smartFHE: Privacy-Preserving Smart Contracts from Fully Homomorphic Encryption

Ravital Solomon[1] and Ghada Almashaqbeh[2]

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

Abstract. Smart contract-enabled blockchains represent a powerful tool in supporting a large variety of applications. Despite their salient features of transparency, decentralization, and expressiveness, building privacy-preserving applications using these platforms remains an open question. Existing solutions fall short in achieving this goal since they support a limited operation set, only support private computation on inputs belonging to one user, or even ask the users themselves to perform the computations off-chain.

In this paper, we propose *smartFHE*, a modular framework to support private smart contracts that utilizes fully homomorphic encryption (FHE). The smartFHE framework allows users to build arbitrary decentralized applications that preserve input/output privacy for inputs belonging to the same user or even different users. This is achieved by employing single and multi-key FHE to compute over private (encrypted) data and account balances, along with efficient zero-knowledge proof systems to prove well-formedness of private transactions. Crucially, our framework is "modular" since any FHE and ZKP scheme can be used so long as they satisfy certain minimal requirements with respect to correctness and security. Furthermore, smartFHE reduces the burden on the user, since miners translate smart contract code into public or private operations based on whether the accounts involved are public or private.

In proposing smartFHE, we define notions for a privacy-preserving smart contract (PPSC) scheme along with its correctness and security. We provide a concrete instantiation of a PPSC using the smartFHE framework. Finally, we consider further extensions/optimizations.

1 Introduction

Cryptocurrency can be traced back to (at least) 1983 when Chaum first proposed the concept of *electronic cash* using blind signatures [1]. Extending Chaum's design, Bitcoin [2] was introduced a few decades later and removed the need for a trusted party. It was an appealing concept: a way to exchange currency in a cryptographically secure way without relying on banks. Since then, hundreds of cryptocurrencies have been deployed with a total market cap exceeding $940 billion [3].

Cryptocurrency and Privacy. In Bitcoin [2], Nakamoto introduced the notion of a public distributed ledger called blockchain through which users could

exchange currency directly with one another. Although users have addresses that serve as their pseudonymous identities in the system, anyone can trace transactions on the blockchain to see exactly how much was exchanged and by which addresses. There have even been successful attempts to link these addresses with real-world identities [4].

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

Smart Contracts and Privacy. In parallel to the development of a private cryptocurrency, a very different question about Bitcoin's functionality was asked. Could Bitcoin be extended to support arbitrary user-defined applications? The answer was *yes* but with major changes to its UTXO-based design. Thus, Ethereum was born, defining an account-based model and a Turing-complete scripting language that could support arbitrary user-defined programs called smart contracts [7]. Using smart contracts, individuals can build applications processing highly sensitive data such as auctions and voting. Although Ethereum offers a highly expressive functionality, it provides no privacy out of the box.

Over the last few years, several attempts have been made to bring privacy to smart contracts. However, supporting arbitrary computation with privacy even for a *single* user's inputs and outputs has proved to be quite the challenge. Some constructions (such as Zether [8]) built upon the theoretical approach used for private currency transfer, operating directly on encryptions/commitments. Specifically, Zether exploits additive homomorphisms in the ElGamal encryption scheme and then uses ZKPs to prove that certain relations hold [8]. However, additive homomorphisms can support only a limited set of applications with input/ouput (I/O) privacy. Other constructions (such as Zexe [9] and Zkay [10]) abandoned the aforementioned approach since it did not yield immediately practical results in the short-term. Instead, their schemes offload *all* work to the user to do offline. Users perform all the computations themselves on plaintext data, encrypt the inputs and outputs of the computation, and create a ZKP certifying correctness of computation with respect to these encryptions. The blockchain miner's only role here is to verify correctness of the ZKP. We refer to this approach as the "pure ZKP approach" as it relies on the power of ZKPs to perform computations with I/O privacy.

1.1 Our Contributions

Operating directly on encrypted values has proven invaluable across numerous applications—from searching over encrypted databases [11], to protecting user data in machine learning applications involving neural networks [12]. Prominent private smart contract schemes may have chosen to abandon this approach as

it did not yield practical results in the short-term; we believe such a viewpoint may also be short-sighted.

Supporting additive and multiplicative homomorphisms on ciphertexts leads us to the holy grail of fully homomorphic encryption (FHE) which provides I/O privacy for any computation. Using FHE, users could supply encrypted inputs along with a simple ZKP showing well-formedness of the initial ciphertexts and that certain relations on the plaintexts are satisfied. Miners check the proof and then perform the requested computations directly on the encrypted inputs. No need for the user to remain online during the computation or provide complex ZKPs attesting to correctness of computation. Unlike the pure ZKP approach, here the private computation is performed on-chain.

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

Most importantly, the pure ZKP approach cannot scale to support privacy-preserving computation on multi-user inputs without utilizing (usually) highly-interactive MPC protocols.[3] Regardless of the particular MPC protocol chosen, users would still be responsible for coordinating the entire computation off-chain themselves and determining who would perform it. Our single-key FHE approach, like the pure ZKP approach, can only directly support computations with I/O privacy on inputs belonging to the same user. However, we have an ace up our sleeve to address the multi-user problem.

We follow the same blueprint as in single-key FHE to create smart contracts with I/O privacy for multi-user inputs. Users encrypt their own inputs using a multi-key FHE scheme and provide simple ZKPs showing well-formedness of each of their own ciphertexts. Miners can then perform arbitrary computations directly on any subset of the users' encrypted inputs. No off-chain coordination needed for the computation. No interaction necessary for the computation. No need for the users to even be online.

In an ambitious forward-looking world, one may entertain the thought of harnessing the power of multi-key FHE to realize these advantages—advantages that cannot be realized using the pure ZKP approach or its MPC-based extensions. Although multi-key FHE is far from being ready for practical deployment, current state-of-the-art schemes provides highly appealing theoretical

[3] Several works have dealt with MPC in a non-interactive setting where an output-producing party is available all the time; users submit their inputs and then this party computes the intended functionality output. However, these works also leak the residual function [14] and require additional setup assumptions such as pre-dealt correlated randomness [15], or rely on the existence of a PKI [14] or even indistinguishability obfuscation [16].

results—namely, non-interactivity for the computation itself and a one round decryption process [17]—that we can utilize in bringing this dream into reality.

We take a foundational approach to realizing smart contracts with I/O privacy by proposing a smart contract framework harnessing the power of single and multi-key FHE. Crucially, we design our framework to be *modular*, allowing the instantiator to choose the specific FHE and ZKP schemes assuming they satisfy certain minimal conditions.

New Notion for Privacy-Preserving Smart Contracts. We define a notion for privacy-preserving smart contracts (PPSCs) that captures the support of arbitrary computation with I/O privacy for inputs belonging to one or multiple users. Furthermore, we extend previous definitions of correctness and security (in terms of privacy/ledger indistinguishability and overdraft safety/balance) from Zether [8] and Zerocash [5] to provide formal guarantees for a PPSC scheme. We believe our PPSC definition is of independent interest as it is general enough to be used in future private smart contract constructions.

smartFHE: An FHE-based PPSC Framework. We propose a modular framework to support PPSCs using FHE and ZKPs. To the best of our knowledge, we are the first to use FHE in the smart contract (or even blockchain) setting.

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

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

Operating directly on private (i.e. encrypted) accounts requires us to address resulting concurrency issues. Say, for example, Alice has submitted her own private transaction to be processed and (while waiting) Bob successfully sends her currency. Alice's encrypted balance has changed and, thus, the ZKP in her transaction is no longer valid which would cause Alice's own transaction to be rejected. Worse, she would lose the fees associated with this transaction. We resolve such conflicts using a locking mechanism reminiscent of a mutex.

Finally, we show how the smartFHE framework supports a correct and secure PPSC scheme. Our framework will additionally protect against front-running and replay attacks.

A Harmonious Union. From our work, we observe that FHE and blockchain are well-suited to one another, particularly in theoretical terms. Blockchain allows FHE to address the pain point of verifying correctness of homomorphic computation. Using a blockchain also allows users to offload their computations to a party who is financially incentivized to always be online.

A Solution to Verifying Homomorphic Computations. When using FHE, the party performing the homomorphic computations (i.e. the evaluation party) is usually *different* from the private key owner. However, there is no immediate way for this key owner to verify that the evaluation party performed the homomorphic computations correctly.

There have been solutions to the challenge of verifiable computation, but adding verification to even simple homomorphic computations of multiplicative depth 2 can double the cost for the key owner [19]. By using a blockchain, we can solve the first problem in a simpler way through consensus. That is, the underlying security assumption of blockchain—namely, that the majority of the mining power is honest—provides guarantees with regards to correctness [20]. Miners re-execute the computation and only accept blocks that agree with what they computed. Thus, the owner of the encrypted data can rest assured that the evaluation party (in this case, the miners) performed the homomorphic computation correctly. No need for additional tools to verify correctness of computation. This also holds true when we use multi-key FHE in our framework.

An Always-available and Financially-incentivized Evaluation Party. To successfully outsource computation (as in the FHE setting), we need an evaluation party who is readily available; blockchain provides the perfect setting for this. In certain account-based models, such as Ethereum [7], miners are financially rewarded for being online and performing computations for the users. Thus, blockchain can provide an always-available and financially motivated evaluation party to perform private computations for users.

Additionally, FHE can be computationally intensive and require specialized hardware. This is not an issue in proof-of-work blockchains since the miners are often expected to have powerful and potentially even specialized machines available. No need for the users to be online during the computation; they can simply check the blockchain later for the result.

An Instantiation. We propose an instantiation of a PPSC scheme based on the smartFHE framework. We start with a simpler construction using single-key FHE; we then show how to extend it to support multi-key FHE. In particular, we employ current state-of-the-art FHE and ZKP schemes—specifically, the lattice-based BGV (fully homomorphic encryption) scheme [21] and the discrete log

proofs construction [18] which allows for efficient proofs of lattice-based relations using Pedersen commitments.

Our single-key FHE-based instantiation is trustless—requiring no trusted third party or trusted setup. Additionally, it supports arbitrary user-defined smart contracts with I/O privacy for a single user's inputs (like previous constructions Zexe [9], Zkay [10]) along with public and private payments. We demonstrate in Appendix B how our instantiation can be used to support some popular applications that operate on inputs from different users with additional logic in the smart contract code.

Finally, we extend our single-key FHE-based instantiation to support arbitrary computation with I/O privacy on multi-user inputs using a recent multi-key FHE scheme [17]. None of the specific schemes chosen are binding—we have just used current state-of-the-art. As stated previously, our framework is modular and future PPSC schemes will make use of what qualifies as state-of-the-art for *their* time.

Evaluation Results. Our construction involves benchmarking several cryptographic primitives and transaction operations to get a sense of their practicality. We are in the process of using existing implementations and libraries to report their costs.

1.2 Related Work

Hawk [20] was one of the first works to construct a private smart contract scheme using zero-knowledge proofs. They support a UTXO-based system in the style of Zerocash but, like Ethereum, they assume that their blockchain can support any Turing-complete program. Hawk requires the involvement of a semi-trusted manager—trusted with protecting the privacy of the users' inputs to the contract, but *not* trusted for execution or correctness. We do not use any such semi-trusted manager in our scheme.

Zether is a private transaction scheme that can support a limited class of private smart contracts on Ethereum—namely, those that can be expressed via homomorphic addition [8]. Using single-key FHE in the smartFHE framework, we will be able to support a larger class of applications via homomorphic multiplication. Additionally, smartFHE can also support arbitrary computation with I/O privacy on multi-user inputs—which is not possible using Zether. Unlike Zether, we do not support user anonymity, which we leave as future work.

Zexe [9] is a private computation scheme that allows users to perform arbitrary computations on their individual inputs offline. Along with hiding the inputs and outputs of a user-defined function, Zexe supports hiding the function itself. Their UTXO-based construction cannot support loops whereas our PPSC scheme can support public loops. As they follow the pure ZKP approach, Zexe cannot readily support computation with I/O privacy on inputs belonging to different users. The authors suggest using MPC but this requires users to coordinate the entire computation off-chain themselves. Our framework allows users

to perform computations with I/O privacy on multi-user inputs on-chain. No need to coordinate the computation or even be online.

Zkay [10] proposes a language and type system to easily identify who private data belongs to when writing smart contracts with I/O privacy. We view their language and type system as compatible with our smartFHE framework. However, they focus on the pure ZKP approach when constructing such contracts. Concerningly, they do not address how to resolve concurrency issues that result when implementing their private smart contracts.

There have been recent works, such as Ekiden [22], combining the use of trusted hardware with smart contracts. We do not use any trusted hardware in our system.

Finally, Kachina [23] formally defines and models private smart contracts. Their work is primarily of theoretical interest but can be viewed as compatible to ours in the sense that smart contracts providing input and output privacy can be realized as Kachina-style contracts.

2 Preliminaries

In this section, we introduce some of the cryptographic building blocks that will be needed in our schemes—namely, fully homomorphic encryption, zero-knowledge proofs, and digital signatures. As our framework builds upon ideas from Ethereum [7] and Zether [8], we also review these systems.

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

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

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

Ethereum. Ethereum [7] is a smart contract-enabled cryptocurrency that allows users to perform simple currency transfer in its native currency, Ether, as well as deploy complex applications via the creation of user-defined smart contracts. To this end, Ethereum introduces a Turing-complete language and maintains a virtual machine to execute contracts written in this language. Ethereum relies on an account-based model rather than the UTXO model like Bitcoin [2]. Thus, it introduces a more advanced notion of ledger state, which includes the state of all accounts in the system.

Ethereum provides two types of accounts: externally owned accounts (EOAs) that are controlled by users and contract accounts that are controlled by their contract code. The state of an EOA mainly consists of a nonce (to prevent replay attacks) and a balance. The state of a contract account includes its code and any state variables created by this code. 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 in Ethereum has some associated cost in terms of gas. Additionally, Ethereum's blockchain has a gas limit which constrains the total number of operations that can be executed in a single block.

Zether. Zether [8] brings privacy to currency transfers in Ethereum. It supports confidential transactions (in which the transfer amount and balances of the users are kept hidden), as well as anonymous transactions (that additionally hide the identities of the users involved). The latter, however, cannot be implemented on Ethereum as the cost exceeds the gas limit per block [8].

Zether is instantiated as a token on top of Ethereum with Zeth as its currency. Thus, users have accounts on the Ethereum network, as well as accounts for Zether's smart contract. To perform confidential transactions, Zether makes use of an additively homomorphic encryption scheme (namely ElGamal) and Sigma-Bullets (a Sigma protocol version of Bulletproofs) [8]. Account balances are encrypted and balance updates are done over the ciphertexts. To handle concurrency, incoming funds to private accounts are held in a pending state to ensure that ZKPs in private transactions are not deemed invalid if the sender's account balance changes. Pending currency transfers must be rolled over by the account owner at the start of an epoch (a period of k contiguous blocks). As Zether uses an additively homomorphic encryption scheme, only applications that can be represented in an additive manner support input/output privacy.

2.1 Fully Homomorphic Encryption

FHE supports computations directly on ciphertexts via the use of homomorphic addition and homomorphic multiplication. All currently known schemes rely on lattice-based cryptography, thus providing post-quantum security guarantees. Single-key FHE allows for arbitrary computation over data encrypted under the same key.

Multi-key FHE, on the other hand, allows for arbitrary computation over data encrypted under different keys. Ideally, the multi-key FHE scheme chosen for the smartFHE framework would be "multi-hop" [24]. Multi-hop schemes, unlike single hop schemes, allow evaluated ciphertexts to be used in further homomorphic computations.

FHE schemes model computation in one of three ways—as boolean circuits, modular arithmetic, or floating point arithmetic [25]. Floating point arithmetic will provide only approximate values and, thus, is a poor choice here since we need precise balance and transfer amounts for our ZKPs.

Specifically, we will use the BGV scheme [21] and then the Mukherjee-Wichs multi-key scheme [17] in our instantiation.

BGV Scheme. We use a single-key FHE scheme that models computation as arithmetic circuits—specifically, the BGV scheme [21]. It 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 level has its own set of (public and secret) keys. Bootstrapping can be used as an optimization to avoid having to specify the number of levels (i.e. multiplicative depth) in advance.

A ciphertext can be decrypted successfully if its noise is small enough. However, each time we perform homomorphic operations (especially multiplication), the ciphertext's noise grows. To help manage the noise growth, a refreshing procedure is introduced that can be performed by anyone.

Correctness of the BGV scheme is guaranteed so long as the noise does not wrap modulo q_j or q_{j-1} (we change from modulus q_j to smaller modulus q_{j-1} as part of the refreshing procedure to move between levels). Its semantic security follows from a standard hybrid argument from security of the basic Ring-LWE (Learning with Errors) encryption scheme [26].

Additional details on the BGV scheme can be found in Appendix A.

Mukherjee-Wichs Multi-key FHE Scheme. We consider the multi-key FHE scheme of [17] that extends the single-key GSW scheme. Unlike the BGV scheme, the Mukherjee-Wichs construction models computations as boolean circuits and requires a trusted setup. However, this scheme relies only on the hardness of LWE and has a one round decryption process. Note that this scheme provides both correctness and semantic security.

2.2 Zero-Knowledge Proofs

Our smartFHE framework uses FHE which can only be constructed using lattice-based cryptography. There have been recent improvements to lattice-based ZKPs (namely [27], [28], and [29]) 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 system [18]. We take this approach in our instantiation to obtain small proof sizes (in the single digit kilobyte range). We will also use Bulletproofs [30] to prove properties of the plaintext (such as the committed value being in a particular range). Both of these ZKP systems provide soundness, completeness, and zero-knowledge guarantees and can be made non-interactive using the Fiat-Shamir transform [31]. Additionally, neither requires a trusted setup.

Bulletproofs. Bulletproofs [30] support logarithmic-sized zero-knowledge proofs for general arithmetic circuits. Their security relies on the the hardness of the discrete log problem. Additionally, Bulletproofs are universal (so that a single

reference string can be used to prove any NP statement). Bulletproofs are particularly well-suited to short range proofs (i.e. proving that a committed value is in some range) which we make use of in our PPSC construction.

Short Discrete Log Proofs. This proof system [18] allows us to efficiently prove knowledge of a short vector $\vec{\mathbf{s}}$ such that:

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

for public $\mathbf{A}$ and $\vec{\mathbf{t}}$ over the polynomial ring $R_q = \mathbb{Z}_q[X]/(f(x))$, where $f(x)$ is a monic, irreducible polynomial of degree d in $\mathbb{Z}[X]$.

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

Then, to prove the linear relation, a variant of Bulletproofs is used, which differs from the original Bulletproofs construction in that the inner-product proof will be zero-knowledge [18]. Using the initial Pedersen commitment to $\vec{\mathbf{s}}$, we can use Bulletproofs to prove properties of the plaintext—such as a secret value being in a particular range. The soundness of the proofs is based on the discrete log problem whereas secrecy is based on Ring-LWE, a problem generally considered to be hard even for quantum computers [26].

2.3 Lattice-Based Signature Schemes

We require a signature scheme that is correct (so that valid signatures can be produced on arbitrary messages) and existentially unforgeable (to prevent adversaries from creating valid signatures on new messages having seen some previous valid signatures). This signature scheme will be used for signing transactions that originate from private accounts. In practice, we would like for such a scheme to be fairly efficient and compatible with our lattice-based FHE scheme.

In our instantiation, we suggest using the lattice-based Falcon signature scheme, a round 3 finalist for NIST's post-quantum cryptography competition [32].

3 Defining a Privacy-Preserving Smart Contract Scheme

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

Our PPSC scheme is applied on top of a public smart contract-enabled cryptocurrency such as Ethereum. It can be viewed as the extension needed to support privacy-preserving execution of smart contracts and payments on an account-based ledger. Hence, a PPSC scheme inherits all the public functionality and data structures found in the underlying public system. These include

the append-only ledger $\mathcal{L}$ that stores states for accounts (e.g. their balances, and contract code if applicable). Users have access to this ledger at any time and can initiate basic currency transfer transactions or deploy arbitrary smart contracts. Processing transactions and performing computations (the code portions of smart contracts) change the state of the ledger, where such changes are applied when a new block is mined. Thus, issuing any transaction or implementing any code relies on the latest ledger state (i.e. the latest changes reflected by the most recent mined block). In our definition below, we focus on the new modules needed to support private transactions and smart contract execution with private inputs and outputs.

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

- Setup*: Takes as input a security parameter λ. Outputs public parameters* pp*.*
- CreateAccount*: Takes as inputs public parameters* pp *and a privacy level (private or public). It generates key pair* $(\mathsf{sk}, \mathsf{pk})$ *and an address* addr *(derived from* pk*) with a postfix indicating if it is for a private or public account. It also initializes the account state consisting of a nonce* $\mathsf{ctr}[\mathsf{pk}] = 0$ *that is incremented with each transaction, a balance* $\mathsf{Bal}[\mathsf{pk}] = 0$ *associated with the account, and a lock entry* $\mathsf{Lk}[\mathsf{pk}] = \perp$ *indicating the address to which the account is locked ($\perp$ means the account is unlocked). If the account is private,* $\mathsf{Bal}[\mathsf{pk}]$ *will be secret. Finally,* CreateAccount *outputs the key pair, address, and state.*
- CreateTransaction*: Takes as input public parameters* pp*, transaction semantics, syntax, and information. Outputs a transaction* tx *of one of the following types:*
  - $\mathsf{tx}_{\mathsf{shield}}$: *Transfers currency from a public account to a private account. The transfer amount is public.*
  - $\mathsf{tx}_{\mathsf{deshield}}$: *Transfers currency from a private account to a public account. The transfer amount is public.*
  - $\mathsf{tx}_{\mathsf{privtransf}}$: *Transfers currency from one private account to another private account. The transfer amount is secret.*
  - $\mathsf{tx}_{\mathsf{lock}}$: *Locks a private account to some other account, thereby transferring account ownership to the recipient and preventing the locked account balance from being altered until unlocked.*
  - $\mathsf{tx}_{\mathsf{unlock}}$: *Unlocks a private account, returning control back to its owner. The transaction is only successful if it is issued by the same account to which the private account was locked.*
- VerifyTransaction*: Takes as inputs public parameters* pp*, transaction* tx*, and the transaction's syntax/semantics for the types mentioned above. Outputs 1 if* tx *is valid and 0 otherwise.*
- Compute*: Takes as inputs public parameters* pp*, an arithmetic or boolean circuit C, and inputs (public or secret) $x_1, ..., x_n$ for this circuit. If $x_1, ...x_n$ are public, then apply C as is on these inputs. If $x_1, ..., x_n$ are secret, transform C*

into an equivalent circuit C' operating on secret inputs and producing secret outputs, then apply C' to $x_1, ..., x_n$. If computation was successful, output 1. Otherwise, output 0.

– UpdateState*: Takes as inputs public parameters* pp*, current ledger state $\mathcal{L}$ which includes the state of all accounts, and a list of pending operations* $\mathsf{Ops} = \{\mathsf{op}_i\}$ *such that* op_i *can be a transaction* tx_i *or a computation* $\mathsf{Compute}(\mathsf{pp}, C_i, \{x_{i,1}, \ldots, x_{i,n}\})$ *as described above.* UpdateState *proceeds in blocks and epochs (an epoch is k consecutive blocks). Changes induced by all operations are reflected at the end of a block except for* $\mathsf{tx}_{\mathsf{shield}}$ *or* $\mathsf{tx}_{\mathsf{privtransf}}$*, which are processed at the end of the epoch (i.e. the last block in an epoch). Incoming transactions to a locked account will not be processed until the next epoch after which the account is unlocked. After applying* Ops *to $\mathcal{L}$ based on these rules, output an updated state $\mathcal{L}'$.*

Intuitively, correctness of a PPSC scheme requires that if we start with a valid ledger state and apply an arbitrary sequence of valid (or honestly generated) operations, the resulting state will also be valid. Towards this end, we define what constitutes a valid operation list, ledger state evolution, and an incorrectness game, INCORR, in which a challenger $\mathcal{C}$ and a ledger sampler $\mathcal{S}$ interact with each other. The goal of $\mathcal{S}$ is to produce an Ops that leads to an inconsistent ledger state. Thus, in this game, after receiving the public parameters pp, $\mathcal{S}$ samples a ledger state, a list of operations Ops, and two accounts; one is public and the other is private. Ops will be applied to each account starting with the same initial ledger state (for the public account, a public version of Ops will be used, but for the private account, an equivalent private version will be used). $\mathcal{S}$ wins the game if at the end, these accounts contain different balance values (meaning that the private operations in a PPSC scheme do not produce a consistent output with their public version). A PPSC scheme is correct so long as the advantage of $\mathcal{S}$ in winning the INCORR game is negligible (as defined below).

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

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

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

With respect to security, we define two requirements for a PPSC scheme—ledger indistinguishability and overdraft safety. To this end, we define a common security game between a challenger $\mathcal{C}$, representing the honest users, and an adversary $\mathcal{A}$. Both interact with the PPSC oracle $\mathcal{O}_{\mathsf{PPSC}}$. The adversary can ask $\mathcal{C}$ to perform various user algorithms. $\mathcal{A}$ can also submit his own operations to $\mathcal{O}_{\mathsf{PPSC}}$ for processing and request arbitrary subsets of pending operations to be processed.

Informally, ledger indistinguishability ensures that the ledger produced by the PPSC scheme Π does not reveal additional information beyond what was publicly revealed. We define a Ledger-Indistinguishability-Game to represent the interaction between $\mathcal{C}$ and $\mathcal{A}$. At some point in the game, $\mathcal{C}$ chooses a bit b at random (choosing between two operations) and the task of $\mathcal{A}$ is to guess which operation was selected (i.e. return b'). As in Zerocash [5], we define public consistency to rule out trivial wins by the adversary. Our definition requires taking into account not only the new transactions produced from CreateTransaction, but also the computations resulting from Compute.

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

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

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

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

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

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

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

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

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

4 The smartFHE Framework

In this section, we provide an overview of how the smartFHE framework can be used to support PPSCs. Specifically, we define the smart contract-enabled cryptocurrency architecture that we target, outline the functionalities that our framework supports, show how to handle concurrency issues, and provide a high-level description of how correctness and security of the resulting PPSCs are satisfied using FHE and ZKP. An instantiation of a PPSC using the smartFHE framework can be found in Section 5.

4.1 Architecture

Our framework can be viewed as extending a public smart contract-enabled cryptocurrency to support privacy. We require the underlying system to support an account-based model, a Turing-complete scripting language, and a virtual machine with a cost associated for each smart contract operation. For smartFHE, we consider extending Ethereum's design. Thus, our framework will provide additional functionalities to support both private currency transfer and smart contracts that operate on private data.

The default operation of smartFHE is the *public* mode—meaning that everything will be logged in the clear on the blockchain (including the account balances, smart contract code, and all transactions), and that smart contract code will be operating on public inputs/outputs. Working in the *private* mode requires explicit quantifiers to indicate that a user wants to conduct private payments or execute smart contacts that operate on private data or accounts and produce private outputs (for simplicity, we refer to these as private smart contracts).

As in Ethereum, smartFHE has two types of accounts: contract owned and externally (or user) owned. However, we further subdivide externally owned accounts into two types: *public* and *private*. Private accounts will be used to initiate private transactions and participate in private smart contracts. To differentiate between these two types (since both are represented by public addresses that could be hashes of the actual public keys), a prefix is added to an account address. Namely, we can let 00 indicate a public account address, whereas 11 will indicate a private one.

Given that smartFHE supports both private currency transfers and private smart contracts, we need to augment Ethereum's network protocol [7] with new types of transactions and cryptographic capabilities to permit operations on private accounts.

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

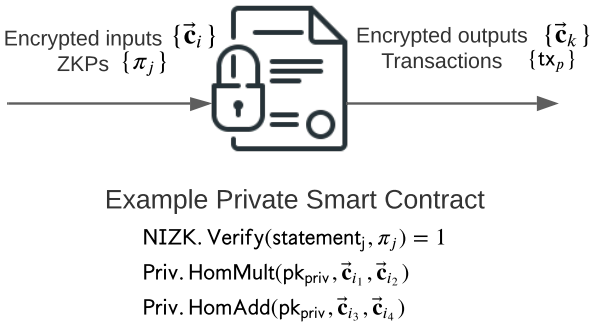

Fig. 1: A high-level description of private smart contract service in smartFHE.

4.2 Supported Functionality

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

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

Private Payments. For private payments, smartFHE allows users to issue transactions that hide the transfer amount and the users' balances.

To hide the balance of a private account, we use an FHE scheme (either single or multi-key) to encrypt the balance. Thus, examining the blockchain does not reveal the total amount of currency an account owns. Furthermore, for private transactions, the transfer amount will also be encrypted using the FHE scheme. This allows for updating the sender and receiver's account balances using homomorphic operations (i.e. homomorphic addition). Since his balance and transfer amount is hidden, the sender will need to provide ZKPs to show that certain conditions are met (i.e. he has enough currency in his account, the transfer amount is non-negative, the ciphertext is well-formed).

We only need an *additively* homomorphic encryption scheme to achieve private currency transfer. For example, we can restrict ourselves to the basic Ring-LWE encryption scheme [26] (rather than generating the full FHE key set from the BGV scheme) in our instantiation if only private currency transfer is desired.

Private Smart Contracts. smartFHE supports arbitrary computations on private inputs belonging to the same user if single-key FHE is used in the framework. If multi-key FHE is used in the framework, then smartFHE can support arbitrary computations on private inputs belonging to different users.

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

Since the code may operate on hidden values, the users participating in the contract need to provide ZKPs showing that their initial ciphertexts are well-formed and satisfy certain conditions (dependent on the application). Operations (aka circuits) over private accounts' data will be translated into fully homomorphic computations over these private inputs. smartFHE provides such translations behind the scenes for users.

Miners (who can be trusted for correctness and availability in the blockchain model) will check these ZKPs, perform the requested homomorphic computations directly on the ciphertexts, and update the blockchain state accordingly.

4.3 System Operation

We discuss the setup process, show how to handle concurrency issues resulting from distributed computation on private values, and outline the new transaction types we support.

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

System setup involves launching the 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.

For user-related setup, once system setup is complete, users can now join and create their own accounts. A user needs at least one of a public or private account (although users can own multiple accounts of each type if desired). For a public account, a user generates a key pair (specified by $(\mathsf{pk}_{\mathsf{pub}}, \mathsf{sk}_{\mathsf{pub}})$) to be used for signatures. The public account can be used to initiate public transactions and participate in public smart contracts. For a private account, a user generates a key set from the given single or multi-key FHE scheme (specified by $(\mathsf{pk}_{\mathsf{priv}}, \mathsf{sk}_{\mathsf{priv}})$) along with an appropriate signature scheme. Private accounts can be used to initiate private transactions and participate in private smart contracts.

Smart contract setup is dependent on whoever creates the smart contract. The contract creator may or may not participate in the contract himself. The smart contract will usually specify the sorts of inputs the contract will take in (which may be encrypted or in the clear), along with the operations to be performed on the inputs. If the smart contract offers privacy, there may also be ZKP verification step to ensure that contract-related conditions are satisfied with respect to the encrypted inputs.

Handling Concurrency. A challenge in designing privacy-preserving mechanisms for smart contract-enabled systems is how to handle both privacy and concurrency for accounts [23].

Suppose Alice submits a private transaction to be processed by the miners. While this transaction is waiting to be processed, Bob submits his own transaction in which he transfers currency to Alice. If Bob's transaction is processed *first* by the miners, then Alice's transaction will be subsequently rejected since her account's state has changed and the ZKP (generated with respect to her previous balance) is no longer valid.

Epochs. We adopt Zether's [8] approach to handle this issue; namely, we divide time into epochs consisting of some predetermined number of blocks (this value will be chosen in $\mathsf{System.Setup}$) and hold all incoming transfers to private accounts in a pending state until an epoch is complete. We ask private account users to submit deshielding and private transfer transactions at the start of an epoch so that they will be processed by the end of the epoch. Public accounts do not have to worry about submitting transactions at any particular point in time; additionally, incoming funds to public accounts are not subject to a waiting period (i.e. waiting until the start of the next epoch to be included in the account balance).

Our system (unlike Zether [8]) will roll over the funds to private account's balance automatically so that it can be spent at the start of the next epoch. Users do not have to explicitly call any algorithm for this to occur.

The length of an epoch must be chosen carefully (in the system setup phase) to ensure that a transaction submitted at the start of an epoch is processed before the epoch ends. When Alice and Bob submit their own respective transactions, miners will verify the transaction against the sender's current balance at that point in time. In practice, the sender should view the transaction amount as being deducted from his own account and reflected in his account balance immediately. We do this to prevent possible double-spending attacks. There is no issue with this approach as users should know how much currency they have sent to others. Additionally, transaction order will be enforced by the nonce value.

Private Account Locking. Our system supports more than just private currency transfer so dividing time into epochs and rolling over transfers does not suffice for handling front-running. For example, users may participate in private smart contracts that span an undetermined number of epochs. To handle front-running in these applications, we allow accounts to be *locked* to other accounts. We use locking differently from how it is used in Zether [8]; specifically, users can lock their account to *any* type of account, not just smart contract accounts.

If desired, epochs can be eliminated entirely in favor of using a locking mechanism for private accounts. We discuss this possibility in Section 6.

Transactions. As the smartFHE framework is used to support PPSCs, smartFHE supports all the transaction types mentioned in Section 3. Namely, we can support $\mathsf{tx}_{\mathsf{shield}}, \mathsf{tx}_{\mathsf{privtransf}}, \mathsf{tx}_{\mathsf{deshield}}, \mathsf{tx}_{\mathsf{lock}}$, and $\mathsf{tx}_{\mathsf{unlock}}$.

4.4 Security and Correctness

We briefly outline how the smartFHE framework supports a correct and secure PPSC scheme (full details can be found in Appendix C). Additionally, we show how the smartFHE framework handles system implementation level attacks, such as replay and front-running attacks.

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

Correctness. As discussed in the previous section, correctness of smartFHE's private operations will depend on the correctness of its cryptographic primitives—specifically, correctness of the signature scheme, correctness of the single or multi-key FHE scheme, and completeness of the ZKP system. Thus, any PPT adversary will win the incorrectness game INCORR (outlined in Section 3) with negligible probability assuming these requirements are satisfied.

Ledger Indistinguishability. Informally, an adversary that wins the Ledger-Indistinguishability-Game (outlined in Section 3) with non-negligible probability can be used to break semantic security of the (single or multi-key) FHE scheme and/or the zero-knowledge property of smartFHE's proof system.

Overdraft Safety. Informally, an adversary that wins the Overdraft-Safety-Game (outlined in Section 3) with non-negligible probability can be used to break the soundness of smartFHE's proof system.

Replay Attacks. In a replay attack, a transaction that has already been processed is replayed on the network. 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.

Front-running. To mitigate this risk, we initially hold all incoming transfers to private accounts in a pending state (until the end of an epoch) and allow users to lock their private accounts to other accounts. Thus, they can put their accounts on hold—preventing any state changes while their own private transactions are still pending.

5 Our Instantiation

We provide an instantiation of a PPSC scheme using the smartFHE framework. For our single-key instantiation, we use the BGV fully homomorphic encryption

scheme [21], Bulletproofs [30], and short discrete log proofs [18]. We can extend our instantiation to support private computation on multi-user inputs using the Mukherjee-Wichs multi-key FHE scheme [17].

5.1 Syntax

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

System Related. First, we perform the setup for the entire system. This includes choosing the parameters for the signature schemes (for the public and private account), the BGV fully homomorphic encryption scheme, and the non-interactive zero-knowledge proofs—specifically discrete log proofs [18] and Bulletproofs [30]). Please see Figure 2 for details. Important considerations for choosing some of these parameters are discussed in 5.3.

Public Account Related. A public account owner maintains key pair $(\mathsf{pk}_{\mathsf{pub}}, \mathsf{sk}_{\mathsf{pub}})$ to sign outgoing $\mathsf{tx}_{\mathsf{transf}}$ and $\mathsf{tx}_{\mathsf{shield}}$ transactions, an unencrypted balance $\mathsf{balance}$, and a nonce $\mathsf{ctr}[\mathsf{pk}_{\mathsf{pub}}]$ that is incremented with each transaction. We handle operations relating to public account in the same manner as Ethereum [7]. Ethereum uses the Elliptic Curve Digital Signature Algorithm [33] but in theory any efficient digital signature scheme (providing correctness and existential unforgeability) can be employed.

1. $\mathsf{Pub.CreateAccount}(\mathsf{pp})$: To create a public account, a user calls the $\mathsf{Pub.CreateAccount}$ algorithm which outputs key pair $(\mathsf{pk}_{\mathsf{pub}}, \mathsf{sk}_{\mathsf{pub}})$. $\mathsf{pk}_{\mathsf{pub}}$ will be used as the account address/to identify the account owner. In practice, the account address is usually a hash of the public key.
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} using the Elliptic Curve Digital Signing Algorithm on message m with secret key $\mathsf{sk}_{\mathsf{pub}}$.
4. $\mathsf{Pub.VerifySig}(\sigma_{\mathsf{pub}}, \mathsf{pk}_{\mathsf{pub}})$: Verifies if the signature σ_{pub} produced is valid and belongs to $\mathsf{pk}_{\mathsf{pub}}$. It outputs 1 if σ_{pub} is valid and 0 otherwise.

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 $\mathsf{ctr}[\mathsf{pk}_{\mathsf{priv}}]$ that is incremented with each transaction. The user will use a signature scheme to sign outgoing $\mathsf{tx}_{\mathsf{deshield}}$ and $\mathsf{tx}_{\mathsf{privtransf}}$ transactions.

1. $\mathsf{Priv.CreateAccount}(\mathsf{pp})$: To create a private account, a user calls the $\mathsf{Priv.CreateAccount}$ algorithm. $\mathsf{Priv.CreateAccount}$ generates the keys for the BGV scheme which are set to be the keys for the private account $(\mathsf{pk}_{\mathsf{priv}}, \mathsf{sk}_{\mathsf{priv}})$,

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

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

Initializes:

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

Also outputs:

- MAX, maximum currency amount smartFHE can support
- E, epoch length

Fig. 2: System setup

along with generating the keys for a lattice-based signature scheme $(\mathsf{sigpk_{priv}}, \mathsf{sigsk_{priv}})$. We have a secret key $\mathsf{sk_{priv}} = \{\mathbf{s}_j\}$ for each level j in the leveled BGV scheme. The public key $\mathsf{pk_{priv}}$ consists of matrix $\mathbf{A}_j$ along with auxiliary information $\tau_{s''_{j+1} \to s_j}$ for key switching. If we assume circular security, we will use the same public and secret key for every level [21].

2. $\mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk_{priv}}, \mathsf{m})$: Calls $\mathsf{BGV.Encrypt}$ on the message m. Outputs ciphertext $\vec{\mathbf{c}}$ which is encrypted with respect to level L.
3. $\mathsf{Priv.Decrypt}(\mathsf{pp}, \mathsf{sk_{priv}}, \vec{\mathbf{c}})$: Decrypts a ciphertext $\vec{\mathbf{c}}$ encrypted under $\mathsf{pk_{priv}}$ for level j by running $\mathsf{BGV.Decrypt}(\mathsf{pp}, \mathbf{s}_j, \vec{\mathbf{c}})$. Returns the corresponding plaintext message m.
4. $\mathsf{Priv.ReadBalance}(\mathsf{sk_{priv}})$: Returns the unencrypted balance $\mathsf{balance}$ belonging to a private account $\mathsf{pk_{priv}}$. If no such account exists, returns $\perp$.
5. $\mathsf{Priv.Sign}(\mathsf{sigsk_{priv}}, \mathsf{m})$: Produces a signature σ_{priv} on message m using the signature scheme for the private account.
6. $\mathsf{Priv.VerifySig}(\sigma_{\mathsf{priv}}, \mathsf{sigpk_{priv}})$: Verifies if the signature σ_{priv} produced is valid and belongs to $\mathsf{sigpk_{priv}}$. It outputs 1 if σ_{priv} is valid and 0 otherwise.

7. $\mathsf{CheckLock}(\mathsf{pk}_{\mathsf{priv}})$: Checks if the account corresponding to $\mathsf{pk}_{\mathsf{priv}}$ is currently locked. If $\mathsf{pk}_{\mathsf{priv}}$ is locked, returns the address of the account it is locked to. Otherwise, returns $\perp$.

Transaction Related. Users can engage in four types of transactions using their key pairs. Details are provided in Section 5.2.

1. $\mathsf{Transfer}(\mathsf{sk}_{\mathsf{pub}}^{\mathsf{from}}, \mathsf{pk}_{\mathsf{pub}}^{\mathsf{to}}, \mathsf{amnt})$: Transfer is used to send currency from one public account to another public account. It outputs $\mathsf{tx}_{\mathsf{transf}}$.
2. $\mathsf{VerifyTransfer}(\mathsf{tx}_{\mathsf{transf}})$: VerifyTransfer verifies if all the conditions for $\mathsf{tx}_{\mathsf{transf}}$ have been satisfied. If yes, it outputs 1. Otherwise, it outputs 0.
3. $\mathsf{Shield}(\mathsf{sk}_{\mathsf{pub}}^{\mathsf{from}}, \mathsf{pk}_{\mathsf{priv}}^{\mathsf{to}}, \mathsf{amnt})$: Shield is used to send currency from a public account to a private account. It outputs $\mathsf{tx}_{\mathsf{shield}}$.
4. $\mathsf{VerifyShield}(\mathsf{tx}_{\mathsf{shield}})$: VerifyShield verifies if all conditions for $\mathsf{tx}_{\mathsf{shield}}$ have been satisfied. If yes, it outputs 1. Otherwise, it outputs 0.
5. $\mathsf{Deshield}(\mathsf{sk}_{\mathsf{priv}}^{\mathsf{from}}, \mathsf{pk}_{\mathsf{pub}}^{\mathsf{to}}, \mathsf{amnt})$: Deshield is used to send currency from a private account to a public account. It outputs $\mathsf{tx}_{\mathsf{deshield}}$ which includes a ZKP.
6. $\mathsf{VerifyDeshield}(\mathsf{tx}_{\mathsf{deshield}})$: VerifyDeshield verifies if all conditions for $\mathsf{tx}_{\mathsf{deshield}}$ have been satisfied. If yes, it outputs 1. Otherwise, it outputs 0.
7. $\mathsf{PrivTransfer}(\mathsf{sk}_{\mathsf{priv}}^{\mathsf{from}}, \mathsf{pk}_{\mathsf{priv}}^{\mathsf{to}}, \mathsf{amnt})$: PrivTransfer is used to send currency from one private account to another private account. It outputs $\mathsf{tx}_{\mathsf{privtransf}}$ which includes a ZKP.
8. $\mathsf{VerifyPrivTransfer}(\mathsf{tx}_{\mathsf{privtransf}})$: VerifyPrivTransfer verifies if all conditions for a $\mathsf{tx}_{\mathsf{privtransf}}$ have been satisfied. If yes, it outputs 1. Otherwise, it outputs 0.
9. $\mathsf{Lock}(\mathsf{sk}_{\mathsf{priv}}^{\mathsf{from}}, \mathsf{addr}^{\mathsf{to}})$: First checks that $\mathsf{sk}_{\mathsf{priv}}^{\mathsf{from}}$ is not already locked by calling CheckLock. Locks private account corresponding to $\mathsf{pk}_{\mathsf{priv}}^{\mathsf{from}}$ to account corresponding to $\mathsf{addr}^{\mathsf{to}}$. A user can even lock his account to itself. Funds from transactions sent to $\mathsf{pk}_{\mathsf{priv}}^{\mathsf{from}}$ will not be rolled over into his balance until the epoch after which the account is unlocked. Outputs $\mathsf{tx}_{\mathsf{lock}} = (\mathsf{addr}^{\mathsf{to}}, \sigma_{\mathsf{lock}})$ where $\sigma_{\mathsf{lock}} = \mathsf{Priv.Sign}(\mathsf{sigsk}_{\mathsf{priv}}, (\mathsf{addr}^{\mathsf{to}}, \mathsf{ctr}[\mathsf{pk}_{\mathsf{priv}}]))$.
10. $\mathsf{Unlock}(\mathsf{pk}_{\mathsf{priv}})$: First checks that $\mathsf{pk}_{\mathsf{priv}}$ is locked by calling CheckLock. 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 $\mathsf{CheckLock}(\mathsf{pk}_{\mathsf{priv}})$. Outputs $\mathsf{tx}_{\mathsf{unlock}}$.

Private Smart Contract Related. Operations on inputs belonging to a private account will be translated into homomorphic computations.

1. $\mathsf{Priv.HomAdd}(\mathsf{pk}_{\mathsf{priv}}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: Runs BGV.HomAdd on the ciphertexts $\vec{\mathbf{c}}_1$ and $\vec{\mathbf{c}}_2$ (which are encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) to produce the sum of the two ciphertexts. Assuming they are encrypted with respect to the same level j, output $\vec{\mathbf{c}}_3 = \vec{\mathbf{c}}_1 + \vec{\mathbf{c}}_2 \mod q_j$. If not, use Priv.Refresh first to obtain two ciphertexts at the same level.
2. $\mathsf{Priv.HomMult}(\mathsf{pk}_{\mathsf{priv}}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: Runs BGV.HomMult on the ciphertexts $\vec{\mathbf{c}}_1 = (c_{1,0}, c_{1,1}), \vec{\mathbf{c}}_2 = (c_{2,0}, c_{2,1})$ (which are encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) to obtain the "product" $\vec{\mathbf{c}}_3 = (c_{1,0} \cdot c_{2,0}, c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}, c_{1,1} \cdot c_{2,1})$. We call Priv.Refresh on $\vec{\mathbf{c}}_3$ and output the result. If the initial ciphertexts are not

encrypted with respect to the same level, we use the $\mathsf{Priv.Refresh}$ procedure first to obtain two ciphertexts at the same level.

3. $\mathsf{Priv.Refresh}(\vec{\mathbf{c}}, \tau, q_j, q_{j-1})$: Runs $\mathsf{BGV.Refresh}$ on the ciphertext $\vec{\mathbf{c}}$ (encrypted with respect to $\mathsf{pk}_{\mathsf{priv}}$) using auxiliary information τ associated with private account $\mathsf{pk}_{\mathsf{priv}}$ to facilitate key switching and modulus switching from q_j to modulus q_{j-1}.

5.2 Instantiating the Payment Mechanism

We discuss our payment scheme in detail; namely, we show how users perform our new 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 (described in SEAL [34]) to represent integer value currency amounts for private accounts in our PPSC scheme. The technique is 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, we can pass our newly obtained polynomial (that represents some integer amount) into $\mathsf{Priv.Encrypt}$ to obtain an encryption.

Shielding Transaction. The 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 the receiver with private account $(\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{sk}^{\mathsf{to}}_{\mathsf{priv}})$ and encrypted balance $\vec{\mathbf{b}}'$. The sender will issue 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 of the network.

For the transaction to be verified, miners check that the following conditions are met:

- Valid signature from sender
- Receiver's public key exists/is valid
- Ciphertexts are well-formed
- Transfer amount is positive: $\mathsf{amnt} \in [0, \mathsf{MAX}]$

– Encrypted transfer amount matches plaintext amount with published randomness: $\mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{amnt}; \mathsf{r}) \stackrel{?}{=} \vec{\mathbf{c}}$
– Sender's remaining balance will be non-negative: $\mathsf{balance}^{\mathsf{from}} - \mathsf{amnt} \in [0, \mathsf{MAX}]$

If all conditions are satisfied (i.e. $\mathsf{VerifyShield}(\mathsf{tx}_{\mathsf{shield}}) = 1$), miners update the sender's account balance to $\mathsf{balance}^{\mathsf{from}} - \mathsf{amnt}$ and the receiver's balance to $\vec{\mathbf{b}}' + \vec{\mathbf{c}}$. The transaction $\mathsf{tx}_{\mathsf{shield}}$ is recorded on the blockchain.

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

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

The proof follows from a simple, straightforward application of discrete log proofs [18]. The sender signs the transaction along with his nonce $\mathsf{ctr}[\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}]$, producing signature $\sigma^{\mathsf{from}}_{\mathsf{priv}}$. He then broadcasts the transaction $\mathsf{tx}_{\mathsf{deshield}}$ to the miners of the network.

For the transaction to be verified, miners check that the following conditions are met:

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

If all conditions are satisfied (i.e. $\mathsf{VerifyDeshield}(\mathsf{tx}_{\mathsf{deshield}}) = 1$), miners update the sender's encrypted balance to $\vec{\mathbf{b}}' = \vec{\mathbf{b}} - \vec{\mathbf{c}}$ and the receiver's balance to $\mathsf{balance}^{\mathsf{to}} + \mathsf{amnt}$. The transaction $\mathsf{tx}_{\mathsf{deshield}}$ is recorded on the blockchain.

Private Transaction. In a private transaction, the sender has private account $(\mathsf{pk}^{\mathsf{from}}_{\mathsf{priv}}, \mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}})$ and maintains encrypted balance $\vec{\mathbf{b}}$. He wishes to send some amnt of currency to the recipient who is also using a private account $(\mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{sk}^{\mathsf{to}}_{\mathsf{priv}})$ and maintains some encrypted balance $\vec{\mathbf{b}}'$. However, the sender does not wish to reveal the transfer amount or his balance to other users in the system.

Thus, the 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: $\vec{\mathbf{c}}' = \mathsf{Priv.Encrypt}(\mathsf{pp}, \mathsf{pk}^{\mathsf{to}}_{\mathsf{priv}}, \mathsf{amnt}; \mathsf{r})$
- Sender's remaining encrypted balance: $\vec{\mathbf{b}}^*$
- Proof that $\vec{\mathbf{c}}, \vec{\mathbf{c}}'$ encrypt same transfer amount amnt with same randomness r and that this transfer amount is in $[0, \mathsf{MAX}]$ [4]
- Proof that sender's remaining balance is non-negative (i.e. $\mathsf{Priv.Decrypt}(\mathsf{pp}, \mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \vec{\mathbf{b}}^*) = \mathsf{balance}^* \in [0, \mathsf{MAX}]$)

The 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 of the network.

For the transaction to be verified, miners check that he following conditions have been met:

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

The sender's public key can be represented as matrix $\mathbf{A}$. The receiver's public key can be represented by the matrix $\mathbf{B}$. Let $\vec{\mathbf{m}}$ contain the transfer amount amnt and randomness. Then we can form the equation:

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

This equation verifies that $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}'}$ do in fact encrypt the same amount ammt with respect to the sender's public key $\mathbf{A}$ and the receiver's public key $\mathbf{B}$. Thus, we will need to show that $\vec{\mathbf{m}}$ satisfies the above equation and that the amount amnt represented in it is non-negative. This can be done using discrete log proofs [18]. We will also have another proof that the sender's remaining balance $\mathsf{Priv.Decrypt}(\mathsf{pp}, \mathsf{sk}^{\mathsf{from}}_{\mathsf{priv}}, \vec{\mathbf{b}}^*)$ is non-negative; this proof is identical to the one that will be provided in $\mathsf{tx}_{\mathsf{deshield}}$.

If all conditions are satisfied (i.e. $\mathsf{VerifyPrivTransfer}(\mathsf{tx}_{\mathsf{privtransf}}) = 1$), miners update the sender's encrypted balance to $\vec{\mathbf{b}} - \vec{\mathbf{c}}$ and the receiver's encrypted balance to $\vec{\mathbf{b}}' + \vec{\mathbf{c}}'$. The transaction $\mathsf{tx}_{\mathsf{privtransf}}$ is recorded on the blockchain.

[4] 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].

5.3 Public Parameter Considerations

We highlight some especially important considerations when choosing parameters for the system (performed as part of the System.Setup process before users can join the system). The following is not intended to be an exhaustive discussion.

Epoch Length. The number of blocks in an epoch, k, must be chosen based on:

- the length of time it takes to incorporate a transaction into the blockchain
- the *gap* between a user's view of the blockchain and the latest state of the blockchain

These are precisely the considerations Zether outlines for determining epoch length [8].

Overflow. Recall that we are working over a polynomial ring R_q for lattices. As part of the system setup, we will need to determine the maximum amount of currency our system can support along with the modulus q used in our ring. It is important that MAX is chosen to be much, much smaller than modulus q to prevent possible overflow for balance/transfer amounts.

Discrete Log Proofs. Our scheme relies on the hardness of both the Ring-LWE problem [26] *and* the discrete log problem. Let p be the order of the group $\mathbb{G}$ in which the discrete log problem is hard. For discrete log proofs [18], the authors observe that the modulus q (in R_q) for applications of the Ring-LWE problem is usually much smaller than the group order p. While their proof system works even if $q = p$, they note that proofs would be much less computationally efficient [18]. Recall that our primary motivation for using discrete log proofs in our PPSC system is to achieve small proof sizes (single digit kilobytes). Otherwise, there exist pure lattice-based constructions (such as [28]) that achieve proof sizes on the order of hundreds of kilobytes.

Levels and Bootstrapping. When using the leveled BGV scheme (as we suggest), the number of levels L must be specified as part of the setup process. L determines the maximum depth of the circuit that can be homomorphically evaluated along with the per-gate computation cost [21]. We must consider the sorts of private contracts we expect users to participate in and choose L accordingly to support a certain number of homomorphic multiplications. If such a decision is impossible to make and/or efficiency is of little concern, bootstrapping can be used to support computations of multiplicative depth exceeding L [21].

5.4 Supporting Multi-user Inputs using Multi-key FHE

We now consider supporting computation with I/O privacy for multi-user inputs, possibly using the Mukherjee-Wichs multi-key FHE scheme [17]. Users could

employ this scheme when generating a private account key pair. Transactions would proceed as they do when using single-key FHE, as neither homomorphic multiplication nor multi-user inputs are needed here.

Private smart contract operations could now take in data encrypted under different keys. Participating users will need to prove in zero-knowledge that each of their individual ciphertexts are well-formed and satisfy contract-dependent relations to ensure that the private computations proceed as expected. Short discrete log proofs could possibly be used here as they support proving LWE relations but with a loss in efficiency [18]. As before, users then ask the miners to perform the private computations directly on the ciphertexts after checking the ZKPs. To decrypt the result, users (whose ciphertexts were operated upon) would take part in a one round decryption process.

6 Optimizations and Extensions

We discuss some potential optimizations and extensions for our framework and instantiation.

Eliminating Epochs. Epochs may introduce synchronization issues for users. If desired, we can eliminate epochs entirely from the smartFHE framework. As part of a deshield or private transfer transaction, Alice will lock her account to itself. We can augment the network protocol so that once the ZKP is verified and the transaction is processed, Alice's account will automatically be unlocked. Note that we would still keep the $\mathsf{Lock}, \mathsf{Unlock}$ procedures to handle front-running issues in private smart contracts (to transfer ownership of the user account and keep away incoming transactions for an unknown amount of time).

Circular Security. If we assume circular security, the same keys can be used for all levels in the BGV scheme [21]. We leave this decision to whoever performs the initial setup of the scheme.

BFV Scheme. The BFV scheme is another type of fully homomorphic encryption scheme that models computation as arithmetic circuits [36]. It is closely related to the BGV scheme but is often considered simpler to work with since it is scale invariant. There are user-friendly libraries available for the BFV scheme such as SEAL [34]. If desired, the BFV scheme can be used in place of the BGV scheme in our instantiation with appropriate (minimal) changes.

7 Conclusion

In this paper, we defined a notion for a PPSC scheme and introduced smartFHE as a modular framework for supporting secure and correct PPSCs. smartFHE is the first framework to investigate the utility of fully homomorphic encryption, combined with non-interactive zero knowledge proofs, in the blockchain

model. Users can ask miners to execute arbitrary computations on their private inputs and produce private outputs. Our system supports both public and private modes with respect to payments and smart contracts. We believe that the blockchain model presents exciting opportunities for the full potential of homomorphic encryption to be realized.

References

1. D. Chaum, "Blind signatures for untraceable payments," in *Advances in cryptology*. Springer, 1983, pp. 199–203.
2. S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2019.
3. "Coin market capital," https://coinmarketcap.com/.
4. A. Biryukov and S. Tikhomirov, "Deanonymization and linkability of cryptocurrency transactions based on network analysis," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 172–184.
5. E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 459–474.
6. S. Noether, "Ring signature confidential transactions for monero." *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 1098, 2015.
7. G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014.
8. B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, "Zether: Towards privacy in a smart contract world," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 423–443.
9. S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 820–837.
10. S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. Vechev, "zkay: Specifying and enforcing data privacy in smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1759–1776.
11. D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: data structures and implementation." in *NDSS*, vol. 14. Citeseer, 2014, pp. 23–26.
12. R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 142–156.
13. I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe."
14. S. Halevi, Y. Lindell, and B. Pinkas, "Secure computation on the web: Computing without simultaneous interaction," in *Annual Cryptology Conference*. Springer, 2011, pp. 132–150.
15. U. Feige, J. Killian, and M. Naor, "A minimal model for secure computation," in *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, 1994, pp. 554–563.

16. S. Halevi, Y. Ishai, A. Jain, I. Komargodski, A. Sahai, and E. Yogev, "Non-interactive multiparty computation without correlated randomness," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 181–211.
17. P. Mukherjee and D. Wichs, "Two round multiparty computation via multi-key fhe," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016, pp. 735–763.
18. R. del Pino, V. Lyubashevsky, and G. Seiler, "Short discrete log proofs for fhe and ring-lwe ciphertexts," in *IACR International Workshop on Public Key Cryptography*. Springer, 2019, pp. 344–373.
19. D. Fiore, R. Gennaro, and V. Pastro, "Efficiently verifiable computation on encrypted data," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 844–855.
20. A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.
21. Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," Cryptology ePrint Archive, Report 2011/277, 2011, https://eprint.iacr.org/2011/277.
22. R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.
23. T. Kerber, A. Kiayias, and M. Kohlweiss, "Kachina-foundations of private smart contracts." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 543, 2020.
24. C. Peikert and S. Shiehian, "Multi-key fhe from lwe, revisited," in *Theory of Cryptography Conference*. Springer, 2016, pp. 217–238.
25. R. A. Hallman, K. Laine, W. Dai, N. Gama, A. J. Malozemoff, Y. Polyakov, and S. Carpov, "Building applications with homomorphic encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2160–2162.
26. V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 1–23.
27. J. Bootle, V. Lyubashevsky, and G. Seiler, "Algebraic techniques for short (er) exact lattice-based zero-knowledge proofs," in *Annual International Cryptology Conference*. Springer, 2019, pp. 176–202.
28. J. Bootle, V. Lyubashevsky, N. K. Nguyen, and G. Seiler, "A non-pcp approach to succinct quantum-safe zero-knowledge," in *Annual International Cryptology Conference*. Springer, 2020, pp. 441–469.
29. T. Attema, V. Lyubashevsky, and G. Seiler, "Practical product proofs for lattice commitments." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 517, 2020.
30. B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 315–334.
31. A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the theory and application of cryptographic techniques*. Springer, 1986, pp. 186–194.
32. P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, "Falcon: Fast-fourier lattice-based

compact signatures over ntru," *Submission to the NIST's post-quantum cryptography standardization process*, 2018.
33. D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.
34. "Microsoft SEAL (release 3.5)," https://github.com/Microsoft/SEAL, Apr. 2020, microsoft Research, Redmond, WA.
35. Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," *SIAM journal on computing*, vol. 38, no. 1, pp. 97–139, 2008.
36. J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.
37. S. Parsons, M. Marcinkiewicz, J. Niu, and S. Phelps, "Everything you wanted to know about double auctions, but were afraid to (bid or) ask," *City University of New York: New York2005*, 2006.

A Fully Homomorphic Encryption

A.1 Basic Ring-LWE Encryption Scheme

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

Let λ be the security parameter. All operations will be performed over the polynomial ring $R_q = \mathbb{Z}_q[x]/(f(x))$ where q is an integer and $f(x) \in \mathbb{Z}[X]$ is a monic, irreducible polynomial of degree d. The original BGV paper chooses the plaintext space to be 2 (such that $p = 2$ in the syntax below). We follow the presentation from short discrete log proofs here [18].

$\mathsf{E.Setup}(1^\lambda, 1^\mu)$: The setup algorithm takes as inputs security parameter λ and positive integer μ. $\mathsf{E.Setup}$ outputs public parameters $\mathsf{e.pp} = (p, q, d, \chi)$ where p is the size of the plaintext space (often chosen to be binary), q is a μ-bit modulus, $d = d(\lambda, \mu)$ is a power of 2 for $R = \mathbb{Z}[x]/f(x)$ where $f(x) = x^d + 1$, and $\chi = \chi(\lambda, \mu)$ is a "small" noise distribution. The parameters are chosen such that the scheme is based on a Ring-LWE instance that achieves 2^λ security against known attacks [21].

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

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

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

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

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

Correctness is straightforward. Semantic security of the encryption scheme is based on the hardness of Ring-LWE for ring R [26]. Recall that the Ring-LWE problem with appropriately chosen parameters can be reduced (via a quantum reduction) to the Shortest Vector Problem over ideal lattices. For details on the reduction, see [26].

A.2 BGV Scheme

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

Recall that each time we perform a homomorphic operation, the noise associated with the ciphertext grows. To prevent the noise from growing so large such that decryption fails, a technique called *modulus switching* is used to keep the noise level roughly constant [21]. When we multiply two ciphertexts $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}}'$ together, we get a long resulting ciphertext that is decryptable under a long secret key. Having to work with these long keys and ciphertexts impacts the efficiency of the scheme so BGV utilizes an additional technique called *key switching* that instead allows us to work with a smaller ciphertext and secret key in place of the originals. Both of these techniques—modulus switching and key switching—are encapsulated in the refreshing procedure that can be performed by anyone.

1. $\mathsf{BGV.Setup}(1^\lambda, 1^L)$: The setup algorithm $\mathsf{BGV.Setup}$ takes as inputs the security parameter λ and the number of levels L. It outputs the parameters $\mathsf{bgv.pp_j}$ for each level $j \in \{L, ..., 0\}$—which includes a modulus, noise distribution, and an integer. We also obtain a ladder of decreasing moduli that will be used in the modulus switching procedure in the algorithm $\mathsf{BGV.Refresh}$.

2. $\mathsf{BGV.KeyGen}(\{\mathsf{bgv.pp_j}\})$: The key generation algorithm BGV.KeyGen takes as inputs the parameters $\{\mathsf{bgv.pp_j}\}$. It outputs a secret key sk which consists of the secret key $\mathbf{s}_j$ for each level j from L down to 0 (obtained by running $\mathsf{E.SecretKeyGen}(\mathsf{e.pp_j})$), a public key pk which consists of public keys $\mathsf{pk_j}$ for each level j (obtained by running $\mathsf{E.PublicKeyGen}(\mathsf{e.pp_j}, \mathbf{s_j})$), and auxiliary information $\{\tau\}$ needed to facilitate the key switching procedure in BGV.Refresh.
3. $\mathsf{BGV.Encrypt}(\mathsf{bgv.pp}, \mathsf{pk}, \mathsf{m})$: The encryption algorithm BGV.Encrypt takes as inputs the scheme's parameters bgv.pp, the public key pk, and a message m. It runs $\mathsf{E.Enc}(\mathsf{e.pk_L}, \mathsf{m})$ (which is the same as $\mathsf{E.Enc}(\mathbf{A_L}, \mathbf{m})$) and outputs a ciphertext $\vec{\mathbf{c}}$.
4. $\mathsf{BGV.Decrypt}(\mathsf{bgv.pp}, \mathsf{sk}, \vec{\mathbf{c}})$: The decryption algorithm BGV.Decrypt takes as inputs the scheme's parameters bgv.pp, the appropriate secret key sk for the level, and a ciphertext $\vec{\mathbf{c}}$. It outputs the corresponding plaintext m by running $\mathsf{E.Decrypt}(\mathbf{s}_j, \vec{\mathbf{c}})$ (assuming the ciphertext was encrypted with respect to level j).
5. $\mathsf{BGV.HomAdd}(\mathsf{pk}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: BGV.HomAdd is used to add two ciphertexts together. It takes as inputs two ciphertexts—$\vec{\mathbf{c}}_1 = (c_{1,0}, c_{1,1}), \vec{\mathbf{c}}_2 = (c_{2,0}, c_{2,1})$—and the public key pk under which they are encrypted. If the ciphertexts are not encrypted with respect to the same level, then run the BGV.Refresh procedure first. We set $\vec{\mathbf{c}}_3 = (c_{1,0} + c_{2,0}, c_{1,1} + c_{2,1})$—the sum of the two ciphertexts $\vec{\mathbf{c}}_1$ and $\vec{\mathbf{c}}_2$ from performing component-wise vector addition. If desired, we can call BGV.Refresh on $\vec{\mathbf{c}}_3$ and output the "refreshed" result [21]. Otherwise, output $\vec{\mathbf{c}}_3$.
6. $\mathsf{BGV.HomMult}(\mathsf{pk}, \vec{\mathbf{c}}_1, \vec{\mathbf{c}}_2)$: BGV.HomMult is used to multiply two ciphertexts together. It takes as inputs two ciphertexts—$\vec{\mathbf{c}}_1 = (c_{1,0}, c_{1,1}), \vec{\mathbf{c}}_2 = (c_{2,0}, c_{2,1})$—and the public key pk under which they are encrypted. If the ciphertexts are not encrypted with respect to the same level, then run the BGV.Refresh procedure first. We obtain $\vec{\mathbf{c}}_3 = (c_{1,0} \cdot c_{2,0}, c_{1,0} \cdot c_{2,1} + c_{1,1} \cdot c_{2,0}, c_{1,1} \cdot c_{2,1})$, the "product" of the two ciphertexts. Finally, we call BGV.Refresh on $\vec{\mathbf{c}}_3$ and output this result.
7. $\mathsf{BGV.Refresh}(\vec{\mathbf{c}}, \tau, q_j, q_{j-1})$: BGV.Refresh takes as inputs a ciphertext $\vec{\mathbf{c}}$, auxiliary information τ to facilitate key switching from secret key $\mathbf{s}_j$ to $\mathbf{s}_{j-1}$, the current modulus q_j, and the next modulus q_{j-1}. It then does the following:
   (a) "Expands": Expand the ciphertext into a powers-of-2 representation.
   (b) "Switch Moduli": Scales the ciphertext to prepare it for modulus switching according to the new modulus q_{j-1}.
   (c) "Switch Keys": Performs the key switching procedure resulting in a new ciphertext $\vec{\mathbf{c}}'$ decryptable under key $\mathbf{s}_{j-1}$ for modulus q_{j-1}.

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

B Applications

In this section, we outline how smartFHE can be used to support several real-world applications using only single-key FHE.

B.1 Sealed-bid Auctions on Multiple Items

Most existing private cryptocurrency and smart contract systems offer bid privacy in auctions only when bidding on *a single* item. However, bidding on *multiple* items of a good is of interest in many financial and trading services.

The stock exchange, for example, allows potential buyers to bid on multiple shares of a stock using auctions [37]. These auctions allow buyers and sellers to specify not only the per-item price, but also the quantity (or number of shares) they are willing to buy or sell. To settle the auction, the auctioneer (which will be a smart contract in our case) needs to compute the market clearing price (which could be the highest per-item price among all bids), match buyers with sellers, and enforce currency transfer from the buyer to the seller.[5] The last condition requires multiplying together the ciphertexts of the per-item price and the quantity of items in the matched bid—a homomorphic multiplication operation. In what follows, we show how smartFHE can be used to implement this multi-item sealed-bid auction.

A smart contract, representing a simplified stock exchange, can be deployed to allow buyers and sellers to post bids and offers respectively. In its simplest form, each seller can publicly specify the maximum number of shares of a given stock she is willing to sell. Buyers can submit their sealed bids; each of which is composed of a private per-item price and a private quantity value (encrypted with respect to their private accounts). The auction proceeds in two phases: a bidding phase during which bidders post private bids along with proofs of their correctness and a matching phase during which those bidders reveal their bids to allow settlement. [6]

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

In the reveal phase, all bids that were not rejected (due to invalid ZKPs) will be given a timeout to be revealed. The exchange smart contract will decide the winning buyer(s) according to a given policy, such as first matching the bidder with the highest per-item price.

[5] For physical goods, we assume that the custody of the physical items and transferring them after settlement are provided by out-of-band means such as a custodial service. For digital assets, these can be handled by extending the contract code to hold custody and execute asset transfer.

[6] To prevent potential buyers from possibly spending all currency in their accounts (after posting bids but prior to auction settlement), we require bidders to lock their private accounts to the smart contract account as part of the bidding process. At the end of the auction, the smart contract will unlock all bidders' accounts.

Alice, the winner of the auction, will then create a private transfer transaction of the total amount—namely, the plaintext value $bid_{total} = np$—and present it to the smart contract. Alice's balance after the private transaction will be updated to $b'_{Alice} - (n'p')$. Bob's balance will be updated to $b'_{Bob} + bid'_{total}$ (which is the sum of Bob's private account balance and the winning bid amount encrypted under Bob's key).

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

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

However, such an approach exposes the system to a DoS attack. A malicious bidder can provide a valid ZKP proving that they can cover bid_{total}, but with invalid n and p values such that $np \neq bid_{total}$. This will be detected in the reveal phase if the bidder reveals the bid. At this stage, the exchange smart contract will reject such a fraudulent bid but *after* performing all computations needed to verify the attached ZKPs. Thus, an attacker may exploit this vulnerability and submit a large number of fraudulent bids, making the exchange unavailable to honest users. Although other means can be used here, such as punishing a malicious party financially via a penalty deposit, it may potentially be infeasible to compute a lower bound for this financial punishment (which would require knowing the utility gain of the attackers). Supporting homomorphic multiplication removes the need for additional countermeasures and makes our system secure against all efficient adversaries, rather than only rational and efficient ones.

B.2 Private Inventory Tracking

In certain trading scenarios, buyers and sellers may agree to trade a quantity of items that have yet to be produced. One such example is art work production, where a buyer is interested in buying several items of an art piece that an artist (after agreeing on the per-item price) will produce in the near future. In such a scenario, there is a chance that the seller may not produce the agreed upon amount within the specified timeline. To mitigate this risk, the buyer may resort to contacting several sellers to increase the chances of finalizing the deal on

time. However, only one seller will be able to finalize the trade and the effort of other sellers will be for nothing. This issue is of particular concern for highly customizable goods that may not be of interest to other buyers.

Thus, a binding trading contract between the buyer and seller is needed (which automatically settles the trade once the seller produces the items and financially punishes the seller if he does not meet the agreed-upon timeline). Although such functionality can be provided using traditional legal services, smart contracts provide a cheaper and more transparent way to satisfy this need, especially for frequent traders. Furthermore, with privacy-preserving smart contracts, no information about the price or the quantity sold is revealed to anyone other than the involved parties.[7]

In particular, Alice, the buyer, can create a smart contract to track the inventory of m products. For each of these m products, the contract will store a private per-item price, denoted as p'_i, and a private counter tracking the number of items produced so far, denoted as n'_i for $i \in \{1, \ldots, m\}$. The trading process is composed of two stages: deal term negotiation and item production. In the negotiation period, Alice negotiates the per-item price, quantity, and the timeline with Bob, the seller. This stage concludes with Alice registering Bob as the seller for a product in the list, and Bob recording the per-item price and quantity they agreed on. The latter is done by encrypting these two values under Alice's public key and storing them on the smart contract.[8] Furthermore, Bob records the production deadline which is simply the index of some future block on the blockchain.

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

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

[7] Similar to the previous application, we assume an external custodial service for physical goods, and proper code extension of the contract for custody and transference of digital assets.

[8] The contract code can be extended to allow Alice to dispute these values in case of mismatch. We leave such details out for now and focus on the contract functionality instead.

is issued within a given period, the trading contract will give Alice's penalty deposit to Bob. Otherwise, the trading contract will refund the parties their deposits and reset the inventory tracking variables to allow them to start another trade (if desired).

Without homomorphic multiplication, the trading smart contract would not be able to track private inventory on the fly and settle trades.

C Definitions and Proofs

In this section, we define notions for correctness and security of a PPSC scheme. We then prove that smartFHE supports a correct and secure PPSC scheme based on these notions. As mentioned earlier, our definitions are inspired by those in Zerocash [5] and Zether [8].

C.1 Correctness

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

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

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

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

shortly). Ops will be applied separately to acc_{pub} (as is) and acc_{priv} (with an equivalent private version of Ops here) starting with $\mathcal{L}$ in each case.

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

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

Definition 6 (Correctness of a PPSC Scheme—Definition 2 revisited). *A PPSC scheme $\Pi = (\mathsf{Setup}, \mathsf{CreateAccount}, \mathsf{CreateTransaction}, \mathsf{VerifyTransaction}, \mathsf{Compute}, \mathsf{UpdateState})$ is correct if no PPT ledger sampler $\mathcal{S}$ can win the INCORR game with non-negligible probability. In particular, for every PPT $\mathcal{S}$ and sufficiently large security parameter λ, we have*

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

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

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

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

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

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

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

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

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

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

The game proceeds as follows:

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

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

Correctness and smartFHE. Informally, correctness is derived from the correctness of the cryptographic building blocks. Every operation, whether a valid transaction or an arithmetic/boolean circuit computation, will be processed successfully in smartFHE and leads to a verifiable ledger update. This can easily be seen for each transaction type in the system. By relying on the completeness of the ZKP system, the correctness of the FHE scheme, the locking process (to lock account states to avoid invalidating any pending ZKPs), and the rolling over process at the end of each epoch, it can be shown that valid transactions will update the ledger state as expected. The same is true for Compute requests. For arithmetic/boolean computations on private inputs, the correctness of the results is based on the correctness of the chosen FHE scheme.

Accordingly, in the INCORR game, applying Ops to acc_{pub} and applying an equivalent private version Ops' to acc_{priv}, will lead to the same final balance value. Given that all balance values are not allowed to exceed some maximum value MAX determined by the system's setup, the homomorphic operations on account balances will not cause an overflow. Based on this discussion, we have the following lemma(we omit the formal proof which follows from the logic above):

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

C.2 Security

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

We first define the common security game Security-Game that will be used in overdraft safety and ledger indistinguishability.

Let $\mathcal{A}$ represent the adversary; $\mathcal{C}$, the challenger (who represents honest users in our system); $\mathcal{O}_{\mathsf{PPSC}}$, the oracle for our PPSC scheme. Both $\mathcal{C}$ and $\mathcal{A}$ have access to the oracle; however, $\mathcal{A}$ has full view of the oracle $\mathcal{O}_{\mathsf{PPSC}}$.

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

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

The following types of queries are permitted from $\mathcal{A}$:

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

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

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

Definition 7 (Overdraft Safety—Definition 4 revisited). *A PPSC scheme Π provides overdraft safety if for all* PPT *adversaries $\mathcal{A}$, the probability that*

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

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

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

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

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

Lemma 2. *Assuming the proof system satisfies soundness and smartFHE supports correct PPSCs, then smartFHE supports PPSC providing overdraft safety (cf. Definition 4).*

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

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

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

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

Ledger Indistinguishability. Ledger indistinguishability ensures that the ledger produced by the PPSC scheme Π does not reveal additional information beyond what was publicly revealed. We define a $\mathsf{Ledger\text{-}Indistinguishability\text{-}Game}$ to capture this. It is the same as $\mathsf{Security\text{-}Game}$ except that at some point in the game, $\mathcal{A}$ will send two publicly consistent instructions instead of one (we define publicly consistent instructions below, which is needed to rule out trivial wins of $\mathcal{A}$). $\mathcal{C}$

will execute one of these instructions based on bit b that is hidden from $\mathcal{A}$ which is chosen at random and in advance. $\mathcal{A}$ will have to guess which instruction $\mathcal{C}$ performed at the end of the game. Let b' be $\mathcal{A}$'s guess.

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

Definition 8 (Public Consistency). *Two instructions are publicly consistent if:*

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

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

Definition 9 (Ledger Indistinguishability—Definition 3 revisited). *A PPSC scheme Π satisfies ledger indistinguishability if for all* PPT *adversaries $\mathcal{A}$, the probability the $b' = b$ in the Ledger-Indistinguishability-Game is $1/2 + \mathsf{negl}(\lambda)$ where the probability is taken over the coin tosses of $\mathcal{A}$ and $\mathcal{C}$.*

Ledger Indistinguishability and smartFHE.

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

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

- A deshielding transaction.
- A private transfer transaction.

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

The same argument holds for two consistent private transactions. $\mathcal{A}$ has a negligible advantage of winning the game due to the zero-knowledge property

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

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