Pesca: A Privacy-Enhancing Smart-Contract Architecture*

Wei Dai
w.dai@baincapital.com
Bain Capital Crypto

ABSTRACT

Public blockchains are state machines replicated via distributed consensus protocols. Information on blockchains is public by default—marking *privacy* as one of the key challenges.

We identify two shortcomings of existing approaches to building blockchains for general privacy-preserving applications, namely (1) the reliance on external trust assumptions and (2) the dependency on execution environments (on-chain, off-chain, zero-knowledge, etc.) with heterogeneous programming frameworks.

Towards solving these problems, we propose Pesca—a privacy-enhancing smart contract architecture. Pesca utilizes generic building blocks such as threshold fully-homomorphic encryption (FHE), distributed key generation (DKG), dynamic proactive secrete sharing (DPSS), Byzantine-fault-tolerant (BFT) consensus, and universal succinct non-interactive zero-knowledge proofs (zk-SNARKs).

First, we formalize the problem of replicating state machines augmented with threshold decryption protocols and discuss how existing BFT consensus protocols can be adapted to this setting. We describe how to instantiate a blockchain with a fixed FHE public key and have FHE-encrypted chain states programmatically decrypted via consensus.

Next, we describe a smart-contract framework for engineering privacy-preserving applications, where programs are expressed—in a unified manner—between four types of computation: transparent on-chain, confidential (FHE) on-chain, user off-chain, and zero-knowledge off-chain.

Lastly, to showcase the generality and expressiveness of Pesca, we provide two simple application designs for constant function market makers (CFMMs) and first-price sealed-bid auctions (FPSBAs), both with maximal privacy guarantees.

1 INTRODUCTION

Public blockchains, aka distributed ledgers, are publicly-accessible state machines replicated via distributed consensus protocols. Public blockchains allow anyone to run a consensus node and any user to submit transactions to be processed by the state machine. As a result, most blockchain applications do not offer any privacy guarantee, leaving the financial information of users to public scrutiny.

There has been three main approaches to building general privacy-preserving applications in the public blockchain setting. The first is employing trusted execution environments (TEEs), such as Intel SGX. This is pioneered in the work of Ekiden [19] and implemented in projects such as Oasis and Secret Network. Such an approach effectively replaces the trust on the decentralized node operators to the, very much centralized, hardware providers.

The second approach is delegation of computation to off-chain trusted or multi-party-computation-based (MPC) services, following the pioneering work of Hawk [43]. There have been efforts on reducing the trust required on managers using zero-knowledge and MPC techniques [6]. But solutions in this space remain reliant on trust assumptions that are external to the blockchain in question.

Finally, an approach that has seen a huge surge in interest is use of non-interactive zero-knowledge proofs (NIZKs, or more colloquially ZKPs), following the pioneering work of Zerocash [8] and recent advances in succinct proofs systems (zk-SNARKs) such as Groth16 [39], Marlin [20], and Plonk [34]. There are many known techniques to obtain private direct payments on public blockchains using ZKPs [8, 15, 29, 33]. There have also been efforts on extending the approach of ZKPs to general programs. The most notable effort is the work of Zexe [12], which allows any general program to be executed off-chain and verified on-chain against state commitments in zero-knowledge. However, general zero-knowledge state updates cannot be made concurrent [25]. As a result, shared-state applications on Zexe-like platforms require usage and trust (in terms of privacy) of off-chain sequencers.

Summarizing the state of affairs, there are no known designs of blockchains that *natively support* programming of general privacy-preserving applications without *external trust assumptions*.

Problem 1: Can we build blockchains supporting *general* privacy-preserving applications *without external trust assumptions*?

Smart contracts and privacy-preserving applications. Smart contract architectures are the "operating systems" for blockchains, enabling user-programmable applications that time-share the execution capabilities of blockchains. However, for privacy preserving applications, there are three distinct execution environments that each application has to program for: on-chain, off-chain (done by user), and zero-knowledge computation. For example, the Ethereum blockchain uses the Ethereum virtual machine (EVM) execution environment, where code is often written in a contract programming language called Solidity. Client-side, in-browser, compute require frameworks such as `web3.js` [4] and is written in Javascript. Zero-knowledge circuits are programmed with libraries such as Circom [3] and Zokrates [5] in custom domain-specific languages. The heterogeneous nature of these tools makes it harder to design and engineer privacy-preserving applications.

Problem 2: Can we build smart contract platforms where privacy-preserving applications can be designed and engineered in a *unified framework*?

Pesca: *Privacy-Enhancing Smart-Contract Architecture.* Towards solving the above two key problems, we propose Pesca—a blueprint of a blockchain and a unified smart contract architecture to build privacy-preserving applications relying solely on the *honest super majority*—or so-called Byzantine—assumption of consensus nodes. We provide a technical overview in the rest of the section.

Threshold FHE and key management (Section 2). The high-level idea is simple, we utilize fully homomorphic encryption (FHE) to

*This is a preliminary draft of ongoing work.

compute over encrypted program states and inputs. To not have any single authority over these encrypted values, we utilize threshold cryptography to entrust these encrypted values to the *chain*, i.e. the consensus set. Specifically, we utilize a threshold FHE scheme [11] supporting Shamir-type keys [50], as well as are well-studied mechanisms to generate and maintain Shamir secret shares among a dynamic consensus sets, via the use of distributed key generation (DKG) [26, 40] and dynamic proactive secret sharing (DPSS) [37, 40, 45] schemes.

Blockchain protocols supporting threshold decryption (Section 3). There has been a vast amount of research on Byzantine fault tolerance (BFT) distributed consensus and state machine replication. The canonical setting requires a state machine with an efficiently computable update function as it is computed by all consensus nodes. In contrast, state machines of interest in our work are *not efficiently computable by any single party*, due to our usage of FHE threshold decryption. Instead, augmenting the efficient state update is a *threshold protocol* between consensus nodes. We first formalize and study the replication of state machines with such threshold release protocols.

A (t, n)-threshold release protocol is a one-round broadcast protocol between n parties that achieves desired security properties assuming t nodes are honest. Roughly, security asks that the release result $\mathsf{Release}(\mathsf{st})$ to be uncomputable by any adversary unless some honest node releases its share $\mathsf{PartRelease}(\mathsf{sk}_i, \mathsf{st})$, even with $t-1$ adversarially controlled nodes.

Next, we study the replication of state machines with threshold release protocols where the threshold t is set to the BFT-type guarantee of $2f+1$, where f is the number of *faulty* or *adversarial* nodes. We make a simple observation that any BFT-type protocol (e.g [14, 18, 51]) satisfying *safety* and *liveness* can be modified to replicate threshold release state-machines satisfying new notions called *release safety* and *release liveness* (assuming partially synchrony).

At this point, we are ready to list the main ingredients of our blockchain. We will fix:

- A (t, n)-threshold FHE scheme FHE with Shamir secrets, where t is between $f+1$ and $2f+1$. We assume that there is a distributed key generation phase where a chain FHE key pk is computed (and *fixed* for the lifetime of the chain) and each consensus node obtains its FHE secret key sk_i.
- A state machine M (whose exact functionality is fixed later) where an explicitly marked component of each state contains FHE ciphertexts to be decrypted and fed back to the machine as part of subsequent inputs. Note that M is a state machine with a threshold release protocol.
- A BFT-type consensus protocol satisfying liveness and correctness to replicate M.
- A universal (or transparent) zk-SNARK system Π.

We are interested in programming the above state machine M in an expressive manner utilizing the zk-SNARK.

The Smart Contract Architecture (Section 4). We design a smart-contract architecture enabling expressive programming of privacy-preserving applications. We do so by homogenizing different forms of computation. In particular, we consider four types of computation supported by Pesca and required by privacy-preserving applications:

- *Transparent on-chain computation* is the bread and butter of smart contracts. They are replicated among all consensus nodes in the plain.
- *Confidential on-chain computation* is computation done inside circuits of the threshold FHE scheme with a fixed public key pk. We let smart contract programmers specify arbitrary logic inside FHE circuits.
- *User off-chain computation* captures computation done by users to construct transactions. They have access to user secrets and (possibly stale) contract states.
- *Zero-knowledge computation*, done via zk-SNARK system Π, pertains to both the off-chain prover component run by users and the on-chain verification component run by contracts.

We write pseudocode to describe all four forms of computation (for an example, see Figure 3), with the understanding that one programming framework can be designed to unify these four different execution environments.

Attacks and mitigations. We remark that a naive design is vulnerable to attacks where an attacker copies the confidential FHE state of a target contract to an attacking contract (with arbitrary logic) for decryption. We design a mechanism that achieves proper *contract scope separation* and prevents such attacks, utilizing simulation knowledge soundness of the zk-SNARK.

Maximally privacy-preserving applications (Section 5). Finally, to showcase the expressiveness of our framework, we describe designs for constant function market makers (CFMMs) and first-price sealed-bid auctions (FPSBAs) with maximal privacy guarantees.

By maximal privacy guarantees we mean that, additional to information that is required to be released for liveness of the application:

- External observers do not learn anything about confidential user inputs or confidential applications state.
- Users only learn *binary* information on whether their inputs were validly executed or not.

Additionally, for both applications, the confidential FHE computation contain only a couple of arithmetic operations on (e.g. 64-bit) integers, making it practical under the performance characteristics of existing FHE schemes on GPUs.

To keep the confidential FHE state small, we utilize existing techniques to do privacy-preserving token accounting. In particular, underlying both applications is a multi-asset shielded pool inspired by the Orchard design [41] of Zcash [8]. To have it interoperate with the on-chain FHE service, we modify the action circuit to declare net value change and asset type encrypted under the fixed chain FHE key pk.

Privacy-preserving CFMMs. Constant function market makers [1], a special case of automated market makers have seen tremendous adoption following the launch of Uniswap. We provide a simple design where users construct exact trades that are executed in sequence. Our construction has no additional information leakage on

trades or asset reserves beyond the programmably-determined release of spot price. Furthermore, since users construct exact trades, they only learn a binary outcome.

Privacy-preserving FPSBAs. Recently, there has been a surge in auctions conducted on blockchains due to rising interest in non-fungible tokens (NFTs). We ask if auctions can be conducted so that non-winning bids are *never revealed* to anyone even after the settlement of the auction. Such property allows asset reserves of users to remain hidden and are extremely desirable for recurring auctions. We provide a simple application design for first-price sealed-bid auctions with such privacy guarantee.

2 PRELIMINARIES

In this section, we review the various building blocks including distributed consensus via Byzantine fault tolerance (BFT), universal and transparent zero-knowledge succinct non-interactive argument of knowledge (zk-SNARKs), threshold cryptography, as well as fully homomorphic encryption (FHE).

Global setup. Throughout, we fix n parties out of which f are adversarial. We assume that there is a global *setup phase* Setup where a public parameter pp and private keys for each party $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ are generated. All statements involving pp, $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ are quantified over the randomness of such a setup process. We write $[[\mathsf{sk}_i]]_S$ where $S \subseteq \{1, \ldots, n\}$ to denote private inputs or outputs for protocols, i.e. each party $i \in S$ takes private input or obtains private output sk_i. Looking ahead, Setup shall include any setup required for sub-protocols (e.g. distributed key generation) and cryptographic primitives (e.g. zk-SNARKs). We use λ to denote the security parameter and 1^λ its unary representation. We use $\mathrm{poly}(\cdots)$ and $\mathrm{negl}(\cdot)$ to denote the classes of polynomial and negligible functions respectively.

2.1 Consensus and State Machine Replication

A *state machine* M consists of an efficient state update function $M.\mathsf{Update}$, with function signature $M.\mathsf{Update}(\mathsf{st}, \mathsf{in}) \to \mathsf{st}'$, where $\mathsf{st} \in \{0,1\}^*$, $\mathsf{in} \in M.\mathsf{IN}$, and $\mathsf{st}' \in \{0,1\}^* \cup \bot$. For any state st, we say that $\mathsf{st} \to \mathsf{st}'$ is a *valid transition* upon input in if $M.\mathsf{Update}(\mathsf{st}, \mathsf{in}) = \mathsf{st}' \neq \bot$. A chain C of length n for M is a sequence $C = (\mathsf{st}_0 = \epsilon, \mathsf{in}_1, \mathsf{st}_1, \ldots, \mathsf{in}_n, \mathsf{st}_n)$, such that $\mathsf{st}_{i-1} \to \mathsf{st}_i$ is a valid transition under input in_i for all $i = 1, \ldots, n$. We say that $C \preceq C'$ if both C and C' are valid chains and that C is a prefix of C'.

Blockchains are state machines replicated via distributed consensus mechanisms. We focus on Byzantine consensus mechanisms for "streamlined blockchains" with a fixed set of participants per round, examples of such include Tendermint [14], Casper [16], HotStuff [51], Internet Computer Consensus [17], and Streamlet [18].

Streamlined BFT consensus protocols operate in a synchronous or partially synchronous setting and proceeds in rounds. Nodes send and receive messages to each other in each round via authenticated channels. Preceding each round r, the consensus algorithm at node i takes input a state machine input in_r^i. At the end of each round r, the consensus algorithm at node i outputs a *finalized* chain C_r^i for machine M.

Safety asks that any finalized chains from two honest nodes must agree, meaning that either $C_r^i \preceq C_{r'}^j$ or $C_{r'}^j \preceq C_r^i$ for any r and r' and honest nodes i and j. *Liveness* asks that if one honest node finalizes a chain C_i^r at round r, then any other honest node must finalize a chain extending C_i^r by some finite round $r' \geq r$.

2.2 Non-interactive Zero-knowledge

We require a universal or transparent zk-SNARK, examples of which include Sonic [44], Marlin [20], and various variants of Plonk [34].

We follow the definition framework of [20] for preprocessing SNARKs and adopt a game-based definition for NIZKs following [7]. An *indexed* relation R consists of tuples of the form (C, x, w), where C is called the index, x is the instance, and w the witness. Throughout, we fix an indexed relation $\mathbb{R}_{p,N}$ where each index C is the encoding of an *arithmetic circuit* over prime field $\mathbb{F}_p$ of size at most N. Relation $\mathbb{R}_{p,N}$ is defined as $(C, x, w) \in R_{\mathbb{F}_p}$ iff $C(x, w) \neq 0$ and $|C| \leq N$.

A universal non-interactive proof system Π for R_p consists of algorithms $\Pi.\mathsf{Setup}$, $\Pi.\mathsf{Compile}$, $\Pi.\mathsf{P}$, and $\Pi.\mathsf{V}$. We require the circuit compiler $\mathsf{Compile}$ (also known as the indexer) to be *deterministic*. We say that Π is *succinctness* if for any $\mathsf{pp} \leftarrow\!\$\ \mathsf{Setup}(\lambda)$, the running time of $\mathrm{V}(\mathsf{pp}, \cdot, x, \cdot)$ is $\mathrm{poly}(\lambda, |x|)$. When succinctness is not emphasized, and particularly inside security definitions and proofs, we opt to omit writing of $\mathsf{Compile}$ and write $\mathrm{P}(\mathsf{pp}, (C, x), w)$ and $\mathrm{V}(\mathsf{pp}, (C, x), \pi)$ to denote $\mathrm{P}(\mathsf{pp}, \mathsf{pk}_C, x, w)$ and $\mathrm{V}(\mathsf{pp}, \mathsf{vk}_C, x, \pi)$ respectively, where $(\mathsf{pk}_C, \mathsf{vk}_C) = \mathsf{Compile}(\mathsf{pp}, C)$. We consider the usual definition of completeness, zero-knowledge, and simulation knowledge soundness for the induced (non-universal and non-succinct) proof system on relation $\mathbb{R}'_p$ consisting of tuples of the form $((C, x), w)$.

Completeness. Perfect completeness says that for any adversary $\mathcal{A}$, the following game $\mathbf{G}^{\mathrm{cmpl}}_{\Pi,\mathcal{A}}$ outputs true with probability 1.

Game $\mathbf{G}^{\mathrm{cmpl}}_{\Pi,\mathcal{A}}(\lambda)$

1 $\mathsf{pp} \leftarrow\!\$\ \Pi.\mathsf{Setup}(1^\lambda)$; $(x, w) \leftarrow\!\$\ \mathcal{A}(1^\lambda)$; $\pi \leftarrow\!\$\ \Pi.\mathrm{P}(\mathsf{pp}, x, w)$
2 Return $\Pi.\mathrm{V}(\mathsf{pp}, x, \pi)$

Zero-knowledge and simulation knowledge soundness. To define zero-knowledge and simulation knowledge soundness, we require a zero-knowledge simulator $\Pi.\mathrm{S}$, its associated setup algorithm $\Pi.\mathsf{SSetup}$, as well as an extractor $\Pi.\mathsf{Ext}$. Consider the zero-knowledge game $\mathbf{G}^{\mathrm{zk}}_{\Pi,\mathcal{A}}$ given below.

Game $\mathbf{G}^{\mathrm{zk}}_{\Pi,\mathcal{A}}(\lambda)$

1 $b \leftarrow\!\$\ \{0,1\}$
2 $(\mathsf{pp}_0, \mathsf{td}) \leftarrow\!\$\ \Pi.\mathsf{SSetup}$
3 $\mathsf{pp}_1 \leftarrow\!\$\ \Pi.\mathsf{Setup}$
4 $b \leftarrow\!\$\ \mathcal{A}^{\mathrm{Pr},\mathrm{Ex}}(\mathsf{pp}_b)$
5 Return $(b = b')$

$\mathrm{Pr}(x, w)$:

6 Require $(\mathrm{R}(\mathsf{pp}, x, w))$
7 $\pi_0 \leftarrow\!\$\ \Pi.\mathrm{S}(\mathsf{pp}_0, \mathsf{td}, x)$
8 $\pi_1 \leftarrow\!\$\ \Pi.\mathrm{P}(\mathsf{pp}_1, x, w)$
9 $Q \leftarrow\!\cup\ (x, \pi_b)$
10 Return π_b

We define the zero-knowledge advantage of an adversary $\mathcal{A}$ to be $\mathbf{Adv}^{\mathrm{zk}}_{\Pi,\mathcal{A}}(\lambda) = 2 \cdot \Pr[\mathbf{G}^{\mathrm{zk}}_{\Pi,\mathcal{A}}(\lambda)] - 1$.

We move on to define simulation extractability [27, 38, 49]. Which is a strong notion requiring that the extractor to extract valid witnesses from forged proofs for an adversary even if the adversary has seen simulated proofs on possibly incorrect instances. Formally, consider the game $\mathbf{G}^{\mathrm{xt}}_{\Pi,\mathcal{A}}(\lambda)$ given below.

Game $\mathbf{G}^{\mathrm{xt}}_{\Pi,\mathcal{A}}(\lambda)$

1 $(\mathsf{pp}, \mathsf{td}) \leftarrow\!\$\ \Pi.\mathsf{SSetup}$
2 $\mathcal{A}^{\mathrm{Pr},\mathrm{Ex}}(\mathsf{pp})$
3 Return win

$\mathrm{Pr}(x)$:
4 $\pi \leftarrow \Pi.\mathsf{S}(\mathsf{pp}, \mathsf{td}, x)$
5 $Q \leftarrow\cup\ (x, \pi)$
6 Return π

$\mathrm{Ex}(x, \pi)$:
7 Require $((x, \pi) \notin Q)$
8 Require $(\Pi.\mathsf{V}(\mathsf{pp}, x, \pi))$
9 $w \leftarrow\!\$\ \Pi.\mathsf{Ext}(\mathsf{pp}, \mathsf{td}, x, \pi)$
10 win $\leftarrow \neg\mathsf{R}(\mathsf{pp}, x, w)$
11 Return win

We define the simulation extractable (XT) advantage of $\mathcal{A}$ against Π to be $\mathbf{Adv}^{\mathrm{xt}}_{\Pi,\mathcal{A}}(\lambda) := \Pr[\mathbf{G}^{\mathrm{xt}}_{\Pi}(\mathcal{A})]$. As usual, we say that Π is zero-knowledge and simulation knowledge sound if the corresponding advantages are negligible in λ for polynomial-time adversaries. We remark that Plonk and Sonic are known to simulation extractable (aka knowledge sound) [42] in the random oracle model.

2.3 Threshold FHE

Threshold decryption for a public-key encryption scheme allows splitting of the master decryption key into n key shares so that decryption can be done given any t shares without reconstructing the master key. Moreover, the reconstruction of the master key is not possible given less than t shares. This makes t-out-of-n threshold cryptography a natural candidate for distributed systems with similar adversarial assumptions, such as proof-of-stake (PoS) and Byzantine fault-tolerant (BFT) consensus systems.

It is well-known that public-key additively homomorphic encryption schemes such as ElGamal [31] and Paillier [47] can support threshold decryption [28] with Shamir [50] secret-shared keys. Fully homomorphic encryption (FHE) schemes can also be constructed to support threshold decryption [10, 11].

We fix a set of complete boolean gates $\mathcal{G}$[1] where each $g \in \mathcal{G}$ is a function of the form $\{0,1\}^{\ell} \to \{0,1\}$.

Threshold FHE [10]. A threshold FHE scheme, supporting $\mathcal{G}$, consists of algorithms five probabilistic polynomial-time algorithms Kg, Enc, Eval, $\mathsf{PartDec}$, FinDec.

- Key generation. $\mathsf{Kg}(1^{\lambda}, t, n) \xrightarrow{\$} (\mathsf{pk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n)$, upon input the unary security parameter, threshold t, and size n, key generation returns the public key pk and n secret keys.
- Encryption. $\mathsf{Enc}(\mathsf{pk}, m) \xrightarrow{\$} c$ for any message $m \in \{0,1\}$. Encryption of a bit m takes additional input only on the public key pk.
- Homomorphic evaluation. For any gate $g \in \mathcal{G}$. Suppose $c_1, \ldots, c_\ell$ are ciphertexts each encrypting one bit, then $c \leftarrow \mathsf{Eval}_{\mathsf{pk}}(f, (c_1, \ldots, c_\ell))$ is a ciphertext.
- Decryption of a ciphertext c, denoted $\mathsf{Dec}([[\mathsf{sk}_i]]_{\{1,\ldots,n\}}, c)$, is distributed protocol with two algorithms $\mathsf{PartDec}$ and FinDec. First, each i-th party computes partial decryption as $d_i \leftarrow\!\$\ \mathsf{PartDec}(\mathsf{sk}_i, c)$. Combining at least t honest partial decryptions, any party can obtain the final decryption via $m \leftarrow \mathsf{FinDec}(\mathsf{pk}, \{(i, d_i)\}_{i \in S})$.

We refer to [11, Section 5] for a formal treatment of compactness, correctness and security. Roughly, compactness says that decryption should be efficient regardless of the depth of the computation done on the ciphertexts. Correctness says that any homomorphic evaluations on honestly generated ciphertexts should yield the correct decryption. Standard (IND-CPA) security asks that an encryption of 0 to be indistinguishable from an encryption of 1. Simulation-based threshold security asks that as long as less than threshold t number of parties are corrupt, decryption shares are simulatable without access to real secret keys.

Verifiable partial decryption. We require a strengthening of threshold decryption where each partial decryption share can be publicly verified. Specifically, we require an algorithm $\mathsf{PartVerify}$, which decides the validity of a partial decryption d as a decision bit $\mathsf{PartVerify}(\mathsf{pk}, i, d_i, c)$. We say that $\mathsf{PartVerify}$ satisfies *correctness* if for $d_i \leftarrow\!\$\ \mathsf{PartDec}(\mathsf{sk}_i, c)$, we have $\mathsf{PartVerify}(\mathsf{pk}, i, d_i, c) = 1$. Intuitively, *security* requires that without knowledge of sk_i, an adversary cannot construct correct new decryption shares, even after observing partial decryption shares for adversarially chosen ciphertexts. We remark that, efficiency aside, verifiable partial decryption can be obtained generically via simulation extractable zk-SNARKs, where we additionally include a commitment of each secret key sk_i inside the public key pk. Specifically, we can consider $\mathsf{PartDec}'$ and $\mathsf{PartVerify}$, which utilizes a zk-SNARK for a circuit C computing a commitment and $\mathsf{PartDec}$, as follows.

Circuit $C(\mathsf{cm}, c, d; r_{\mathsf{cm}}, r_{\mathsf{PartDec}})$:
1 Assert $(\mathsf{cm} = \mathsf{Com}(\mathsf{sk}, r_{\mathsf{cm}})$ and $d = \mathsf{PartDec}(\mathsf{sk}, c; r_{\mathsf{PartDec}}))$

Algorithm $\mathsf{PartDec}'(\mathsf{pk}, (\mathsf{sk}_i, r_{\mathsf{cm}}), c; r_{\mathsf{PartDec}})$:
2 $d \leftarrow\!\$\ \mathsf{PartDec}(\mathsf{pk}, \mathsf{sk}_i, c; r_{\mathsf{PartDec}})$
3 $\pi \leftarrow\!\$\ \Pi.\mathsf{P}(\mathsf{pp}, C, \mathsf{pk}.\mathsf{cm}_i, c, d'; r_{\mathsf{cm}}, r_{\mathsf{PartDec}})$
4 Return (d, π)

Algorithm $\mathsf{PartVerify}(\mathsf{pk}, i, (d, \pi), c)$:
5 Assert $\Pi.\mathsf{V}(\mathsf{pp}, C, (\mathsf{pk}.\mathsf{cm}_i, c, d), \pi)$

Shamir secret sharing. For any prime field $\mathbb{F}_p$, the t-out-of-n Shamir secret sharing scheme [50] over $\mathbb{F}_p$ shares a secret $s \in \mathbb{F}_p$ into n shares $p(1), \ldots, p(n)$, where p is a uniform randomly sampled degree $t+1$ polynomial such that $p(0) = s$.

Based on the learning-with-errors (LWE) problem, a special class of FHE schemes [10, Definition 3.9], covering BGV [13] GSW [36] and FHEW [30] families, can be made compatible with Shamir secret keys. We summarize the modifications required on these schemes in Appendix A.

Note that naively as shown here, we require a trusted third party to run the key generation algorithm Kg and distribute key shares to relevant parties. This is undesirable for blockchain systems due to centralization. We assume the existence of a trusted third party in this work for simplicity but point out generic solutions to generate and manage Shamir secret shares. We do not formally specify or prove secure the composed systems here.

2.4 Threshold Key Management

Blockchain protocols deal with heterogeneous parties that may join and leave the protocol at arbitrary times. We are particularly interested in the t-out-of-n Shamir secret-sharing scheme [50], for which there are known distributed key generation (DKG) [26, 40] and dynamic proactive secret sharing (DPSS) [37, 40, 45] schemes. Furthermore, for large consensus sets where running DPSS over all nodes is not feasible, secrets can be kept and passed between small committees [9, 32, 35].

Distributed key generation. To generate Shamir secret shares without relying on a trusted party, a distributed key generation (DKG) protocol [26, 40] can be used. Concretely, the execution of a DKG protocol $\mathsf{DKG}(\lambda, n, t) \xrightarrow{\$} [[\mathsf{sk}_i]]_{\{1,\ldots,n\}}$ returns private outputs

[1] For simplicity this could be the singleton set containing the xnor gate. For efficiency, we may have many fan-in 2 and 3 gates such as and, or, and mux.

sk_i to each party i, which form a t-out-of-n Shamir secret sharing of some master secret that is never reconstructed.

Dynamic proactive secret sharing. To support a dynamic consensus sets where nodes are joining and leaving, dynamic proactive secret sharing (DPSS) [37, 40, 45] can be utilized. Concretely, the execution of a DPSS protocol $\mathtt{DPSS}([[\mathsf{sk}_i]]_S) \$\longmapsto [[\mathsf{sk}'_i]]_{\{1,\ldots,n\}}$ requires at least t honest participants with their respective Shamir secret shares sk_i and returns refreshed Shamir secret shares sk'_i for each participant i, while keeping the same master secret.

For the rest of the paper, we fix a threshold FHE scheme FHE with Shamir secrets where Kg is replaced by a compatible Shamir DKG protocol.

3 REPLICATION OF STATE MACHINES WITH THRESHOLD RELEASE

Blockchains are state machines replicated via distributed consensus protocols. While the state machine model is general, it only captures efficiently computable transitions, as the state machine is executed independently by each consensus node. In particular, the model does not capture transitions that require interaction between consensus to evaluate. One example of such blockchain design is threshold decryption. For example, both Ferveo [46] and Penumbra [48] modify the underlying consensus algorithm to support threshold decryption. In these protocols, the set of nodes will run a threshold decryption protocol to decrypt a pre-specified part of the state. The state machine then proceeds by taking input of the result of threshold decryption.

In this section, we provide a general definition of state machines with threshold release protocols and discuss how existing consensus protocols can be modified to replicate these state machines.

3.1 State Machines and Release Functions

State release function. Let M be a state machine. A release function for M is a function $\mathsf{Release} : \{0,1\}^* \to \{0,1\}^*$ which takes input a state st and outputs the *state release information* $\mathsf{Release}(\mathsf{st})$. Looking ahead, the release function is not assumed to be efficiently computable as a function but can be evaluated efficiently via an external protocol.

To model useful applications, it is often paramount for the release information to be fed back into the state machine as input. To model this, we consider the input space IN as $\{0,1\}^* \times (\mathbb{N} \times \{0,1\}^*)^*$, where the second component encodes an arbitrary sequence of release information from previous states.

For a particular sequence of inputs $\mathsf{in}_1, \ldots$ and correctly computed valid states $\mathsf{st}_0, \ldots$, we say that st_i is released with delay d if in_{i+d} contains $(i, \mathsf{Release}(\mathsf{st}_i))$. We let $M.\mathsf{Update}$ encode the validity conditions on the necessary delay. For example, $M.\mathsf{Update}$ could reject inputs in_{i+1} unless it contains a release of the previous state $\mathsf{Release}_{\mathsf{st}_i}$ is provided.

3.2 Replicating Threshold-release Machines

Threshold release protocol. We focus on a release function with threshold release protocols. A (t,n)-threshold protocol ThRelease to compute Release consists of four efficient algorithms Setup, PartRelease, Combine, and PartVerify. Consider the following (strong) correctness game against any polynomial-time adversary $\mathcal{A}$.

Game $\mathbf{G}^{\mathsf{thcor}}_{\mathsf{ThRelease},\mathcal{A}}(\lambda)$

1 $(\mathsf{pp}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n) \leftarrow\$ \mathsf{Setup}(1^\lambda)$
2 $(x, D) \leftarrow\$ \mathcal{A}(\mathsf{pp}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n)$
3 Assert $(|D| \geq t$ and $\forall (i,d) \in D : \mathsf{PartVerify}(\mathsf{pp}, i, d, x) = 1)$
4 Return $(\mathsf{Combine}(\mathsf{pp}, D) = \mathsf{Release}(x))$

We say that ThRelease is correct if the probability of the above game is negligibly close to 1 for any polynomial-time adversary $\mathcal{A}$, i.e. $1 - \Pr[\mathbf{G}^{\mathsf{thcor}}_{\mathsf{ThRelease},\mathcal{A}}(\lambda) \Rightarrow \mathsf{true}]$ is negligible in λ. We call a set D satisfying the check at line 3 *a release witness* for x. We say that (y, D) is a *valid release* of x if $\mathsf{Combine}_{\mathsf{pp}}(D) = y$ and that D is a valid release witness for x.

Security of threshold release protocols. We consider a natural unpredictability-style security notion for threshold release protocols. Intuitively, we want an adversary to not be able to compute $\mathsf{Release}(\mathsf{st})$ for any state st unless it obtains the right shares, even with access to corrupted shares. Formally, consider the below game.

Game $\mathbf{G}^{\mathsf{thsec}}_{\mathcal{A}}(\lambda)$

1 $(\mathsf{pp}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n) \leftarrow\$ \mathsf{Setup}(1^\lambda)$
2 $S \leftarrow\$ \mathcal{A}(\mathsf{pp})$; Assert $|S| < t$
3 $(x, y) \leftarrow\$ \mathcal{A}^{\mathsf{Eval}}(\{sk_i\}_{i \in S})$
4 $y' \leftarrow \mathsf{Combine}(\mathsf{pp}, \{\mathsf{PartRelease}(\mathsf{sk}_i, x)\}_{\{1,\ldots,t\}})$
5 Return $(x \notin Q$ and $(y = y'))$

$\mathsf{Eval}(i, x)$:
6 $Q \leftarrow\cup\, x$; Return $\mathsf{PartRelease}(\mathsf{sk}_i, x)$

We say that ThRelease is secure if $\Pr[\mathbf{G}^{\mathsf{thsec}}_{\mathsf{ThRelease},\mathcal{A}}(\lambda) \Rightarrow \mathsf{true}]$ is negligible in λ.

Let M be a state machine with a (t,n)-threshold release protocol for Release. We describe how existing consensus protocol on n nodes can be modified to replicate M with the following property:

- *Release livensss.* If state st_i is finalized, then a valid release $(\mathsf{Release}(\mathsf{st}_i), D)$ appears in the input in_j of some finalized state st_j for $j > i$.
- *Release safety.* If a valid release $(\mathsf{Release}(\mathsf{st}_i), D)$ appears in the input in_j of any state st_j (not necessarily finalized), then it must be that st_i is finalized (for some honest node).

We remark that the above two properties can be achieved by modifying any BFT blockchain protocols as long as $f+1 \leq t \leq 2f+1$ as follows:

(1) In any round, if a state st becomes finalized for node i, then it will release and broadcast its release share $\mathsf{PartRelease}_{\mathsf{sk}_i}(\mathsf{st})$.
(2) If a node receives some set D of at t release shares that forms a valid release witness for some st, then it computes $\mathsf{Release}(\mathsf{st}) = \mathsf{Combine}_{\mathsf{pp}}(D)$.
(3) In any round, if the leader knows a finalized state st and its release information $\mathsf{Release}(\mathsf{st})$ with release witness D but has not observed $\mathsf{Release}(\mathsf{st})$ in any proposed blocks, it will include $(\mathsf{Release}(\mathsf{st}), D)$ in the newly proposed state machine input in.

We claim that with the above modifications, any BFT blockchain protocol satisfying safety and liveness additionally satisfies release safety and release liveness.

Proposition 3.1. *If a BFT protocol satisfies liveness, then the modified BFT protocol satisfy release liveness assuming partially synchronous network.*

Proof. Suppose a state st becomes finalized for an honest node i. By liveness, we know that all honest nodes will eventually consider st finalized, which by modification (1) means that all honest nodes broadcast their release shares. Hence, in some later round, some honest leader node must have obtained $2f + 1$ valid shares and be able to compute Release(st) since $t \leq 2f + 1$. □

Proposition 3.2. *If a BFT protocol satisfies safety and* ThRelease *is a secure threshold release protocol, then the modified BFT protocol satisfy release safety.*

Proof. Suppose a valid release $(\text{Release}(\text{st}), D = \{(i, d_i)\})$ appears in some input in_j. We know that with overwhelming probability, due to the security of ThRelease, that all d_i values are computed via $\text{PartRelease}(\text{sk}_i, \text{st})$. This means that at least one $(t - f \geq 1)$ honest nodes have considered st_i finalized. □

4 PRIVACY-ENHANCING SMART CONTRACT

In this section, we describe an architecture for expressively programming of state machines with threshold release.

Overview of a smart contract platform. We base our architecture off of standard smart contract platforms such as Ethereum, Solana, and Dfinity. Smart contracts are stateful deterministic programs that run on a blockchain. We assume that smart contracts can invoke "public" functions of other contracts and that such calls are *blocking*. We assume that each execution of a smart contract is initiated by an end-user via a *transaction*. Errors encountered during the execution of smart contracts *revert* the entire transaction. Each contract has a distinct owner and a contract identifier which is programmatically accessible via the variable `this`.

Compared to smart contract platforms such as Ethereum, our architecture adds the ability to programmatically invoke threshold decryption of an FHE scheme. As a result, we have four types of computation of interest.

Transparent on-chain computation. This is the bread and butter of smart contract platforms. Transparent on-chain computation is replicated among the set of all consensus nodes and hence public and transparent. In pseudocode, they are denoted as either *public* or *private* contract functions depending on standard programming access control idioms. Public refers to the fact that it can be invoked by external contracts. Private refers to the fact that it can only be invoked within the contract itself.

User off-chain computation. Often, users need to provide structured inputs to on-chain smart contracts. This is especially true for users that need to provide zero-knowledge proofs. We denote user computation as *User functions*. User functions are allowed access to contract storage as well as user-specific secret information such as keys. Note that user off-chain computation are never deployed to chain.

Zero-knowledge off-chain computation. With the help of a universal or transparent zk-SNARK Π, we can express any validity check as a circuit. In pseudocode, they are denoted as *ZK circuits*. Zero-knowledge circuits only return a boolean value and *do not have access to contract storage*. We assume that the for each ZK circuit C, the contract owner computes $(\text{pk}_C, \text{vk}_C) \leftarrow \Pi.\text{Compile}(C)$ before deploying the contract. In the same contract, $C.\texttt{verify}(x; \pi)$ is a shorthand for $\Pi.\text{V}(\text{vk}_C, (\texttt{this}, x), \pi)$. For any user function, $C.\texttt{prove}(x; w)$ is a shorthand for $\Pi.\text{P}(\text{pk}_C, (\texttt{this}, x), w)$. Here, we draw attention to the use of the contract identifier `this` as part of the instance. The reason is to properly scope separate proofs for one contract from another.

Confidential (fully homomorphic) on-chain computation. With the help of a threshold FHE scheme, we can express any computation on confidential states as *FHE circuits*. FHE circuits do not have access to contract storage. Inside any contract function and for any FHE circuit C of the contract, we write $C.\text{Eval}(\text{ein})$ as a shorthand for $\text{FHE.Eval}_{\text{pk}}(C, \text{ein})$.

Threshold decryption. Additionally, at any point during a transparent on-chain computation, a call to *threshold decryption* can be made. The call is made with three parameters, some FHE ciphertexts est to be decrypted, a variable x to store the decrypted value, and an asynchronous code block that is executed after the threshold decryption result is made available. We write this as "Async $x \leftarrow \texttt{ThDec}(\text{est})$" followed by the code block. We allow state read and access in the asynchronous code block, with the understanding that the content of the contract state can change arbitrarily between the threshold decryption call and the execution of the asynchronous code block.

4.1 Safety of FHE application states

Since anyone can deploy a contract that invokes threshold decryption, an adversary can decrypt any previously observed FHE state and call threshold decryption on it. To prevent this attack, we employ the zk-SNARK to enforce "knowledge of secret" of any new FHE ciphertexts. Consider the following zero-knowledge circuit and contract function.

```
Contract FHEBase
Priv Func VerifyFHEInput(eb):
  1 Assert KnowledgeCheck.verify(eb; eb.π)
ZK Circuit KnowledgeCheck(eb; b, r):
  2 Assert (eb = FHE.Enc_pk(b; r))
```

Figure 1: Base contract for utilizing FHE state and threshold decryption.

We remark that each contract that utilizes the on-chain FHE functionality must use the `FHEBase` contract as a library. However, proofs constructed for `KnowledgeCheck` are *not compatible* across different contracts, as we assume the contract identifier `this` is inserted into the public instance, i.e. $\texttt{KnowledgeCheck.verify}(eb; \pi) = \Pi.\text{V}(\text{vk}_{\texttt{KnowledgeCheck}}, (\texttt{this}, \text{eb}), \pi)$. By simulation extractability of Π, proofs for different contract identifiers are properly separated.

We require the state machine to ensure that:

- An input FHE ciphertext eb is marked *valid* only if it contains a valid knowledge proofs, i.e. $\texttt{VerifyFHEInput}(\text{eb}) = \texttt{true}$.
- Inputs to $\text{FHE.Eval}_{\text{pk}}(\cdot, \cdot)$ and `ThDec` must only contain *valid* FHE ciphertexts.
- Outputs of $\text{FHE.Eval}_{\text{pk}}$ are marked as *valid*.

The above can be done by exposing $\mathsf{FHE.Eval}_{\mathsf{pk}}$ and ThDec as special "system calls."

We claim that if the state machine enforces the above validity rules, then trivial attacks on privacy cannot occur.

5 APPLICATIONS

In this section, we outline two applications programmed via Pesca framework–constant function market makers (CFMMs) and first-price sealed-bid auctions (FPSBAs), both providing maximal privacy guarantees.

We chose these two applications to showcase the expressiveness of the framework due to their simplicity and limited shared state–both applications only maintain a small constant shared state regardless of how many users are interacting with the application. To achieve this property, we utilize techniques from Zerocash to do token accounting and usage. We utilize an on-chain threshold FHE service to compute over shared states of CFMMs and FPSBAs, which are of size independent of the number of users interacting with these applications. Connecting between them, we require that spends to declare inputs encrypted to the FHE service, which is done inside the zero-knowledge circuits for spending tokens (called Action circuits).

5.1 Shielded Assets

At the foundation of our privacy-preserving trading applications is a design of a multi-asset shielded pool supporting token usage that declares inputs to the on-chain FHE service. For this shielded pool, we take a design approach closely following the Orchard design of the Zcash protocol [41], which is the third iteration of the implementation effort of the pioneering work of Zerocash [8]. The pseudocode of the contract is given in Figure 2. We explain each component in detail below.

Contract state. The state of our smart contract consists of a Merkle tree of notes MT and a set of spent note nullifiers NS. We assume a Merkle tree implementation supporting insertion and lookups via MT.add and MT[path]. We also assume that the Merkle tree implementation exposes a list of periodically snapshotted historical Merkle tree roots via MT.RTS.

Notes. A note is a fundamental data unit encoding four values, the public key pk of the owner, the asset identifier typ, a nonce ρ, a value $\mathtt{val} \in V$, where V is a set of allowable values, e.g. $V = \{0, \ldots, 2^{64} - 1\}$. We write $\mathtt{note} = (\mathsf{pk}, \mathtt{typ}, \mathtt{val}, \rho)$. Associated with each note are two algorithms NoteCommit and DeriveNullifier. Algorithm NoteCommit takes input a note and returns a note commitment ncm. Algorithm DeriveNullifier takes input a secret key sk and a note note to return a nullifier nf.

The Action circuit. The action circuit declares a transaction that spends a note (via revealing a nullifier nf) and creates a note (via revealing a note commitment ncm). It also declares the two fields tx.ev and tx.et, encrypting the net value change and asset type (of both notes) under the FHE public key pk. We let the action circuit declare these values so that they can be passed to FHE circuits as inputs.

Transaction creation. The user function CreateTx contains code that creates a transaction. It takes input a secret key sk, some actionInfo containing path to a note oldNote owned by the user, a newNote to be created, and encryption randomness $r_{\mathtt{val}}$ and $r_{\mathtt{typ}}$. It prepares a transaction tx containing fields rt, ncm, nf, ev, et, and π. The Merkle root rt of the transaction is set to the most recently available Merkle root MT.rt. The note commitment ncm is set to the commitment of the created newNote. The nullifier nf is set to the derived nullifier of note to be spent oldNote using the user secret key sk. The encrypted value ev is set to the net difference (a signed integer) between oldNote.val and newNote.val. The encrypted type et is set to the encryption of the type that both notes must share. Finally, the validity proof π is computed by invoking the zk-SNARK prover for the Action circuit.

Transaction processing. Unlike Zerocash, we do not balance of values when transactions are processed. Private (in terms of accessibility) contract function Process first verifies that the spend encoded in a transaction tx is valid before processing it. Validity checking amounts to checking that if $\mathtt{tx.nf} \neq \perp$ then tx.rt must have been a historical Merkle root and that the Action circuit proof is valid. Processing of a tx simply adds the new note commitment ncm to the Merkle tree MT and the spent note nullifier nf to NS.

The Balance circuit. "Balance" refers to that two transactions tx and tx′ declare the exact opposite value changes of the same asset type. In Zcash Orchard, balance is checked by utilizing the additive homomorphism of the commitment scheme. We use a ZK circuit to ensure balance here. We note that we could simplify the balance circuit by utilizing additionally another binding commitment scheme instead of FHE.Enc since full homomorphism is not required. But we opt to present the concept in an expressive manner without considering constant factor efficiency differences.

Public type declaration. We have chosen to hide the asset types by default for transactions to enable multi-asset confidential transfers. However, it shall be convenient for our later applications for transactions to (optionally) publicly declare the asset type. This is done via fields tx.typ and $\mathtt{tx}.r_{\mathtt{typ}}$. We require the type declaration to be correct, meaning that $\mathtt{tx.et} = \mathsf{FHE.Enc}_{\mathsf{pk}}(\mathtt{tx.typ}; \mathtt{tx}.r_{\mathtt{typ}})$—a check that is checked implicitly for all incoming transactions.

Relations with Decentralized Anonymous Payments (DAPs). A closely related primitive, first formalized in the work of Zerocash [8] is called Decentralized Anonymous Payments (DAPs). A DAP allows anonymous and confidential transactions modifying account balances in a verifiable manner. A ledger maintains a state that encodes the balances of users. To transact, users submit a transaction declaring certain properties, which is then processed by the ledger. For us, a transaction tx can declare any amounts of tokens minted (created) or burnt (destroyed), which the corresponding value declared via tx.ev. With the Balance circuit, our construction do realize the properties of DAPs but we do not formalize this here since it is not the focus of our work and due to syntactical differences.

```
Contract Token
State variables:
1  public MT // Merkle tree of note commitments
2  public NS // Set of spent note nullifiers
Priv Func Process(tx):
3  Assert (tx.nf = ⊥ or tx.nf ∉ NS)
4  Assert (tx.rt ∈ MT.RTS and Action.verify(tx.rt, tx; tx.π))
5  MT.add(tx.ncm) ; NS.add(tx.nf)
ZK Circuit Action(tx; sk, actionInfo):
6  (path, oldNote, newNote, r_val, r_typ) ← actionInfo
7  If tx.nf ≠ ⊥ then // Action spends an old note
8      ncm ← NoteCommit(oldNote)
9      Assert (tx.rt[path] = ncm)
10     Assert (oldNote.pk = Kg(sk))
11     Assert (tx.nf = DeriveNullifier_sk(oldNote) = newNote.ρ)
12     Assert (newNote.typ = oldNote.typ))
13     val ← oldNote.val − newNote.val
14 Else tx.nf = ⊥ then // Action only creates a new note
15     val = newNote.val
16 Assert (tx.ncm = NoteCommit(newNote))
17 Assert (tx.ev = FHE.Enc_pk(val; r_val))
18 Assert (tx.et = FHE.Enc_pk(newNote.typ; r_typ))
User Func CreateTx(sk, actionInfo):
19 (path, oldNote, newNote, r_val, r_typ) ← actionInfo
20 tx.rt ← MT.rt ; tx.ncm ← NoteCommit(newNote)
21 tx.nf ← DeriveNullifier_sk(oldNote)
22 tx.ev ← FHE.Enc(pk, oldNote.val − newNote.val; r_val)
23 tx.et ← FHE.Enc(pk, newNote.typ; r_typ)
24 tx.π ←$ Action.prove(tx; sk, path, oldNote, newNote, r_val, r_typ)
25 Return tx
ZK Circuit Balance(tx, tx′; actionInfo, actionInfo′):
26 (·, oldNote, newNote, r_val, r_typ) ← actionInfo
27 (..., r′_val, r′_typ) ← actionInfo′
28 typ ← oldNote.typ ; val ← oldNote.val − newNote.val
29 Assert (tx.ev = FHE.Enc(val; r_val))
30 Assert (tx′.ev = FHE.Enc(−val; r′_val))
31 Assert (tx.et = FHE.Enc(typ; r_typ))
32 Assert (tx′.et = FHE.Enc(typ; r′_typ))
User func CreateBalanceTx(sk, actionInfo, actionInfo′):
33 tx ← CreateTx(sk, actionInfo)
34 tx′ ← CreateTx(sk, actionInfo′)
35 π ← Balance.prove(tx, tx′; actionInfo, actionInfo′)
36 Return tx, tx′, π
```

Figure 2: Contract design for multi-asset shielded pool.

5.2 Trading: constant function market makers

Constant function market makers (CFMMs) have seen wide adoption on computing and storage constrained smart contract platforms such as Ethereum. Applications such as Curve and Uniswap, which are special cases of CFMMs, manage billions worth of assets and daily trades. CFMMs allow traders to trade against assets of liquidity providers (LPs) according to a fixed, simple rule. Mathematically, the application state consists of three numbers $(x, y, z) \in (\mathbb{R}^+)^3$, with x, y representing the overall asset reserves and z representing the representing pool tokens. In absence of fees, a trade of $(\delta_x, \delta_y) \in \mathbb{R}^2$ is valid if and only if

$$\phi(x + \delta_x, y + \delta_y) \geq \phi(x, y)\ , \tag{1}$$

where ϕ is a function called the *trading function*. For simplicity, we focus on the widely studied and adopted constant-product function in this work, i.e. $\phi(x, y) = xy$. In applications, reserve and trading amounts are typically represented as unsigned or signed 64-bit integers interpreted as fixed-point rational numbers.

Trade privacy in CFMMs. Two main privacy metrics of CFMMs are anonymity and trade privacy [25]. Roughly, A CFMM has anonymity if observers cannot infer the identity of the traders and it has trade privacy if trade amounts are not revealed. On one hand, it is known how to protect the anonymity of the traders [23, 24] using zero-knowledge proofs. On the other hand, it is known that trade privacy *cannot* be achieved if exact spot prices are released for each trade [2]. There are solutions of using ElGamal to net trades in a batch before executing [48]. Here, we describe how trades can be executed *sequentially* with programmable price release, as an application in the Pesca framework. Together with random permutation of batched trades, we can provably achieve differential privacy of individual execution prices [22].

FHE trade circuit. We first describe the key ingredient in our construction—the FHE trade circuit. We utilize the threshold FHE functionality to decide the validity of an incoming trade, which is encoded in a bit b (with 1 being valid and 0 otherwise) that is decrypted asynchronously via threshold decryption.

```
Contract CFMM extends Token, FHEBase
State variables:
1  public ex, ey, ez // Encrypted balances of assets and pool tokens
2  public typ_x, typ_y // asset types of the pool
FHE Circuit Trade((x, y), (dx, dy)):
3  x′ ← x + dx ; y′ ← y + dy
4  If (x′y′ ≥ xy) then Return ((x′, y′), 1)
5  Return ((x, y), 0)
Pub Func Trade(tx_fund, tx_refund, π, tx_payout):
6  Assert ({tx_fund.typ, tx_payout.typ} = {this.typ_x, this.typ_y})
7  Assert Balance.verify(tx_fund, tx_refund; π)
8  Assert (tx_refund.nf = tx_payout.nf = ⊥) ; this.Process(tx_fund)
9  ((ex, ey), eb) ← Trade.Eval((ex, ey), (tx_fund.ev, tx_payout.ev))
10 Async b ← ThDec(eb):
11     If b = 1 then this.Process(tx_payout)
12     Else this.Process(tx_refund)
User Func CreateTrade(sk, fundInfo, refundInfo, payoutInfo):
13 tx_fund, tx_refund, π ← CreateBalanceTx(sk, fundInfo, refundInfo)
14 tx_refund ← CreateTx(sk, payoutInfo)
15 Return tx_fund, tx_refund, π, tx_payout
```

Figure 3: Contract design for privacy-preserving CFMM.

Contract function Trade. To interface with the FHE Trade circuit, three action transactions are required: $\mathtt{tx}_{\mathtt{fund}}$, $\mathtt{tx}_{\mathtt{refund}}$ and $\mathtt{tx}_{\mathtt{payout}}$. Transactions $\mathtt{tx}_{\mathtt{fund}}$ and $\mathtt{tx}_{\mathtt{refund}}$ should balance, meaning that they spend and refunds the exact same amount. Transactions $\mathtt{tx}_{\mathtt{fund}}$ and $\mathtt{tx}_{\mathtt{payout}}$, if processed together, reflect that the trade is *valid*. Transactions $\mathtt{tx}_{\mathtt{fund}}$ and $\mathtt{tx}_{\mathtt{refund}}$, if processed together, reflect that the trade is *invalid*. Exactly which two transactions get processed is determined from the output bit of the FHE Trade circuit.

Price release and liquidity provision. To enable trades, the spot price of the market must be periodically released. We encapsulate

this in a contract function and FHE circuit ReleasePrice given in Figure 4. When invoked, the function simply releases the amount of normalized balance per pool token of either asset. We assume that there are external mechanisms, e.g. timers, that periodically invoke the price release mechanism so that users and arbitrageurs can trade against the CFMM.

Contract CFMM extends Token, FHEBase

FHE Circuit ReleasePrice(x, y, z):
1 Return $(x/z, y/z)$

Priv Func ReleasePrice():
2 eout ← ReleasePrice.Eval(ex, ey, ez)
3 Async (x, y) ← ThDec(eout):
4 Return x, y

Priv Func Setup(ex, ey, ez):
5 this.InitFHEState(ex, ey, ez)

FHE Circuit Enter($(x, y, z), (dx, dy, dz)$):
6 If $(x \cdot dy) = (y \cdot dx)$ and $(-z \cdot dx) = (x \cdot dz)$ then
7 Return $((x + dx, y + dy, z + dz), 1)$
8 Return $((x, y, z), 0)$

Pub Func Enter($\mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}, \pi, \mathtt{tx}'_{\mathsf{fund}}, \mathtt{tx}'_{\mathsf{refund}}, \pi', \mathtt{tx}_{\mathsf{payout}}$):
9 Assert ($\mathtt{tx}_{\mathsf{fund}}$.typ = this.$\mathsf{typ}_x$ and $\mathtt{tx}'_{\mathsf{fund}}$.typ = this.$\mathsf{typ}_y$)
10 Assert ($\mathtt{tx}_{\mathsf{payout}}$.typ = this)
11 Assert Balance.verify($\mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}; \pi$)
12 Assert Balance.verify($\mathtt{tx}'_{\mathsf{fund}}, \mathtt{tx}'_{\mathsf{refund}}; \pi'$)
13 Assert ($\mathtt{tx}_{\mathsf{refund}}$.nf = $\mathtt{tx}'_{\mathsf{refund}}$.nf = $\mathtt{tx}_{\mathsf{payout}}$.nf = $\perp$)
14 this.Process($\mathtt{tx}_{\mathsf{fund}}$) ; this.Process($\mathtt{tx}'_{\mathsf{fund}}$)
15 (est, eb) ← Enter.Eval(est, ($\mathtt{tx}_{\mathsf{fund}}$.ev, $\mathtt{tx}'_{\mathsf{fund}}$.ev, $\mathtt{tx}_{\mathsf{payout}}$.ev))
16 Async b ← ThDec(eb):
17 If $b = 1$ then this.Process($\mathtt{tx}_{\mathsf{payout}}$)
18 Else this.Process($\mathtt{tx}_{\mathsf{refund}}$) ; this.Process($\mathtt{tx}'_{\mathsf{refund}}$)

User Func CreateEnter(sk, assetAInfo, assetBInfo, payoutInfo):
19 $\mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}, \pi$ ← CreateBalanceTx(sk, assetAInfo)
20 $\mathtt{tx}'_{\mathsf{fund}}, \mathtt{tx}'_{\mathsf{refund}}, \pi'$ ← CreateBalanceTx(sk, assetBInfo)
21 $\mathtt{tx}_{\mathsf{payout}}$ ← CreateTx(sk, payoutInfo)
22 Return $\mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}, \pi, \mathtt{tx}'_{\mathsf{fund}}, \mathtt{tx}'_{\mathsf{refund}}, \pi', \mathtt{tx}_{\mathsf{payout}}$

Figure 4: Contract design for a privacy-preserving CFMM, continued.

Providing liquidity. Besides trading, a CFMM requires the ability for users to deposit liquidity. The design here follows the same outline as the case for trade. The FHE circuit Enter takes an additional input z, which encodes the number of pool tokens minted or burnt. We omit further details here as the pseudocode given in Figure 4 is self-explanatory. We remark that for users to be able to construct valid deposits. The user must know the exact spot trading price, which is only possible after a price release. To make this possible, the CFMM contract can block trades at a predetermined time for a set amount of time and release the up-to-date spot price for liquidity provision.

5.3 Sealed-bid auctions

Overview of contract. The on-chain application act as the auctioneer, accepting encrypted (to chain FHE public key pk) bids from users. The contract is capable of keeping track of arbitrary number of concurrently running auctions, each with some index i. For each auction i, the application state est$[i]$ consists of a tuple of (unsigned) integers (p, j), each initialized to 0. Integer p encodes

Contract FPA extends Token, FHEBase

State variables:
1 public est // List of running auction states
2 public item // List of auction items
3 public refund // List of bid refunds for each auction item
4 public payout // List of payout transactions for each auction item

Pub Func SetupAuction($\mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}, \pi$, est):
5 Balance.verify($\mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}; \pi$) ; this.Process($\mathtt{tx}_{\mathsf{fund}}$)
6 $i \leftarrow$ |this.est| ; this.item$[i]$ ← $\mathtt{tx}_{\mathsf{fund}}$
7 this.payout$[i]$ ← [$\mathtt{tx}_{\mathsf{refund}}$]
8 this.est$[i]$ ← est ; Return i

FHE Circuit Accum$[j](p, k)$:
9 If $(q > p)$ then return (q, j) Else return (p, k)

Pub Func ProcessBid($i, \mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}, \pi_{\mathsf{bid}}, \mathtt{tx}_{\mathsf{payout}}, \pi_{\mathsf{payout}}$):
10 Assert ($\mathtt{tx}_{\mathsf{fund}}$.typ = typ_0) // Some fixed currency type
11 Assert Balance($\mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}; \pi_{\mathsf{fund}}$)
12 Assert Balance(this.item$[i]$, $\mathtt{tx}_{\mathsf{payout}}; \pi_{\mathsf{payout}}$)
13 Assert ($\mathtt{tx}_{\mathsf{refund}}$.nf = $\mathtt{tx}_{\mathsf{payout}}$.nf = $\perp$) ; this.Process($\mathtt{tx}_{\mathsf{fund}}$)
14 this.refund$[i]$.push($\mathtt{tx}_{\mathsf{refund}}$) ; this.payout$[i]$.push($\mathtt{tx}_{\mathsf{payout}}$)
15 est$[i]$ ← Accum[|this.refund$[i]$|].Eval(this.est$[i]$, $\mathtt{tx}_{\mathsf{fund}}$.ev)

User Func CreateBid(i, bidInfo, payoutInfo):
16 $\mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}, \pi_{\mathsf{fund}}$ ← CreateBalanceTx(sk, bidInfo)
17 $\mathtt{tx}_{\mathsf{payout}}, \cdot, \pi_{\mathsf{payout}}$ ← CreateBalanceTx(sk, payoutInfo)
18 Return $\mathtt{tx}_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{refund}}, \pi_{\mathsf{fund}}, \mathtt{tx}_{\mathsf{payout}}, \pi_{\mathsf{payout}}$

Pub Func FinalizeAuction(i):
19 $(\cdot, \mathtt{ej})$ ← this.est$[i]$
20 Async j ← ThDec(ej):
21 this.Process(this.payout$[i][j]$)
22 For $0 < k \neq j$ do this.Process(this.refund$[i][k]$)

Figure 5: Contract design for privacy-preserving sealed-bid first-price auction.

the current max bid and j denotes the index of the max bidder. Here, for simplicity, we assume that all bids are given in some fixed token type typ_0.

In a sealed-bid auction, each bidder commits to a price p_i which is not revealed to any other user. The winner is defined as the party j with the largest bid p_j, and the realized price is typically some function f of the first and second price.

We are interested in sealed-bid auctions with *no user-to-user interaction* and *minimal information leakage*. In particular, we want the bids of users to *stay sealed even after the conclusion of the auction*. This is potentially important for multi-round auctions, as asset reserves of bidders are not revealed.

Setting up an auction. To set up an auction, the item hold simply constructs a funding transaction $\mathtt{tx}_{\mathsf{fund}}$ spending the auction item, a refund transaction $\mathtt{tx}_{\mathsf{refund}}$ which refunds the auction item (default outcome of auction), a balance proof π, an initial auction state est, encoding the minimum bid price, and a state initialization proof π'. We assume that the information about the auction item, namely its type, is released in some channel so that interested users can construct valid bidding transactions.

Accumulate FHE circuit. We design a minimal FHE circuit to support bid processing. Note that the Accum circuit is parameterized by an integer i denoting the incoming bid index. The circuit simply compares the current winning price to an incoming bid and updates the winner index accordingly.

Bid processing. A bid consists of a funding transaction $\texttt{tx}_{\texttt{fund}}$ containing the bid, $\texttt{tx}_{\texttt{refund}}$ to refund the bid amount and associated balance proof π, a payout transaction $\texttt{tx}_{\texttt{payout}}$ claiming the auction item and associated balance proof π'. Balance proof π ensures that $\texttt{tx}_{\texttt{refund}}$ refunds exactly the amount in the bid transaction $\texttt{tx}_{\texttt{fund}}$. Balance proof π' ensures that $\texttt{tx}_{\texttt{payout}}$ claims exactly the item being auctioned. The contract processes the funding transaction and simply pushes $\texttt{tx}_{\texttt{refund}}$ and $\texttt{tx}_{\texttt{payout}}$ to a list. It then accumulates the incoming bid using the Accum FHE circuit with the parameter set to the current bid index (which is equal to the length of the refund transaction array $|\texttt{this.refund}[i]|$), which will update the current max price and winner index if the incoming bid price is higher than previous max price.

Bid generation. To create a bid, a user simply generates a funding transaction $\texttt{tx}_{\texttt{fund}}$, associated refund transaction $\texttt{tx}_{\texttt{refund}}$, and balance proof π.

Finalizing an auction. To finalize an auction with index i, the contract first threshold decrypts the part of the state containing the winner index j. It will process refund transactions of all losing bids and the payout transaction of the winning bidder. Note that since we have previously ensured balance, no additional balance checks are required in this step.

Information leakage analysis. Note that only the `FinalizeAuction` function release information about the confidential state est, and only the winner index is revealed. This means that each bidder can only learn whether their bid was a winning bid or not.

Efficiency consideration. Our FHE circuits require only a couple arithmetic operations on fixed-point numbers. Unfortunately, the most efficient bootstrapped FHE schemes, such as FHEW [30] and TFHE [21], work over binary values. We note that to implement n-bit arithmetic operations using binary gates we need to account for blowup factors of $O(n)$ for addition and mux, $O(n^2)$ for multiplication and division.

6 CONCLUSION

In this work, we propose Pesca, a smart contract architecture where privacy-preserving applications can be built with its security properties solely reliant on the Byzantine assumptions of consensus nodes.

We formalize the notion of state machine augmented with threshold release protocols and study how existing BFT protocols can be adopted to replicate them. We instantiate a blockchain supporting programmable threshold decryption using threshold FHE amongst other widely-studied building blocks. We propose a smart contract architecture on top of such a blockchain where applications can be engineered expressively. We provide two example applications realizing CFMMs and FPSBAs, both with maximal privacy guarantees.

We identify the performance of threshold FHE schemes support Shamir secret keys as a key performance bottleneck. We think that with more research efforts on such types of FHE schemes, as well as more engineering efforts on programming and compiler frameworks for FHE and universal zk-SNARKs, the Pesca blueprint can become truly viable to support real-world applications.

REFERENCES

[1] Guillermo Angeris and Tarun Chitra. 2020. Improved price oracles: Constant function market makers. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. IEEE, 80–91.

[2] Guillermo Angeris, Alex Evans, and Tarun Chitra. 2021. A note on privacy in constant function market makers. *arXiv preprint arXiv:2103.01193* (2021).

[3] Circom authors. 2022. Circom Circuit Compiler. https://docs.circom.io/. Accessed May. 2022.

[4] Web3.js authors. 2022. web3.js – Ethereum JavaScript API. https://web3js.readthedocs.io/en/v1.7.3/. Accessed May. 2022.

[5] Zokrates authors. 2022. Zokrates. https://zokrates.github.io/. Accessed May. 2022.

[6] Aritra Banerjee, Michael Clear, and Hitesh Tewari. 2021. zkhawk: Practical private smart contracts from mpc-based hawk. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 245–248.

[7] Mihir Bellare. 2020. Lectures on NIZKs. (2020). https://cseweb.ucsd.edu/~mihir/cse208-Wi20/main.pdf.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 459–474. https://doi.org/10.1109/SP.2014.36

[9] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. 2020. Can a Public Blockchain Keep a Secret?. In *TCC 2020, Part I (LNCS, Vol. 12550)*, Rafael Pass and Krzysztof Pietrzak (Eds.). Springer, Heidelberg, 260–290. https://doi.org/10.1007/978-3-030-64375-1_10

[10] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. 2017. Threshold Cryptosystems From Threshold Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2017/956. https://eprint.iacr.org/2017/956.

[11] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. 2018. Threshold Cryptosystems from Threshold Fully Homomorphic Encryption. In *CRYPTO 2018, Part I (LNCS, Vol. 10991)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, Heidelberg, 565–596. https://doi.org/10.1007/978-3-319-96884-1_19

[12] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. ZEXE: Enabling Decentralized Private Computation. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 947–964. https://doi.org/10.1109/SP40000.2020.00050

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012*, Shafi Goldwasser (Ed.). ACM, 309–325. https://doi.org/10.1145/2090236.2090262

[14] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938* (2018).

[15] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. 2020. Zether: Towards Privacy in a Smart Contract World. In *FC 2020 (LNCS, Vol. 12059)*, Joseph Bonneau and Nadia Heninger (Eds.). Springer, Heidelberg, 423–443. https://doi.org/10.1007/978-3-030-51280-4_23

[16] Vitalik Buterin and Virgil Griffith. 2017. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437* (2017).

[17] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. 2021. Internet Computer Consensus. Cryptology ePrint Archive, Report 2021/632. https://eprint.iacr.org/2021/632.

[18] Benjamin Y Chan and Elaine Shi. 2020. Streamlet: Textbook Streamlined Blockchains. Cryptology ePrint Archive, Report 2020/088. https://eprint.iacr.org/2020/088.

[19] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 185–200.

[20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *EUROCRYPT 2020, Part I (LNCS, Vol. 12105)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 738–768. https://doi.org/10.1007/978-3-030-45721-1_26

[21] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33, 1 (Jan. 2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[22] Tarun Chitra, Guillermo Angeris, and Alex Evans. 2021. Differential Privacy in Constant Function Market Makers. Cryptology ePrint Archive, Report 2021/1101. https://eprint.iacr.org/2021/1101.

[23] Shumo Chu, Yu Xia, and Zhenfei Zhang. 2021. Manta: a Plug and Play Private DeFi Stack. Cryptology ePrint Archive, Report 2021/743. https://eprint.iacr.org/2021/743.

[24] Wei Dai. 2021. Flexible Anonymous Transactions (FLAX): Towards Privacy-Preserving and Composable Decentralized Finance. Cryptology ePrint Archive,

Report 2021/1249. https://eprint.iacr.org/2021/1249.
[25] Wei Dai. 2022. Navigating Privacy on Public Blockchains. https://wdai.us/posts/navigating-privacy/. Accessed March. 2022.
[26] Sourav Das, Tom Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. 2021. Practical Asynchronous Distributed Key Generation. Cryptology ePrint Archive, Report 2021/1591. https://eprint.iacr.org/2021/1591.
[27] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. 2001. Robust Non-interactive Zero Knowledge. In *CRYPTO 2001 (LNCS, Vol. 2139)*, Joe Kilian (Ed.). Springer, Heidelberg, 566–598. https://doi.org/10.1007/3-540-44647-8_33
[28] Yvo Desmedt and Yair Frankel. 1989. Threshold cryptosystems. In *Conference on the Theory and Application of Cryptology*. Springer, 307–315.
[29] Benjamin E. Diamond. 2021. Many-out-of-Many Proofs and Applications to Anonymous Zether. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1800–1817. https://doi.org/10.1109/SP40001.2021.00026
[30] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *EUROCRYPT 2015, Part I (LNCS, Vol. 9056)*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer, Heidelberg, 617–640. https://doi.org/10.1007/978-3-662-46800-5_24
[31] Taher ElGamal. 1985. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory* 31 (1985), 469–472.
[32] Andreas Erwig, Sebastian Faust, and Siavash Riahi. 2021. Large-Scale Non-Interactive Threshold Cryptosystems Through Anonymity. Cryptology ePrint Archive, Report 2021/1290. https://eprint.iacr.org/2021/1290.
[33] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. 2019. Quisquis: A New Design for Anonymous Cryptocurrencies. In *ASIACRYPT 2019, Part I (LNCS, Vol. 11921)*, Steven D. Galbraith and Shiho Moriai (Eds.). Springer, Heidelberg, 649–678. https://doi.org/10.1007/978-3-030-34578-5_23
[34] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953. https://eprint.iacr.org/2019/953.
[35] Craig Gentry, Shai Halevi, Bernardo Magri, Jesper Buus Nielsen, and Sophia Yakoubov. 2020. Random-index PIR with Applications to Large-Scale Secure MPC. Cryptology ePrint Archive, Report 2020/1248. https://eprint.iacr.org/2020/1248.
[36] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *CRYPTO 2013, Part I (LNCS, Vol. 8042)*, Ran Canetti and Juan A. Garay (Eds.). Springer, Heidelberg, 75–92. https://doi.org/10.1007/978-3-642-40041-4_5
[37] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. 2020. Storing and Retrieving Secrets on a Blockchain. Cryptology ePrint Archive, Report 2020/504. https://eprint.iacr.org/2020/504.
[38] Jens Groth. 2006. Simulation-Sound NIZK Proofs for a Practical Language and Constant Size Group Signatures. In *ASIACRYPT 2006 (LNCS, Vol. 4284)*, Xuejia Lai and Kefei Chen (Eds.). Springer, Heidelberg, 444–459. https://doi.org/10.1007/11935230_29
[39] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT 2016, Part II (LNCS, Vol. 9666)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, Heidelberg, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11
[40] Jens Groth. 2021. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Report 2021/339. https://eprint.iacr.org/2021/339.
[41] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2022. Zcash Protocol Specification. https://zips.z.cash/protocol/protocol.pdf. Accessed April. 2022.
[42] Markulf Kohlweiss and Michał Zając. 2021. On Simulation-Extractability of Universal zkSNARKs. Cryptology ePrint Archive, Report 2021/511. https://eprint.iacr.org/2021/511.
[43] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 839–858. https://doi.org/10.1109/SP.2016.55
[44] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 2111–2128. https://doi.org/10.1145/3319535.3339817
[45] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. 2019. CHURP: Dynamic-Committee Proactive Secret Sharing. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 2369–2386. https://doi.org/10.1145/3319535.3363203
[46] Anoma Network. 2022. Ferveo: A synchronous Distributed Key Generation protocol for front-running protection on public blockchains. https://github.com/anoma/ferveo. Accessed April. 2022.
[47] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT'99 (LNCS, Vol. 1592)*, Jacques Stern (Ed.). Springer, Heidelberg, 223–238. https://doi.org/10.1007/3-540-48910-X_16
[48] Penumbra. 2022. Penumbra Spec: Sealed-Bid Batch Auctions. https://protocol.penumbra.zone/main/zswap/auction.html. Accessed March. 2022.
[49] Amit Sahai. 1999. Non-Malleable Non-Interactive Zero Knowledge and Adaptive Chosen-Ciphertext Security. In *40th FOCS*. IEEE Computer Society Press, 543–553. https://doi.org/10.1109/SFFCS.1999.814628
[50] Adi Shamir. 1979. How to Share a Secret. *Communications of the Association for Computing Machinery* 22, 11 (Nov. 1979), 612–613.
[51] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *38th ACM PODC*, Peter Robinson and Faith Ellen (Eds.). ACM, 347–356. https://doi.org/10.1145/3293611.3331591

A SIZE OF MODULUS FOR THRESHOLD FHE

Fix a finite field $\mathbb{F}_q$, Shamir t-out-of-n secret sharing takes input a master secret $\mathsf{sk} \in \mathbb{F}_q$ and outputs n secret shares $\mathsf{sk}_1 = P(1), \ldots, \mathsf{sk}_n = P(n)$ where $P \in \mathbb{F}_q[X]$ is a random polynomial of degree t with the restriction that $P(0) = \mathsf{sk}$. Given any subset S of key shares with $|S| = t$, a reconstruction of any other share $P(j)$, including the master secret with $P(0)$, can be obtained as a linear combination of shares from S:

$$P(j) = \sum_{i \in S} \lambda_{i,j}^{S} \cdot P(i) \,, \tag{2}$$

where $\lambda_{i,j}^{S}$ are the Lagrange coefficients. Threshold decryption is obtained by observing that reconstruction and decryption are both linear and "commute", i.e. reconstruction can be done over partial decryptions.

Unlike ElGamal or Paillier, encryption schemes based on the learning-with-error (LWE) problem do not directly support threshold decryption naively as described above, as applying the Lagrange coefficients directly to LWE ciphertexts completely removes the information encoded on the message encrypted. The workaround proposed by [10, 11] involves computing $\lambda_{i,j}^{S}$ as a rational number instead of as a field element. During reconstruction, a large multiplier $(n!)^2$ is applied to both sides of (2) to clear out the denominators. As a consequence, the modulus q needs to be "blown up" by a factor of $(n!)^3$ compared to that of non-threshold schemes. This method applies to a special class of FHE schemes [10, Definition 3.9] covering BGV [13] GSW [36] and FHEW [30] families.