

6.2 Results

Setup costs for the system and per user are fairly negligible as observed in Table 1. We evaluate the cost of private transactions and three private smart contract applications ranging from DeFi to statistical computations. Before discussing the results, we note that one of the most important parameters to set is the polynomial modulus degree d for the BFV scheme as it has the largest impact on short discrete log proof performance, which dominates both transaction generation and verification time. Additionally, setting d smaller limits the number of sequential homomorphic multiplications that can be performed. Effectively choosing FHE scheme parameters is a difficult task with many tradeoffs to consider and beyond the scope of this work [69]. In deploying our system, we suggest $d = 1024$ if private transactions are the main focus. For deploying more complex smart contract applications, d should be set to 2048 or 4096.

Private transactions. We evaluate the cost of the main transactions in our system, focusing on the added cost from privacy. As shown in Table 2, shield and deshield transactions are fairly lightweight with regards to generation time and size.[19] The situation is different for private transfer due to the proofs incorporated. Miner cost is particularly high as our experiments were run on a laptop. While our system targets lightweight users, we emphasize that miners are *not* lightweight and require much more heavy duty machines to efficiently verify transactions.[20]

Veri-zexe's performance is given in Table 4. Working in the UTXO model presents a scalability issue when trying to spend coins distributed among several UTXOs. To elaborate, a user must prove that she owns all these (private) inputs, thus increasing the generation cost as the table shows[21]—a problem that does not exist in the account-based model we adopt. Our scheme exhibits superior performance in terms of user execution time even compared to 2x2 transactions using Veri-zexe. The difference is more stark when setting $d = 2048$ or lower.

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

	Operation	**Time (s)**	**Size (KB)**
$d = 1024$	$\mathsf{Shield}(\mathsf{tx}_{\mathsf{shield}})$	0.0002	0.101
	$\mathsf{Deshield}(\mathsf{tx}_{\mathsf{deshield}})$	1.89	2.47
	$\mathsf{PrivTransfer}(\mathsf{tx}_{\mathsf{privtransf}})$	3.57	20.03
$d = 2048$	$\mathsf{Shield}(\mathsf{tx}_{\mathsf{shield}})$	0.0002	0.101
	$\mathsf{Deshield}(\mathsf{tx}_{\mathsf{deshield}})$	3.58	2.53
	$\mathsf{PrivTransfer}(\mathsf{tx}_{\mathsf{privtransf}})$	10.7	64.76
$d = 4096$	$\mathsf{Shield}(\mathsf{tx}_{\mathsf{shield}})$	0.0002	0.101
	$\mathsf{Deshield}(\mathsf{tx}_{\mathsf{deshield}})$	11.17	2.66
	$\mathsf{PrivTransfer}(\mathsf{tx}_{\mathsf{privtransf}})$	23.89	180.1

Private smart contract applications. Next, we consider three applications offering I/O privacy implemented as private smart contracts. Briefly, they are (full details can be found in Appendix B):

A1. *Automated market maker (AMM):* a special form of decentralized exchange for trading cryptocurrency. Adding privacy to AMMs protects users against front running attacks. An AMM trades tokens in pairs, say token A and B, with reserve values denoted as $total_A$ and $total_B$. A user submits a order to swap an (encrypted) amount $amnt_A$ of token A. The AMM contract computes the

[19] As a further optimization, we are able to perform shield and deshield transactions using ciphertext-plaintext operations in the BFV scheme. Thus, the user does not need to encrypt the transfer amount.

[20] We benchmarked the SDLP verifier algorithm on an M2 Max with 30 GPU cores, each of which runs 32 threads in a wavefront. Since scalar multiplication is ALU bound, we expect a validator using e.g. an Nvidia RTX 4090 with 16384 shader cores to conservatively yield a 10-15x performance improvement.

[21] The slowdown is non-linear; interested readers can check [72] for detailed justification of this trend.

Table 3: Private transaction costs for smartFHE's instantiation—miner side.

	Operation	**Time (s)**
$d = 1024$	VerifyShield	0.00017
	VerifyDeshield	0.92
	VerifyPrivTransfer	1.95
$d = 2048$	VerifyShield	0.00017
	VerifyDeshield	1.92
	VerifyPrivTransfer	6.37
$d = 4096$	VerifyShield	0.00017
	VerifyDeshield	6.42
	VerifyPrivTransfer	14.77

Table 4: Base private transaction costs for Veri-zexe.

no. of inputs × no. of outputs	**User generation time (s)**	**Miner verification time (ms)**	**Size (KB)**
2 × 2	27.82	13.21	4.82
3 × 3	54.9	13.14	4.88
4 × 4	59	13.15	4.95
8 × 8	121	13.15	5.2

Table 5: Private smart contract application costs.

Application	**Per user generation time (s)**	**Miner verification/-computing time (s)**	**Size (KB) per user**
AMM ($d = 2048$)	6.4	3.58	33.53
AMM ($d = 4096$)	20.88	12.64	91.4
Mean/variance ($d = 4096$)	20.89	62.9	91.45
Chi-squared ($d = 4096$)	23.89	44.39	26.95

(encrypted) amount $amnt_B$ of token B this order will receive (using the constant product formula): $amnt_B = total_B - total_A \cdot total_B/(total_A + amnt_A)$

A2. *Financial standing of DAO members:* a DAO (decentralized autonomous organization) contract wants to decide if the financial standing of a set of members (5 in our experiment) permits spinning out a new business proposal. Members submit their (encrypted) fund amounts and the DAO computes the mean/variance of these funds.

A3. *Insurance premium:* a decentralized health insurance company relies on the chi-squared test over genetic data to decide premiums. The data is hosted by a group of hospitals (set to 3 hospitals holding data for 98 patients in our experiment). Hospitals submit the private quantities needed to compute the test. The contract aggregates the submissions and facilitates the test computation.

The first and second applications require the user to provide a single encryption, generate a short discrete log proof for two ciphertexts, and a bulletproof. The third application requires each party to provide three encryptions, a short discrete log proof, and a bulletproof. Miners check the proofs and then perform the homomorphic computation (the AMM calculation, mean/variance calculation, or chi-squared calculation, respectively). For these applications, we focus on the core computation related to smartFHE (that involve FHE, ZKP, and signatures) to quantify the overhead of our scheme. The full description of the applications (as found in Appendix B) include additional details that are implementation dependent, e.g. the users share the secret key of the FHE computation, decrypt the result locally (decryption time is 0.087 ms, 0.173 ms, 0.7 ms for $d = 1024, 2048, 4096$, respectively), or notify the smart contract that an encrypted result is within a given range or open the result (if the developer chooses to do that, the costs of these operations are basically that of our private transactions reported earlier).

The cost of our applications is shown in Table 5. Regardless of application, user execution time is superior to Veri-zexe on even two inputs and two outputs; furthermore, it is unclear if an AMM can be implemented in the UTXO model used by Veri-zexe or Zexe.

Scalability. We briefly discuss issues that impact scalability of the system. Increasing the number of users (or clients) does not impact the per-user cost since each of them is concerned with only her own inputs. The miners, on the other hand, have to process more information as the number of users increases. From our benchmarked applications, we observe that the additional cost is dominated by verifying more short discrete log proofs rather than the FHE computation or bulletproof verification. For example, for the mean/variance applications, miner verification time is 63.15 s, 126.19 s, and 189.23 s for 5, 10, and 15 users participating in that application, respectively (where the short discrete log proof verification cost in these scenarios is 62.9 s, 125.8 s, and 188.7 s, respectively).

As such, an important area to focus on is a GPU-accelerated implementation of this proof system for OpenCL so we can target a variety of architectures and provide significantly improved performance for miners (who certainly would not be using an Apple laptop). Performance could be further improved by speeding up multi-scalar multiplication. These are among our future work directions.

Storage is a large concern in the blockchain setting. To get around having large transactions and private smart contract state, a potential solution is to employ a decentralized file storage system like IPFS [5]. Retrieving cached content takes less than a second on average [4]. To prevent data from being discarded in IPFS, it must be "pinned." Storage is tamper resistant as any changes to the data changes the content identifier. Rather than storing the FHE ciphertexts directly on chain, a content identifier may be stored in its place which allows users or miners to then retrieve the relevant ciphertext.

7 Conclusion

In this paper, we defined a notion for a PPSC scheme and introduced smartFHE as a modular framework realizing this notion. smartFHE is the first system to investigate the utility of FHE in supporting private computing in the blockchain model and the first to support private smart contracts for lightweight users. In comparison to a state-of-the-art solution relying on the ZKP-based approach (where users do the computation off-chain and submit ZKPs attesting to correctness), our experiments show that our work offers superior performance for the user. Such results demonstrate the potential viability of FHE-based solutions to private decentralized applications.

References

1. Aleo. `https://www.aleo.org/`.
2. Curve amm. `https://curve.fi/#/ethereum/swap`.
3. Dalek library. `https://github.com/dalek-cryptography/bulletproofs`.
4. Gaining visibility in ipfs systems. `https://blog.cloudflare.com/ipfs-measurements/`.
5. Ipfs. `https://docs.ipfs.tech/concepts/what-is-ipfs/`.
6. Spiral. `https://btc.usespiral.com/`.
7. Uniswap protocol. `https://uniswap.org/`.
8. Ghada Almashaqbeh and Ravital Solomon. Sok: Privacy-preserving computing in the blockchain era. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 124–139. IEEE, 2022.
9. Thomas Attema, Vadim Lyubashevsky, and Gregor Seiler. Practical product proofs for lattice commitments. *IACR Cryptol. ePrint Arch.*, 2020:517, 2020.
10. Aritra Banerjee, Michael Clear, and Hitesh Tewari. zkhawk: Practical private smart contracts from mpc-based hawk. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 245–248. IEEE, 2021.
11. Aritra Banerjee and Hitesh Tewari. Multiverse of hawkness: A universally-composable mpc-based hawk variant. *Cryptography*, 6(3):39, 2022.
12. Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch Lafuente. Maximizing extractable value from automated market makers. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 3–19. Springer, 2022.
13. Carsten Baum, James Hsin-yu Chiang, Bernardo David, and Tore Kasper Frederiksen. Eagle: Efficient privacy preserving smart contracts. *Cryptology ePrint Archive*, 2022.
14. Alex Biryukov and Sergei Tikhomirov. Deanonymization and linkability of cryptocurrency transactions based on network analysis. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 172–184. IEEE, 2019.
15. Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter MR Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38*, pages 565–596. Springer, 2018.

16. Jonathan Bootle, Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Gregor Seiler. A non-pcp approach to succinct quantum-safe zero-knowledge. In *Annual International Cryptology Conference*, pages 441–469. Springer, 2020.
17. Jonathan Bootle, Vadim Lyubashevsky, and Gregor Seiler. Algebraic techniques for short (er) exact lattice-based zero-knowledge proofs. In *Annual International Cryptology Conference*, pages 176–202. Springer, 2019.
18. Jonathan Bootle, Vadim Lyubashevsky, and Gregor Seiler. Algebraic techniques for short (er) exact lattice-based zero-knowledge proofs. In *Annual International Cryptology Conference*, pages 176–202. Springer, 2019.
19. Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 820–837, 2018.
20. Zvika Brakerski. Fundamentals of fully homomorphic encryption-a survey. In *Electron. Colloquium Comput. Complex.*, volume 25, page 125, 2018.
21. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. `https://eprint.iacr.org/2011/277`.
22. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*, pages 423–443. Springer, 2020.
23. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.
24. David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. In *NDSS*, volume 14, pages 23–26. Citeseer, 2014.
25. David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.
26. Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.
27. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*, pages 409–437. Springer, 2017.
28. Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 738–768. Springer, Cham, 2020.
29. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.
30. Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 1–19. Springer, 2021.
31. Tarun Chitra, Guillermo Angeris, and Alex Evans. Differential privacy in constant function market makers. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 149–178. Springer, 2022.
32. Wei Dai. Pesca: A privacy-enhancing smart-contract architecture. *Cryptology ePrint Archive*, 2022.
33. Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd 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*, pages 142–156, 2019.
34. Rafael del Pino, Vadim Lyubashevsky, and Gregor Seiler. Short discrete log proofs for fhe and ring-lwe ciphertexts. In *IACR International Workshop on Public Key Cryptography*, pages 344–373. Springer, 2019.
35. Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM journal on computing*, 38(1):97–139, 2008.
36. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.
37. Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 649–678. Springer, 2019.
38. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

39. Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 844–855, 2014.
40. Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Submission to the NIST's post-quantum cryptography standardization process*, 2018.
41. Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.
42. Priscilla E Greenwood and Michael S Nikulin. *A guide to chi-squared testing*, volume 280. John Wiley & Sons, 1996.
43. Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.
44. Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *Annual International Cryptology Conference*, pages 581–612. Springer, 2017.
45. Roger A Hallman, Kim Laine, Wei Dai, Nicolas Gama, Alex J Malozemoff, Yuriy Polyakov, and Sergiu Carpov. Building applications with homomorphic encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2160–2162, 2018.
46. Muneeb Ul Hassan, Mubashir Husain Rehmani, and Jinjun Chen. Differential privacy in blockchain technology: A futuristic approach. *Journal of Parallel and Distributed Computing*, 145:50–74, 2020.
47. Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.
48. Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1353–1370, 2018.
49. Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Kachina–foundations of private smart contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16. IEEE, 2021.
50. Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.
51. Kristin Lauter, Adriana López-Alt, and Michael Naehrig. Private computation on encrypted genomic data. In *Progress in Cryptology-LATINCRYPT 2014: Third International Conference on Cryptology and Information Security in Latin America Florianópolis, Brazil, September 17–19, 2014 Revised Selected Papers*, pages 3–27. Springer, 2015.
52. Yehuda Lindell. Secure multiparty computation. *Communications of the ACM*, 64(1):86–96, 2020.
53. Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping. *Cryptology ePrint Archive*, 2021.
54. Zeyu Liu and Eran Tromer. Oblivious message retrieval. In *Annual International Cryptology Conference*, pages 753–783. Springer, 2022.
55. Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plancon. Lattice-based zero-knowledge proofs and applications: Shorter, simpler, and more general. *Cryptology ePrint Archive*, 2022.
56. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.
57. Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1568–1568. IEEE Computer Society, 2022.
58. Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *Proceedings on Privacy Enhancing Technologies*, 2021(CONF):291–311, 2021.
59. Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key fhe. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 735–763. Springer, 2016.
60. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, 2019.
61. Ngoc Khanh Nguyen and Gregor Seiler. Practical sublinear proofs for r1cs from lattices. In *Annual International Cryptology Conference*, pages 133–162. Springer, 2022.
62. Shen Noether. Ring signature confidential transactions for monero. *IACR Cryptol. ePrint Arch.*, 2015:1098, 2015.
63. Andreas Park. The conceptual flaws of constant product automated market making. *Available at SSRN 3805750*, 2021.
64. Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.

65. Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.
66. Microsoft SEAL (release 3.5). `https://github.com/Microsoft/SEAL`, April 2020. Microsoft Research, Redmond, WA.
67. Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1543–1543. IEEE Computer Society, 2022.
68. Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1759–1776, 2019.
69. Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1092–1108. IEEE, 2021.
70. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.
71. Aaron Wright. The rise of decentralized autonomous organizations: Opportunities and challenges. *Stanford Journal of Blockchain Law & Policy*, 4(2):152–176, 2021.
72. Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. Veri-zexe: Decentralized private computation with universal setup. *Cryptology ePrint Archive*, 2022.
73. Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols. *ACM Computing Surveys*, 55(11):1–50, 2023.

A Cryptographic Building Blocks

In this section, we provide an extended review of the cryptographic building blocks that we use in our instantiation of smartFHE (Section 5)—namely, fully homomorphic encryption, zero-knowledge proof systems, and digital signatures.

A.1 Fully Homomorphic Encryption

FHE supports computations directly on ciphertexts. All currently known schemes rely on lattice-based cryptography, thus providing post-quantum security guarantees. In our instantiation, we use the BFV scheme [36], which is based on the Ring-LWE public key encryption scheme [56]. We provide an overview of both schemes starting with the latter.

Ring-LWE Encryption Scheme. The Ring-LWE public key encryption scheme [56], which we denote as RPKE is composed of five algorithms: Setup, $\mathsf{SecretKeyGen}$, $\mathsf{PublicKeyGen}$, $\mathsf{Encrypt}$, and $\mathsf{Decrypt}$. All operations are 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. In describing the previous algorithms, we loosely follow the presentation from short discrete log proofs here [34]. Let λ be the security parameter, the RPKE is defined as follows:

$\mathsf{RPKE.Setup}(1^\lambda, 1^\mu)$: Takes as inputs security parameter λ and positive integer μ. It outputs public parameters $\mathsf{rpke.pp} = (p, q, d, \chi)$ 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].

$\mathsf{RPKE.SecretKeyGen}(\mathsf{rpke.pp})$: Takes the public parameters $\mathsf{rpke.pp}$ as input, and outputs a secret key $\mathsf{rpke.sk} = \mathbf{s}$ where $\mathbf{s}$ is a polynomial with small, bounded coefficients from the error distribution χ.

$\mathsf{RPKE.PublicKeyGen}(\mathsf{rpke.pp}, \mathsf{rpke.sk})$: Takes the public parameters $\mathsf{rpke.pp}$ and the secret key $\mathsf{rpke.sk}$ as inputs, and outputs public key $\mathsf{rpke.pk} = (\mathbf{a}, \mathbf{t})$ for $\mathbf{a}, \mathbf{t} \in R_q$ where $\mathbf{a}$ is a random polynomial and $\mathbf{t} = \mathbf{as} + \mathbf{e}$ such that $\mathbf{e}$ is a polynomial with small, bounded coefficients from the error distribution χ.

$\mathsf{RPKE.Encrypt}(\mathsf{rpke.pp}, \mathsf{rpke.pk}, \mathsf{m})$: Takes the public parameters $\mathsf{rpke.pp}$, and the public key $\mathsf{rpke.pk}$, and the message $\mathsf{m} = \mathbf{m} \in R_q$ to be encrypted as inputs (where all the coefficients of $\mathbf{m}$ are in $\mathbb{Z}_p$), and outputs the ciphertext $\vec{\mathbf{c}}$ that is computed as follows:

1. Sample polynomials $\mathbf{r}, \mathbf{e_1}, \mathbf{e_2}$ with small, bounded coefficients from the error distribution χ. Form $\vec{\mathbf{m}}^*$ consisting of the message and randomness as follows: $\vec{\mathbf{m}}^* = \begin{pmatrix} \mathbf{r} \\ \mathbf{e_1} \\ \mathbf{e_2} \\ \mathbf{m} \end{pmatrix}$
2. Form the matrix $\mathbf{A}$ from rpke.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 $\vec{\mathbf{c}} = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix}$

RPKE.Decrypt(rpke.pp, rpke.sk, $\vec{\mathbf{c}}$): Takes the public parameters rpke.pp, the public key rpke.pk, and the ciphertext $\vec{\mathbf{c}}$ as inputs, and outputs the plaintext $\mathbf{m}$ produced by decrypting $\vec{\mathbf{c}}$ as follows: 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 is based on the hardness of Ring-LWE for ring R [56]. 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. Full details on the reduction can be found in [56].

BFV Scheme. The BFV scheme [36] is a 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. As mentioned before, bootstrapping can be used as an optimization to avoid having to specify the multiplicative depth in advance.

When we multiply two ciphertexts $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}}'$ together, we get a long resulting ciphertext. Having to work with these increasingly long ciphertexts impacts the efficiency of the scheme so BFV utilizes an additional technique called *key switching* (or relinearization) that instead allows us to work with a smaller ciphertext that is of the same size as the original. This technique is encapsulated in the refreshing procedure that can be performed by anyone. The security of the BFV scheme follows from the security of the basic Ring-LWE encryption scheme [56].

We present a simplified description of the BFV scheme below, while full details can be found in [36]. The BVF scheme is composed of seven algorithms: Setup, KeyGen, Encrypt, Decrypt, HomAdd, HomMult, and refresh defined as follows:

BFV.Setup(1^λ): Takes the security parameter λ as input, and outputs the parameters bfv.pp, which includes a modulus, noise distribution, two integers, and rpke.pp (produced by invoking RPKE.SecretKeyGen(rpke.pp)).

BFV.KeyGen(bfv.pp): Takes the public parameters bfv.pp as input, and outputs a secret key sk = s obtained by running RPKE.SecretKeyGen(rpke.pp)), a public key pk (obtained by running RPKE.PublicKeyGen(rpke.pp, s)), and auxiliary information $\{\tau\}$ needed to facilitate the key switching procedure in BFV.Refresh.

BFV.Encrypt(bfv.pp, bfv.pk, m): Takes the public parameters bfv.pp, the public key bfv.pk, and a message m as inputs. It outputs a ciphertext $\vec{\mathbf{c}}$ which consists of two Ring-LWE samples. The first sample $\mathbf{c_0}$ encodes the message m using $\mathbf{t}$ from rpke.pk and the second sample $\mathbf{c_1}$ is auxiliary and formed from $\mathbf{a}$ in rpke.pk.

BFV.Decrypt(bgv.pp, bfv.sk, $\vec{\mathbf{c}}$): Takes the public parameters bfv.pp, the secret key bfv.sk, and a ciphertext $\vec{\mathbf{c}}$ as inputs. It outputs the corresponding plaintext m after performing the appropriate modulo reductions and rounding operations on $\mathbf{c_0} + \mathbf{c_1 s}$.

$\mathsf{BFV.HomAdd}(\mathsf{bfv.pk}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: Takes as inputs two ciphertexts $\vec{\mathbf{c}}_1 = (c_{1,0}, c_{1,1})$ and $\vec{\mathbf{c}}_2 = (c_{2,0}, c_{2,1})$, such that both are encrypted under the public key $\mathsf{bfv.pk}$. It outputs $\vec{\mathbf{c}}_3 = (c_{1,0} + c_{2,0}, c_{1,1} + c_{2,1})$, which is the sum of the two ciphertexts $\vec{\mathbf{c}}_1$ and $\vec{\mathbf{c}}_2$ resulted from performing component-wise vector addition.

$\mathsf{BFV.HomMult}(\mathsf{bfv.pk}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: Takes as inputs two ciphertexts $\vec{\mathbf{c}}_1 = (c_{1,0}, c_{1,1})$ and $\vec{\mathbf{c}}_2 = (c_{2,0}, c_{2,1})$, such that both are encrypted under the public key $\mathsf{bfv.pk}$. It computes $(c_{1,0} \cdot c_{2,0}, c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}, c_{1,1} \cdot c_{2,1})$ over the integers and then performs a rounding operation $\mod q$ to obtain $\vec{\mathbf{c}}_3$, which is the product of the two ciphertexts. Finally, it calls $\mathsf{BFV.Refresh}$ on $\vec{\mathbf{c}}_3$ and outputs the result.

$\mathsf{BFV.Refresh}(\vec{\mathbf{c}}, \tau)$: Takes as inputs a ciphertext $\vec{\mathbf{c}}$ and auxiliary information τ to facilitate key switching. It then expands the ciphertext and performs the key switching procedure resulting in a new ciphertext $\vec{\mathbf{c}}'$ of the original size, and outputs $\vec{\mathbf{c}}'$.

A.2 Zero-Knowledge Proofs

As FHE uses lattice-based cryptography, lattice-based ZKPs would be a natural candidate for proving relations about the input plaintexts in our instantiation. There have been recent improvements to lattice-based ZKPs (namely [18], [16], and [9]) 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 [34]. We take this approach in our instantiation to obtain small proof sizes (in the single digit kilobyte range). We will then use Bulletproofs [23] to prove properties of the plaintext (such as ensuring that a currency amount is in a particular value range). Both of these ZKP systems provide soundness, completeness, and zero-knowledge guarantees and can be made non-interactive using the Fiat-Shamir transform [38]. Additionally, neither requires a trusted setup.

Bulletproofs. This proof system [23] 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 [34], 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 [34].

Short Discrete Log Proofs. This proof system [34] 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 [34]. 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 [56].

A.3 Digital Signatures

As our system deals with public and private accounts, issuing public and private transactions and computation requests, a user needs at least two keypairs: one for a private account and one for a public account (assuming the user is interested in both modes of operation). As mentioned earlier, with public accounts, we use ECDSA [47] as in Ethereum. For private accounts, if post-quantum security for signatures is desired, we can use a lattice-based digital signature scheme compatible with our FHE scheme. In practice, we would like for such a scheme to be fairly efficient. We use the lattice-based Falcon signature scheme [40] in our instantiation, one of three finalists for NIST's post-quantum cryptography standardization competition. If post-quantum security is not a concern, we can instead use ECDSA for improved efficiency and compatibility with existing public smart contract-enabled blockchains. The only condition is that the user must use a different keypair for the private account than that of the public account to preserve privacy.

B Applications

In this section, we demonstrate how our smartFHE instantiation can be used to support real-world applications that require privacy. These applications are used for performance evaluation purposes of our framework as shown in Section 6. Note that the goal is to provide representative computations that require FHE to enable evaluating the overhead of our framework.

B.1 Automated Market Makers

Automated market makers, or AMMs, are a form of decentralized exchange for trading cryptocurrency tokens without intermediaries [73]. They connect buyers with sellers, where sellers build liquidity pools for trading particular tokens, e.g. Ether-NU trading. AMMs are a prime example of Decentralized Finance (DeFi) services. They build a platform for automated order matching and execution. The price of the traded token is automatically computed based on supply and demand. Naturally, if supply is low (i.e. liquidity in the pool is low) then the price becomes higher. An AMM is usually implemented as a smart contract to support automatic execution, with many popular systems such as Uniswap [7] and Curve [2] deployed in practice.

A major problem for AMMs is front-running attacks that allow manipulating the market to achieve significant profits. Malicious parties monitor the mempool for large pending orders and submit competing (buy) ones with higher fees, before the matching happens, to front-run unsuspecting user orders. As the competing order may cause significant price slippage, the victim's (buy) order may execute at a worse price than anticipated. The malicious party will then execute a second (sell) order after the victim's order, profiting off the price slippage caused by the prior trades. The root cause of this type of attack is lack of privacy. If AMMs permitted execution of private orders, then malicious parties could not manipulate the pool price in such a manner. This is a critical problem that has received widespread interest [12, 63].

In this application, we show how smartFHE can be used to build a private AMM. We follow the constant product formula for computing the trading price as adopted in Uniswap. This formula keeps the ratio of token reserves, and consequently prices, in the pool as balanced as possible to reduce price slippage. In particular, let token A and token B be the pair of tokens being traded, and $total_A$ and $total_B$ be the pool reserve values of tokens A and B, respectively. This formula ensures that the price of token A multiplied by the price of token B equals a constant number, where the price of token A is $total_B/total_A$ (i.e. it is the number of B tokens required to purchase a single A token); a similar formula applies to the price of token B.

Accordingly, for an order trading an amount of token A, $amnt_A$, the amount of token B, $amnt_B$, that this order receives is computed as:

$$amnt_B = total_B - total_A \cdot total_B/(total_A + amnt_A)$$

A private version of this AMM can be implemented in smartFHE as follows. Alice locks her private account that contains tokens of type A to the AMM smart contract (to avoid any concurrency issues). She then issues an order, which is basically a transaction, to the AMM representing the intended trade. This transaction includes the encrypted amount of token A to be traded, denoted as c_A, and a ZKP attesting that:[22]

- c_A encrypts a non-negative value,
- that Alice's private account can cover the total order value,

[22] It is unlikely that a single trade will drain the pool reserves entirely of token B. Pool reserves are generally quite large and Alice will be required to show that she owns a sufficiently large amount of token A along with locking her account to the contract. Even if this is the case, submitting such an order is to Alice's detriment.

- and that c_A is well-formed.

The AMM smart contract will compute the amount of B tokens Alice will get for her order. This is done using the constant product formula as follows (where c_B is the ciphertext of B token amount resulting from the trade, encrypted under Alice's public key):

$$c_B = total_B - \frac{total_A \cdot total_B}{total_A + c_A}$$

Alice will then finalize the trade by issuing a deshield transaction to send $amnt_A$ tokens out of her private account to the AMM contract account, which the AMM adds to the pool reserve of token A. This transaction will include as auxiliary information the randomness used to encrypt c_A, allowing anyone to verify that indeed c_A encrypts the correct $amnt_A$ value. Upon receiving this transaction and executing it, the AMM contract issues a transfer transaction to send $amnt_B$ tokens to Alice's account (if Alice's B account is private, this will be a shield transaction), and modifies the pool reserves of token B accordingly. Note that even though these values are public, this does not impose any threat to Alice's trade since it has already been executed—thus addressing front-running attacks.

B.2 Statistical Data Analysis

Many financial applications rely on the results of particular statistical analyses of user financial standing or other user-related private information, e.g. health records. The spread of blockchain-based applications to avoid centralized trust has led to several innovations through which traditional services and organizations are reshaped into fully decentralized ones. Decentralized autonomous organizations (DAOs) [71] are one such example. These facilitate company/organization formation, with a transparent way of buying shares, voting on project and business proposals, decision making, e.g. for loan services or health insurance. We discuss two sub-applications under this category—namely, analyzing user financial standing and performing a chi-squared test over genetic data.

B.2.1 Analyzing Financial Standing For this application, we imagine that a DAO is trying to decide on forming a new company or spinning out a project based on the financial standing of the users backing the endeavour. Suppose we have a set of users $P_1, \ldots, P_n$, each of which is willing to put funds $u_1, \ldots, u_n$ to back the proposal, where the fund amounts are private. The goal is to compute the mean and variance to assist in making the decision. Specifically, the mean must be close enough to some target value set by the DAO and the individual funds should not be far from the mean.

This application relies on inputs from multiple users; we want all these to be encrypted under the same public key to allow the DAO contract to compute the mean and variance over the ciphertexts. One approach is to let the users collectively generate a public key pk and share the corresponding secret key sk. Thus, to decrypt any ciphertext under pk, all the users' secret key shares are needed. Each P_i will encrypt her fund value u_i (under pk) and send it to the DAO contract, which in turn computes the mean and the variance over all ciphertexts and updates the contract state with the (encrypted) result. The users can collectively decrypt the results and take further action (if needed). For example, if the statistical analysis values are within the required bounds stated by the DAO, any of these users can issue a transaction to inform the DAO of that (without disclosing the plaintext values of the statistical results).

In terms of smartFHE operations, the above can be done as follows. User P_i locks her own private account (which contains a balance that can cover the individual fund u_i) to the DAO smart contract. Then she submits a ciphertext c_i of the value of her fund u_i (encrypted under the group public key pk) to the DAO. This is done by issuing a transaction, which is basically a function call with the encrypted fund as an input. This call also includes a ZKP attesting that:

- The balance of the private account of P_i can cover the fund value u_i,
- and that the ciphertext of u_i is well-formed.

After receiving $c_1, \ldots, c_n$, the DAO contract will perform the computation. It computes the mean $\bar{c} = \frac{\sum_{i=1}^{n} c_i}{n}$, and the variance $var = \frac{\sum_{i=1}^{n}(c_i - \bar{c})^2}{n-1}$.[23]

[23] Note that the squaring operation is simply a homomorphic multiplication of the quantity by itself and the number of users is public—it is the count of the submitted ciphertexts. Thus, dividing the resulting ciphertext by a public constant is a cheap operation.

The resulting (encrypted) mean and variance will be recorded in the DAO smart contract. The users can retrieve these values and use their shares of the secret key to collectively decrypt the result. If they find out that these values satisfy the conditions set by the DAO, anyone can notify the DAO of that. This is can be done by issuing a transaction with a ZKP attesting that the encrypted mean and variance are within the prescribed bounds.

B.2.2 Chi-squared Test The chi-squared test, or $\mathcal{X}^2$, is a popular statistical test that answers questions with respect to association relations or goodness of fit [42]. In association relations, this involves testing whether a particular factor, such as age or gender, has a significant association with a particular outcome, such as election results or health status. In goodness of fit, this involves testing whether a particular variable (or set of variables) follows a particular distribution. In Genome-Wide Association Studies (GWAS), the chi-squared test is used to test for deviation from the Hardy-Weinberg equilibrium, and, thus, detect potential risks for diseases. Testing such hypotheses is also useful for financial applications, such as setting health insurance premiums.

In this application, we consider a scenario in which a (decentralized) health insurance company relies on the genetic testing results for a population of users of size n to make decisions for insurance premiums. The genetic data of these users is hosted by several trusted medical institution or hospitals. The population is divided into sub-populations of size $n_1, n_2, \ldots, n_q$ such that for $i \in \{1, \ldots, q\}$ the n_i group data is hosted by hospital h_i. All these hospitals want to keep the hosted data private from others.

We use the approach proposed in [51, 69] to compute this genetic test. We start with one group to explain the computation and then show how hospitals $h_1, \ldots, h_q$ will participate together under smartFHE. For population n_1, let the genetic counts be t_0, t_1, t_2 such that $n_1 = t_0 + t_1 + t_2$. For example, assume that the test is for the two alleles A and a of the gene, then $t_0 = t_{AA}$, $t_1 = t_{Aa}$, and $t_2 = t_{aa}$. The chi-squared test is computed in two stages. First, compute the quantities $\alpha = (4t_0t_2 - t_1^2)^2$, $\beta_1 = 2(2t_0 + t_1)^2$, $\beta_2 = (2t_0 + t_1)(2t_2 + t_1)$, and $\beta_3 = 2(2t_2 + t_1)^2$. Second, compute the test value as:

$$\mathcal{X}^2 = \frac{\alpha}{2n_1}\Big(\frac{1}{\beta_1} + \frac{1}{\beta_2}\frac{1}{\beta_3}\Big)$$

The goal is to have all hospitals collectively compute one test value over their data through the insurance company smart contract. Similar to the previous application, these hospitals will generate one public key pk and share the corresponding sk. Then, each hospital h_i will submit the encrypted genetic counts for population n_i, denoted as $c_{t_{i,0}}, c_{t_{i,1}}, c_{t_{i,2}}$ such that $n_i = t_{i,0} + t_{i,1} + t_{i,2}$. Hospital h_i provides a ZKP attesting that the counts are for the given population size n_i (which is public). That is, this ZKP attests that:

- All $t_{i,0}, t_{i,1}, t_{i,2}$ are non-negative,
- that $n_i = t_{i,0} + t_{i,1} + t_{i,2}$,
- and that all submitted ciphertexts are well-formed.

The contract first checks that all ZKPs are valid, and then computes the total encrypted counts:

$$c_{t_0} = \sum_{i=1}^{q} c_{t_{i,0}}\ ,\ c_{t_1} = \sum_{i=1}^{q} c_{t_{i,1}}\ ,\ \text{and}\ c_{t_2} = \sum_{i=1}^{q} c_{t_{i,2}}$$

After that, the contract computes four quantities:

$$\begin{aligned} c_\alpha &= (4c_{t_0}c_{t_2} - c_{t_1}^2)^2 \\ c_{\beta_1} &= 2(2c_{t_0} + c_{t_1})^2 \\ c_{\beta_2} &= (2c_{t_0} + c_{t_1})(2c_{t_2} + c_{t_1}) \\ c_{\beta_3} &= 2(2c_{t_2} + c_{t_1})^2 \end{aligned}$$

The hospitals can retrieve these encrypted quantities from the contract, collectively decrypt them using their shares of sk, and then any hospital can compute the test result $\mathcal{X}^2$ using the formula above.

Any of these hospitals can notify the insurance company smart contract of the result, which is done by issuing a transaction with a ZKP attesting that the test result:

- is correctly computed using the quantities computed by the contract,
- and that this result is in a given premium threshold range set in the contract.

This is done without disclosing the test value.

C Proof of Theorem 1

To prove Theorem 1, we need to prove that no PPT adversary can win the correctness and security games defined in Section 2 with non-negligible probability.

Intuitively, our smartFHE instantiation satisfies these properties by relying on the correctness and security of the underlying cryptographic primitives it uses. The use of a secure zero-knowledge proof guarantees: completeness (a valid honest proof generated by a user will be accepted by the miners), soundness (a user that does not own valid private inputs cannot forge valid proofs for any transaction or computation request), and zero-knowledge (so the proof does not reveal anything about the private inputs and private account state). The use of a semantically secure FHE scheme guarantees that the ciphertexts of the inputs and balances do not reveal anything about the underlying (plaintext) values, and operating on them (by homomorphically adding or multiplying them) will produce valid results. Also, the correctness of the locking process will avoid invalidating any pending ZKPs, and the correctness of the rolling over process at the end of each epoch will gurantee that account balances are updated correctly. The security of the digital signature scheme guarantees that a malicious adversary cannot forge signatures, and thus steal others' currency, and that a man-in-the-middle attacker cannot manipulate any of the messages sent in the system.

Formally, the proof of Theorem 1 requires proving three lemmas showing that smartFHE is correct and supports both overdraft safety and ledger indistinguishability.

Lemma 1. *The smartFHE instantiation described in Section 5 satisfies the correctness property (cf. Definition 2).*

Proof. Correctness follows by the correctness of the FHE scheme, the security of the ZKP system and the digital signature scheme, and the correctness of the locking and roll-over processes. Every operation, whether a valid transaction or a circuit computation, will be processed successfully in our instantiation and leads to a verifiable ledger update. This can easily be seen for each transaction type in the system. By relying on the completeness of the ZKP systems for Bulletproofs [23] and short discrete log proofs [34], the correctness of the BFV fully homomorphic encryption scheme [36], 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 easily seen that valid transactions will update the ledger state as expected.

That is, an adversary, who does not own an account or does not know the actual private inputs and data, cannot forge valid ZKPs; thus, any operation request he issues will not be accepted. Also, this adversary cannot forge a signature on behalf of an honest user or manipulate the operation requests issued by others in the system. For computations on private inputs, the correctness of the results is based on the correctness of the BFV scheme. That is, operating on the ciphertexts of the inputs will produce ciphertexts of the correct results (the same as what would be obtained if these inputs were public and the computation was performed in the clear). Being able to do any of these means violating the security of the underlying cryptographic primitives, which is a contradiction.

Furthermore, the result correctness of any FHE-based is guaranteed by the assumption that the underlying smart contract-enabled blockchain system is secure (so its consensus protocol satisfies the security properties of consensus—liveness, persistence/consistency, and chain quality [64]). In other words, miners will accept blocks that contain only valid results computed by correctly performing a $\mathsf{Compute}$ request using the supplemented user inputs, according to the circuit representing the function call in a given smart contract. Same for transactions issued by the users. This guarantees that the updated ledger state contains only valid state changes even for those over private inputs and private accounts.

Accordingly, in the $\mathsf{INCGame}$ 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 that may lead to invalid updates. Thus, $\mathcal{A}$ will have a negligible probability to succeed in producing an operation transcript that leads to different account balances, which completes the proof.

Lemma 2. *The smartFHE instantiation described in Section 5 satisfies the overdraft safety property (cf. Definition 6).*

Proof. To prove that our instantiation supports overdraft safety, we must consider $\mathsf{Transfer}$, Shield, $\mathsf{Deshield}$, and $\mathsf{PrivTransfer}$, and show that none of these transaction algorithms can be used to send more currency than a user rightfully owns with non-negligible probability. Recall that our instantiation satisfies correctness (as shown in Lemma 1) 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 the transactions listed above.

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

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

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

In $\mathsf{PrivTransfer}$, the state of the sender and receiver's accounts are private. As part of $\mathsf{VerifyPrivTransfer}$, ZKPs will be checked showing that the sender has enough currency in his account to perform the transaction and that the transfer amount encrypted under the sender and receiver's public key is identical and non-negative. Thus, if the sender is able to send more currency than he rightfully owns (i.e. $\mathsf{VerifyPrivTransfer}(\mathsf{tx}_{\mathsf{privtransf}}) = 1$), he has violated the soundness of the ZKP systems of Bulletproofs or short discrete log proofs (which happens with at most negligible probability).

As in the proof of correctness, by the assumption that the underlying blockchain system is secure, the miners will reject any invalid operations and transactions. Thus, any transaction in which its corresponding verify operation outputs 0 will be rejected, so it will not impact the updated ledger state.

Accordingly, in the OSGame game, the adversary $\mathcal{A}$ will have a negligible probability to succeed in spending more currency than what it owns. That is, regardless of how $\mathcal{A}$ interacts with the PPSC scheme (by instructing honest parties to place transactions and computation requests, or by inserting its own via its accounts or those that it corrupted), the probability of breaking overdraft safety is negligible, which completes the proof.

Lemma 3. *The smartFHE instantiation described in Section 5 satisfies the ledger indistinguishability property (cf. Definition 8).*

Proof. Recall that in the $\mathsf{LINDGame}$, starting with some ledger state, $\mathcal{A}$ interacts with the PPSC scheme (through $\mathcal{O}_{\mathsf{PPSC}}$) and can initiate any of the query types this oracle provides. $\mathcal{A}$ can request creating (public and private) accounts, transfer currency between them, request any (public or private) computation execution in any smart contracts it wishes, instruct honest parties to initiate transactions and computation requests, and can corrupt any of these parties and control it. At the end, $\mathcal{A}$ chooses two operations (transactions or computation requests) among which the challenger $\mathcal{C}$ will choose one at random and request $\mathcal{O}_{\mathsf{PPSC}}$ to execute it. Based on the updated ledger state, $\mathcal{A}$ will guess which operation was chosen and will win if the guess is correct. To rule out trivial wins of $\mathcal{A}$, the two operations op_0 and op_1 must be publicly consistent in the sense of Definition 7.

The proof must show that for all operation types in the system (including all transaction types and private computation requests), $\mathcal{A}$ will not be able to win the $\mathsf{LINDGame}$ with non-negligible probability. Without loss of generality, we show that for private computation requests; a similar proof and reasoning arguments can be applied to the rest of the operation types.

Using a proof technique inspired by the one in [37], we prove ledger indistinguishability for private computation operations using a series of hybrids starting with an $\mathsf{LINDGame}$ with $b = 0$ (Hybrid_0), and finishing with an $\mathsf{LINDGame}$ game with $b = 1$ (Hybrid_7). By showing that all these hybrids are indistinguishable, we prove that $\mathcal{A}$ cannot tell which operation $\mathcal{C}$ has chosen for execution.

A private computation operation is composed of a public computation circuit (or code) to be executed, a set of private inputs that are encrypted using the BFV FHE scheme, and ZKPs (using the

Bulletproofs and the short discrete log proofs system) to attest for the well-formedness of the input ciphertexts and that these inputs respect all conditions specified in the system. Based on that, we show the following series of hybrids:

$\underline{\mathsf{Hybrid}_0}$: The game $\mathsf{LINDGame}$ with $b = 0$.

$\underline{\mathsf{Hybrid}_1}$: Same as Hybrid_0, but we replace the zero-knowledge proofs with simulated ones, i.e., we invoke the zero-knowledge property simulator for each of the input values, and we replace the actual proofs in aux_0 and aux_1 with simulated ones.

The hybrids Hybrid_0 and Hybrid_1 are indistinguishable by the zero-knowledge property of the ZKP systems we use. If $\mathcal{A}$ can distinguish these two hybrids, then we can use $\mathcal{A}$ to build an adversary $\mathcal{B}$ that can break the zero-knowledge property of the underlying ZKP systems, which is a contradiction.

$\underline{\mathsf{Hybrid}_2}$: Same as Hybrid_1, but we replace the input ciphertexts in aux_0 and aux_1 with fresh ciphertexts (for plaintext inputs that may have different values than the original ones, but produce same circuit behavior but not necessarily same output value). That is, we choose a fresh FHE keypair and fresh input values, then we encrypt these values using the fresh secret key and use them to replace the original input ciphertexts in aux_0 and aux_1.

The hybrids Hybrid_1 and Hybrid_2 are indistinguishable by the zero-knowledge property of the ZKP system and the semantic security of the FHE scheme. These imply that, regardless of the actual plaintext input values, the input ciphertexts are indistinguishable—neither the ZKPs nor the FHE ciphertexts will reveal anything about the plaintext values. In other words, if $\mathcal{A}$ can distinguish these two hybrids, then we can use it to build two adversaries: $\mathcal{B}_1$ that can break the zero-knowledge property of the underlying ZKP systems, and $\mathcal{B}_2$ that can break the semantic security of the FHE scheme, which is a contradiction.

$\underline{\mathsf{Hybrid}_3}$: Same as Hybrid_2, but we replace the output of the execute query with fresh output produced by executing op_b using the fresh inputs generated in Hybrid_2.

The hybrids Hybrid_2 and Hybrid_3 are indistinguishable by the semantic security of the FHE scheme that our smartFHE instantiation uses. Similar to above, if $\mathcal{A}$ can distinguish them, then we can use it to build $\mathcal{B}$ that can break the semantic security of the FHE scheme, which is a contradiction.

$\underline{\mathsf{Hybrid}_4}$: Same as Hybrid_3, but with $b = 1$. The hybrids Hybrid_3 and Hybrid_4 are indistinguishable by the same argument described above. Since the computation is identical for both op_0 and op_1, by the semantic security of the FHE scheme and the zero-knowledge property of the ZKP scheme, and regardless of the actual input values, $\mathcal{A}$ will not be able to deduce anything about these inputs or the computed output. Thus, $\mathcal{A}$ will not be able to tell that op_1 has been executed instead of op_0 in this hybrid. Otherwise, we can use $\mathcal{A}$ to break the security of the FHE and ZKP schemes that our instantiation uses (as described above), which is a contradiction.

$\underline{\mathsf{Hybrid}_5}$: Same as Hybrid_4, but with aux_0 and aux_1 used in the original game (i.e. original inputs ciphertexts). So this is Hybrid_3 with $b = 1$. The hybrids Hybrid_4 and Hybrid_5 are indistinguishable by the indistinguishability argument of Hybrid_3 and Hybrid_2.

$\underline{\mathsf{Hybrid}_6}$: Same as Hybrid_5, but with the original output (i.e. ledger sate updates) that will be produced by op_b using the original aux_0 and aux_1 from the original game. So this is Hybrid_2 with $b = 1$. The hybrids Hybrid_5 and Hybrid_6 are indistinguishable by the indistinguishability argument of Hybrid_2 and Hybrid_1.

$\underline{\mathsf{Hybrid}_7}$: Same as Hybrid_6, but with the real (original) ZKPs used in the original game instead of the simulated ones. So this is the original $\mathsf{LINDGame}$ with $b = 1$. The hybrids Hybrid_6 and Hybrid_7 are indistinguishable by the indistinguishability argument of Hybrid_1 and Hybrid_0.

This sequence shows that $\mathsf{LINDGame}$ with $b = 0$ is (negligibly) indistinguishable from $\mathsf{LINDGame}$ with $b = 1$. This means that publicly-consistent private computation requests in our smartFHE instantiation are indistinguishable and will not give $\mathcal{A}$ any additional information that allows it to win $\mathsf{LINDGame}$ with non-negligible advantage.

Similar hybrid sequence can be developed for other operation types, and by relying on the security of the ZKP systems and the FHE scheme we use, similar reasoning as above can be used to show that they do not give $\mathcal{A}$ any non-negligible advantage in winning $\mathsf{LINDGame}$, which completes the proof.

Proof of Theorem 1. Follows by Lemmas 1, 2, and 3.