SoK: Programmable Privacy in Distributed Systems

Daniel Benarroch* daniel@inversed.tech Bryan Gillespie* bryan@inversed.tech

Ying Tong Lai[†] yingtong@geometry.dev Andrew Miller[‡] soc1024@illinois.edu

June 17, 2024

Abstract

This Systematization of Knowledge conducts a survey of contemporary distributed blockchain protocols, with the aim of identifying cryptographic and design techniques which practically enable both expressive programmability and user data confidentiality. To facilitate a framing which supports the comparison of concretely very different protocols, we define an epoch-based computational model in the form of a flexible UC-style ideal functionality which divides the operation of privacy-preserving networks into three phases: Independent, Mediated, and Global computation. Our analysis of protocols focuses in particular on features of the Mediated computation phase, which provides the facility to execute nontrivial program logic on private inputs from multiple users. Specifically, we compare implementations in different protocols for private limit order auctions, which we find to be a representative application which is common and relatively simple, but which exhibits adversarial dynamics which demonstrate the capabilities of a non-trivial Mediated computation mechanism. In our analysis, we identify four protocols representative of different high-level approaches used to implement Mediated computations. We compare protocols according to the degree and flexibility of programmability, the privacy properties achieved, and the security assumptions required for correct operation. We conclude by offering recommendations and best practices for future programmable privacy designs.

1 Introduction

Public distributed blockchains provide a powerful primitive for achieving trustless and permissionless consensus on the outcome of shared computational tasks over heterogeneous user inputs. Since early in the evolution of blockchain protocols, there has been broad interest in the development of privacy-enabling techniques allowing transactions on public ledgers to enjoy privacy and confidentiality characteristics similar to those offered by traditional centralized financial services. Early privacy-enabled protocols such as Monero [42] and Zcash [22] demonstrated the viability of this agenda by applying new cryptographic schemes to provide various degrees of confidentiality for transactions witnessed on-chain, albeit with limited functionality.

With the advent of programmable smart-contract chains, the question of user data privacy took on a more urgent tone, as programmability opened up a wide array of applications which drove accelerating adoption, but also recorded an unprecedented amount of sensitive user data in public blockchain records. As practitioners have grappled with this problem, the development of blockchain protocols supporting user data privacy on top of a programmable execution engine has resulted in a tumult of new techniques and designs which exhibit differing degrees of programmability and privacy under varying security assumptions.

In this Systematization of Knowledge, we conduct a survey of contemporary protocols which demonstrate effective *programmable privacy* for distributed blockchains, with the aim of identifying and comparing cryptographic techniques and design patterns which provide user data privacy for extensible applications. Our

^{*}Inversed Tech

[†]Geometry Research

[‡]University of Illinois, Urbana-Champaign

analysis will consider a cohort of four privacy-enhanced blockchain protocols, which were selected with the aim of discussing a *representative* sample of mature designs using different techniques and making different trade-offs in the programmable privacy design space. In particular, our selection of protocols is not meant to be comprehensive, but rather, to faithfully represent the major families of approaches being applied today. The full list of protocols is provided in Section 4.

Recent progress in this space can be explained by the transition to practice of a number of advanced cryptography primitives, such as ZKP, MPC, FHE, and TEEs. Not only have implementations become more performant, but each of these is also associated with general purpose programming frameworks and compilers which increase accessibility for a wider pool of protocol and application developers. We review some of the main ideas underlying these primitives in Section 2.

Though it seems like there could be endless ways to fit these tools together, there turn out to be many similarities between practical designs. We propose a lightweight framework for describing the architectures of these systems with a three phase computational model for processing sensitive user inputs in batches. The first "Independent computation" phase is a local computation by individuals using a combination of public and *personal* private data. Zero-knowledge proofs and shielded pools are most useful here for user privacy, since users can generate proofs which reference data without revealing that data to a verifier. Second, for functionalities that require aggregation, a privacy-preserving "Mediated computation" phase is executed by protocol operators on the combined user inputs of the batch. To ensure privacy in this phase, it is commonly necessary to incorporate an extra trust assumption in the security model, such as requiring that a quorum of key holders do not collude to decrypt more than they should, or that some secure hardware enclaves operating on confidential data have not been compromised. Finally, in the "Global computation" phase, the output of the batch is registered and takes effect on the blockchain. We describe this model in more detail in Section 3, and in particular represent it semi-formally as a UC style ideal functionality in Table 1.

In Section 4, using this computational model as a common framing, we compare the selected protocols in terms of their degree of programmability, their privacy properties, and their underlying security assumptions. We particularly consider the characteristics of the Mediated computation phase, which we find to be the part of a design which provides the clearest distinction between different protocols, and which seems to be a common focus of many of the innovations seen recently in the space. In contrast, we do not consider as part of our analysis differences in development tooling and front-end programming languages implemented by protocols. These features are inarguably important for usability and adoption, but we consider their design to be essentially orthogonal to the underlying cryptographic and computational layer studied in this work. We briefly discuss the relation between protocol designs and the role of domain-specific languages in abstracting cryptographic complexity in Section 5.2.1.

To further direct our analysis, we will restrict our attention to a particular family of applications, *limit* order auctions. Briefly, a limit order mechanism allows users to express orders with reserve prices, and results in a trade allocation that fills orders at the listed prices or better. This mechanism is expressive enough to include as special cases both automated market makers like UniswapV2 that are widespread in DeFi, and the double order books commonly found on centralized exchanges. Limit order auctions can handily be implemented in digital assets and smart contracts using "intents", by encoding price constraints within the trade authorization logic. In mechanism design terms, we can say that a limit order or intent system is built around the "individual rationality" property, which means that by participating in the mechanism, users are always at least as well off as if they had abstained.

To elaborate on the motivation for this choice of application, we have found that the limit order auction is an ideal setting to study the techniques being used to enable general programmable privacy for several reasons. Foremost among these is *economic privacy*. The economic context of the limit order auction, involving exchanges of value between non-trusting and adversarial counterparties, means that computation over private inputs executed verifiably and synchronously is important for proper functioning of the system. If a trading counterparty can see the full details of a party's trading strategy (e.g. past and pending orders and trades) then this reveals information that can give the counterparty an advantage in future interactions. Furthermore, if a decentralized network operator has visibility or direct control over the selection of trades which are executed, then they can give an advantage to themselves or partners by prioritizing their own orders over those of other market participants, in particular enabling various forms of Maximal Extractable Value (MEV) in the system. In more complicated designs which incorporate public or automatic market makers, or provide intent-solving functionality, synchronous processing over many users' private inputs may be required to ensure proper market functioning, despite the distributed and asynchronous characteristics of the underlying execution engine.

As a second motivation for the focus on limit order auctions, the application is both simple enough and common enough that it provides a uniform design setting in which the characteristics of most decentralized protocols aiming to support some combination of programmability and privacy can be compared. This allows for something close to an apples-to-apples comparison between very different protocols—for characteristics such as privacy properties, security assumptions, and protocol failure modes.

In Section 5, we present some concluding remarks on the practical design of privacy oriented protocols that follow from our analysis. We discuss how the framing of our three phase computational model suggests a specifically structured approach for protocol designs, and argue in particular that it is feasible with currently mature technologies to produce a design which achieves general programmability while providing strong user data privacy under cryptographic honest majority assumptions.

1.1 Related work

In [38], the authors motivate programmable privacy in an economic setting, examining the effects of preexecution privacy on (1) user behavior, (2) validator behavior, and (3) DeFi markets. It further imagines a general framework in which arbitrary properties can be proven over private pending transactions. They state that comparison of the underlying cryptographic primitives and schemes is outside of the scope of their paper—this sets one motivation for the exploration of an orthogonal design space, systematized in our work.

The survey [2] discusses a wide range of privacy-preserving blockchain design paradigms according to three main functionalities: (1) private payments; (2) private arbitrary computation; and (3) function privacy that hides the computation itself. In this SoK, we present a unifying framework to reason about all three types of functionalities; we also provide more in-depth analyses of a subset of the chosen protocols. Its main point of overlap with this work is in the categorisation of (2) into a) the "on-chain" category, or homomorphic encryption-based approach, and b) the "off-chain" category, which uses either zero-knowledge proofs or trusted managers/hardware. This work covers the same techniques, but categorises them according to computational phases instead of whether they occur on-chain or off-chain.

A more recent SoK [5] on privacy-enhancing technologies (PETs) in finance takes an application-centric approach, identifying four classes of financial applications that require private inputs from multiple parties— Identity Management, KYC & AML; Markets & Settlement; Legal; and Digital Asset Custody—and enumerates representative examples of PET solutions used in each. In contrast, this SoK centers its analysis on a unifying framework across multiple combinations of PETs in contemporary implementations, and uses a single application, limit order auctions, as an illustrative example.

2 Preliminaries and Definitions

In the following, we formulate background and definitions that we will use when discussing programmable privacy through the remainder of the survey. We clarify our meaning for the widely used phrase "programmable privacy", define notions of privacy and security which will support our analysis, and review common cryptographic primitives used in modern privacy protocols.

2.1 Programmable Privacy

Programmability and privacy in distributed protocols are terms that take on a rather wide variety of meanings in different contexts, so we will elaborate briefly on the meanings that we adopt in this work. We say that a protocol is *programmable* if it provides users with *expressive control* over application logic, either permissionlessly or with minimal coordination with other protocol participants. Two classes of programmability we single out in particular are:

• Application programmability — a user can provide program logic for extended functionality which can be called by other users and is verifiably executed on chain, usually defining a new common API for interaction

• *Transaction programmability* — a user can specify predefined control logic which is used as part of the overall execution of an algorithm in a *shared computational context*

Traditional smart contracts are the most common example of application programmability in the nonprivate setting, while flexibly parameterized transactions such as those in an intent-based trading architecture are an example of transaction programmability.

We say that a protocol operation provides *data privacy* if some information about the operation inputs is kept hidden from parties aside from the party initiating the operation, while still enabling correct and verifiable execution using such inputs. Providing privacy in a distributed and multiparty computational context is challenging for a variety of reasons: data with different privacy constraints must be acted on together; users whose private data is acted on are not usually online to participate directly in the computation; it is generally difficult to keep information about a computation private from an executing node; and distributed protocols of interest usually have to be resilient to quite strong adversarial threat models.

Definition (Programmable Privacy). A decentralized system is said to provide *programmable privacy* if the system supports *application* programmability allowing developers to specify data privacy characteristics for inputs of users calling the functionality.

Of note, we do not include support for *transaction* programmability in this definition, though the degree of transaction programmability may in cases be a measure of the suitability of an underlying execution environment for supporting application programmability and programmable privacy.

Some additional terminology will also be used to refer to specific common notions of privacy at various points. In general, "identities" in a distributed blockchain application should be thought of as being represented as some public/private keypairs allowing typical encryption, decryption, and validation procedures, while "protocol parties" are computational agents with memory, and ownership of some collection of concrete identity keypairs. Protocol operations can provide one or more of the following types of privacy, usually defined in terms of inputs to a user's interface for the operation.

- Anonymity a collection of one or more public keys associated with a protocol operation are not revealed to an adversary lacking knowledge of any private keys from the collection
- *Confidentiality* a collection of one or more data fields associated with a protocol operation are not revealed to an adversary lacking knowledge of any private keys from an authorized collection
- Unlinkability for related pieces of data (public key, commitment, etc.) associated with multiple protocol operations, the relation between the data is not revealed to an adversary observing more than one such operation who lacks knowledge of any private keys from an authorized collection
- Function privacy a function associated with an operation is not revealed to an adversary lacking knowledge of any private keys from an authorized collection

Formalizing these notions to the degree required by a fully rigorous security analysis is involved, and beyond the scope of this work, but we discuss in somewhat more detail in Section 2.3 how these definitions can be reasonably framed in the language of UC style ideal functionalities.

2.2 Security Models

A major component of the security provided by a distributed protocol is described by the *security model* and *trust assumptions* underlying the model's claimed properties. Secure computation relies on the *reduction* of desired properties to the assumptions provided by an underlying security model, which can come in different forms. The security model can play a significant role in how plausible it is (practically or in perception) for an adversary to overcome a privacy scheme. Some common features that are incorporated by security models include the following.

Cryptographic Security and Assumptions. Mathematical constructions used to ensure protocol security rely on the difficulty of solving certain computational problems to guarantee their effectiveness. Some problems such as hardness of discrete logarithms or preimage resistance of well-known cryptographic hash functions are called "standard assumptions" and typically have risk profiles with known parameters, and are usually acceptable as a foundation for production cryptosystems. More exotic cryptographic primitives may rely on other less studied computational problems, often called "non-standard assumptions". Such assumptions, though potentially valid, have underexplored risk profiles, and so their use as a foundation for protocol security in real-world applications protecting much of value would typically be discouraged.

Notably, the distinction between "standard" and "non-standard" assumptions is fuzzy and often debated even by skilled practitioners; as such, it is useful for protocol specifications to be precise about what cryptographic assumptions are required for their correct functioning.

Corruption Models and Economic Security. Protocol security often depends on the degree to which different participants are willing to act dishonestly. A protocol in which nobody is acting according to the rules has no ability to constrain the computational outputs, but it may be possible for a single honest participant in a protocol to still produce valid results due to careful design. In other cases, validity of protocol results may rely on a fixed proportion of participants to behave honestly, or at least only to behave dishonestly in constrained ways. We call such an assumption k of N security, where k is the number of participants out of N in total which are required to correctly follow a protocol. More generally, the description of which protocol participants are assumed to behave dishonestly, and in what ways, is called the "corruption model" of the protocol.

One approach to ensuring a required level of honest participation in a protocol is to tie economic incentives to honest participation. For instance, a protocol may be designed such that bad behavior is possible, but that once this behavior is detected, an amount of a network currency token which was escrowed in order to participate is taken away and distributed to the remaining honest participants. The desired properties of these "economic security" schemes rely on the economic incentives of its participants.

Hardware Security. A layer of protocol security can be provided by requiring the use of hardened secure computational modules implementing a *Trusted Execution Environment* (*TEE*). Such modules run computations within a secure enclave with specified, controlled interfaces for input and output, and provide hardware-enforced assurances of validity and privacy. Under the assumption that a TEE module is physically capable of resisting attacks from a determined adversary, then computations executed using a TEE can be treated as though they were run by a "trusted" computational entity which is assumed to produce correct results and leak no information beyond what is specified by the controlled interface.

The availability of a real-world instantiation of a trusted protocol participant can greatly simplify the design of secure protocols. However, given the prevalence of subtle side-channel attacks on real-world hard-ware, as well as instances where cryptographic secrets have been exfiltrated from TEE modules which were thought to be secure (see for instance [43]), there is some disagreement on the feasibility of producing TEE modules which truly are capable of resisting sophisticated attacks. Nevertheless, even if such concerns are warranted, hardware-based measures may still be valuable for contexts where security does not have a single point of failure (e.g. compromising a single TEE module breaks protocol security), but rather, depends only on a "significant portion" of participants making use of TEE modules as intended.

Application Security. A further aspect of protocol security which is important to consider, but which is broadly outside the scope of this work, is that the overall guarantees provided by a protocol which admits user programmable logic may be dependent on multiple layers of specifications. The underlying global protocol/consensus layer will generally provide some guarantees, while the design of individual applications may provide others. In some cases even the specifics of user input may play into what level of privacy is provided in a particular distributed environment. In our analysis, we will treat protocol level security and privacy as the primary focus, and we may make remarks when warranted about ways in which application or user level security may interact with the protocol level in subtle ways.

2.3 Ideal Functionalities

Throughout this survey we will model protocols using **ideal functionalities** in the style of the Universal Composability (UC) security framework. Such ideal functionalities can be understood as a specification of the correctness and privacy properties required from a protocol, described as an equivalent implementation of the task at hand which is provided by a single trusted third party with secure communication channels to all protocol participants.

Ideal functionalities are related to real implementations by way of the *simulation* paradigm of security: a real implementation securely realizes an ideal functionality if the view of an adversary \mathcal{A} interacting with the real implementation can be simulated by a related adversary \mathcal{S} (the "simulator") who instead interacts with the ideal protocol, in such a way that it is impossible for an efficient algorithm to distinguish between the real and simulated protocol views except with negligible advantage. As part of an ideal functionality, inputs may be accepted directly from the adversary, and some values may be "leaked" to represent information that is not assumed to remain private during protocol execution.

This idea of information leakage in an ideal functionality is particularly useful as a way to express legible privacy properties about a realizing implementation. For the privacy properties listed previously in Section 2.1, for instance, information "not being revealed to an adversary" can be restated as that information "not being present in the information leaked to the adversary". To be careful, for many distributed protocols with an ongoing process of execution, it is important to be clear about the time horizon and other relevant context in which such privacy properties are valid.

Beyond privacy considerations, the simulation based approach provides a rich language for expressing security modeling considerations, in particular by giving an adversary access to a special "backdoor" communication channel which allows them to communicate with protocol participants in a controlled way. For instance, in a security model allowing a fixed number of statically selected corrupt parties in a protocol, an adversary can send a special "corruption" message at the beginning of protocol execution to a selection of parties that they wish to corrupt to signal that their behavior should be controlled by the adversary.

In our discussions, we provide ideal functionalities as a way of expressing a general class of "simplified security definitions" for distributed protocols providing programmable privacy capabilities. These functionalities in particular provide a structured view of protocol operation which usefully emphasizes certain commonalities in the way that programmable privacy is achieved in practice. We do not give proofs that concrete protocols securely realize these ideal functionalities, as this would be beyond the desired scope of this work. For more information on ideal functionalities and simulation-based security, see for instance [11, 28, 27].

2.4 Cryptographic Techniques.

This paper presents a framework generic across low-level cryptographic techniques; these techniques are often composed to enforce protocol-level invariants on both individual and shared private state. For instance, zeroknowledge proofs (ZKPs) are used to ensure well-formation of inputs to interactive mediated computations, such as multi-party computation and homomorphic encryption; ZKPs can be collaboratively generated to attest to the correctness of a multi-party computation [33]; and most protocols relying on homomorphic encryption thresholdize the encryption key (reducing its security to k-out-of-n assumption). The choice of primitives is a consequence of design requirements on expressivity and security, and not a vector of categorization in and of itself. Here, we provide a high-level overview of commonly used primitives and several of their well-known implementations, with the goal of providing intuition on their usage in the protocols to follow.

Commitment Schemes. Commitment schemes, first formalized in [10], enable the construction of a commitment C := Commit(v) to a chosen value v in a *binding* manner – that is, the commitment C cannot then be opened to another value $v' \neq v$. In the context of privacy-preserving protocols, we are furthermore interested in commitment schemes with the *hiding* property, where the commitment reveals no information about the value: in other words, where C' := Commit(v') is (statistically) indistinguishable from C. The hiding property is used in creating privacy-preserving state commitments in Zcash [22], Penumbra [35], and renegade.fi [7], to name a few. Furthermore, *homomorphic* commitments (such as the widely-used

Pedersen commitment [30]) support linear operations on the commitments that, when opened, yield the same result as if the operations had been performed on the original values. This property is used, for instance, in ZSA Swaps [24] to preserve balance invariants on the swap transaction.

Zero-Knowledge Proofs. Zero-knowledge proofs (ZKPs) enable one party, the prover, to convince another party, the verifier, that they possess a satisfying witness for a given statement, without revealing any other information about the witness. These statements are commonly used to enforce protocol invariants on private data, such as spend authority, set membership; and well-formation constraints, such as range checks and proof of correct encryption. In the context of programmable privacy protocols, network nodes typically play the role of verifiers. Protocols may choose to enshrine a fixed set of circuits (e.g. [22], [35]), or to allow users to define their own circuits (e.g. [4], [9]). Multiple developer tools and compilers have been deployed [17, 6, 18, 1, 25, 20, 37, 34] to allow users to specify arbitrary statements in zero-knowledge. Most uses of ZKPs in production today (e.g. Zcash [22]) assume that a single party holds the entire witness; however, multiple provers could also collaborate to produce a proof, as explored in [33]; this is deployed in [7] to prove the correctness of a multi-party computation.

Multi-Party Computation. Multi-party computation (MPC) enables multiple parties, each possessing their own private data, to jointly compute a specified function over their inputs without revealing those inputs to each other. MPC is commonly used to "threshold-ize" other cryptographic systems that depend on a single encryption key, such as fully homomorphic encryption [8] or trusted execution environments [36]; they can also be used as nested protocols to produce non-interactive cryptographic objects like zero-knowledge proofs [33]. The security of an MPC protocol is typically defined in the presence of adversaries (semi-honest or malicious), by demonstrating that for any real protocol execution, there exists a computationally indistinguishable simulated execution, thereby ensuring that no additional information is leaked beyond the computation's output. Notions of *fairness* [15] are also salient when quantifying privacy leakage in the presence of corrupt parties who abort the protocol prematurely.

Homomorphic Encryption. Homomorphic encryption (HE) enables blind computation on encrypted data without requiring access to the secret decryption key, thereby allowing the manipulation of ciphertexts in a manner that, when decrypted, yields the same result as operations performed on the plaintext. Formally, an encryption scheme E is defined as homomorphic over an operation \circ if for any pair of plaintexts m_1, m_2 and their corresponding ciphertexts $c_1 = E(m_1), c_2 = E(m_2)$, there exists an operation \diamond on ciphertexts such that $c_1 \diamond c_2 = E(m_1) \diamond E(m_2) = E(m_1 \circ m_2)$. Penumbra [35] makes use of an additively homomorphic encryption scheme to blindly aggregate individual swaps into a batch swap. A scheme is considered *fully homomorphic* (FHE) if it supports both addition and multiplication operations over ciphertexts, allowing for arbitrary computations; recent advancements [44, 14, 41, 13, 39] in FHE have brought the technology closer to being deployed in production. In the setting of programmable privacy blockchains, HE is often deployed alongside supplementary cryptographic primitives, such as: zero-knowledge proofs to attest to the correct encryption of inputs; and MPC to "thresholdize" the encryption key.

Trusted Execution Environment. Trusted Execution Environments (TEEs) are secure areas within a processor designed to execute code and handle data in a confidential and integral manner. TEEs provide a level of assurance regarding the execution of sensitive operations, safeguarding against unauthorized access and tampering from both software and hardware threats. This isolation ensures that code and data loaded within the TEE are protected with respect to confidentiality and integrity, facilitating a range of secure applications, from mobile payment processing to secure computing in cloud environments. There have been multiple deployed variants of TEE-enhanced blockchains [26]; a notable recent effort is the Sirrah TEE coprocessor, developed as part of the SUAVE project [16], which aims to enable the private execution of order flow operations.

3 Computational Framework

We now describe the computational framework on which we will base our comparison and analysis of privacyoriented blockchain applications. Our model consists of repetition of a basic computational loop over multiple epochs, where users submit preauthorized intents to network operators, who work to combine these potentially *conditional* intents into *unconditional* transactions which can be applied to the global network state. In particular, we are interested in applications where user inputs encode *private* information, and network operators do not learn this private information even when conducting nontrivial computations over inputs from multiple users. Privacy properties of this kind usually have to be coupled with assumptions made by an application's security model, such as trust assumptions about honesty of network operators or hardness assumptions about underlying cryptographic primitives. Later in the section, we will discuss privacy properties and security models as they relate to our specific computational setting.

3.1 Protocol Model

Decentralized blockchain applications take place in a computational setting with many complexities and technical hurdles due to an adversarial environment, a permissionless network architecture, and a need to achieve asynchronous distributed consensus. In defining an abstract protocol model, we aim to represent a class of applications which demonstrate a full range of current and upcoming techniques for efficient multi-user private computation, while also simplifying some details of blockchains that play a smaller role in the fidelity of the analysis. We model a protocol as an iterated computation taking place in a sequence of *epochs*, where several different types of protocol users take turns providing inputs and conducting computations to produce a shared protocol state. Each epoch consists of following three stages of computation.

Independent Computation. In this stage, asynchronous users submit input *intents* to the protocol which provide conditional authorization for state updates involving resources or data fields owned by that user. An intent generically specifies conditions which need to be satisfied in order to be applied to the application state, for instance "I will release up to 1 ETH from my address as long as I receive DOGE in return at an exchange rate of at least 20,000 to 1". However, an application may also allow for unconditional user intents, such as "I will transfer 50 ZEC from my owned address to the following address", representing traditional user transactions.

For privacy-preserving applications, the independent computation stage may involve some computation over private input data (such as the preimage of a cryptographic hash or a value privately committed on chain) which is attested to via a zk-SNARK proof. This enables a user to verifiably generate an arbitrary computational output for the protocol, as long as it depends only on public data and *their own* private data, but not on any other user's private data. Aggregating private data encoded by other users falls into the work done in the next computational stage.

Mediated Computation. In the mediated computation stage, a set of application or protocol *operators* follow a mutually agreed upon protocol to combine intents submitted by users. The output of this stage is one or more *unconditional* transactions which satisfy the requirements imposed by their constituent intents. Because the preferences of different users are integrated during this computation stage, mechanism design surrounding how this integration occurs may have a significant impact on the outcomes for individual participants, e.g. by discouraging MEV, or by enforcing some notion of fair trade execution.

When user intents encode private inputs, techniques are available which allow a verifiable joint computation to take place without operator nodes learning the concealed information from these inputs. Generically, this type of computation is challenging to implement: the techniques involved, such as secure multiparty computation, homomorphic encryption, and the use of secure hardware enclaves, have subtle requirements and failure modes, and rely on non-trivial security assumptions. In light of these challenges, many of the innovations taking place in blockchain applications and protocols today are centered around new ways to structure and integrate the mediated computation stage.

Global Computation. The global computation stage represents the application-level computations that are finally executed by full nodes, with visibility by all network participants. This stage in particular includes transaction sequencing — which transactions are included in a block and in what order — as well as execution of transactions against the current application state. Commonly, this is also where all verifiable computation proofs are validated.

3.2 Ideal Functionality

In Table 1, we give a formal description of this computational framework in the form of a UC-style ideal functionality. The functionality maintains an application state, a collection of received user intents, pending and executed protocol transactions, and a static corruption state describing an adversary's capabilities with respect to the protocol architecture.

An underlying abstraction of this ideal functionality is that protocol end-users are treated as proper "parties" of the protocol who provide input and receive output, while protocol operators are bundled into the logical presentation of the functionality, and are not treated as participants who explicitly interact with the protocol. The observable effects of interaction by protocol operators are instead presented as constrained input provided by the UC adversary, who can choose operator outputs which best suit its needs within some reasonable modeling constraints. We will describe the specific instances of this design pattern in more detail as we discuss the individual phases of the ideal functionality.

Upon initialization, the ideal functionality provides default values for the variables it maintains, including a starting application state and a static corruption state for the protocol. The latter in particular will play an important role throughout the functionality in describing the ways that an adversary is able to bend the basic operation of the protocol to its advantage. For simplicity we choose to represent the locus of adversarial control by a fixed value selected at protocol initialization. However, more dynamic corruption behavior can be accomplished either by modifying the ideal functionality to allow the adversary to periodically update the corruption state according to some rules, or within the present functionality by representing some dynamic capabilities in the specification of the static corruption state. The specific details of how the corruption state is represented and interpreted is left unspecified to allow for flexibility in applying the framework to varied protocol architectures.

In Phase 1, parties submit intents to the network to request further processing and an update to the application state. Intents in the functionality are represented as a "plaintext intent" describing the desired application logic, and "metadata" describing a user's mode of interaction with protocol operators. For example, in a protocol with semi-trusted operators who have view access to a user's protocol inputs, a user may choose which of a collection of operators to submit their intent to for processing in an eventual multiparty computation between operators. This selection would be specified in the ideal functionality by the value of the submitted metadata. Upon submission, some data about the intent may be leaked to the adversary, depending on the intent metadata and the protocol corruption state. For instance, in the previous example, if the semi-trusted operator which was selected by the user is controlled by the adversary, then the full plaintext of the intent would be visible to the adversary, represented by the leakage function $\mathcal{L}_{\text{Submit}}$. In addition, the adversary is allowed to submit intents to the protocol for subsequent execution during this phase.

In Phase 2, we see the first instance of the adversary simulating the behavior of protocol operators by way of a constrained input. The adversary provides an "Advice" input to the ideal functionality which is passed directly to the MediatedComputation function, along with the current list of intents and the application state. This produces a list of new transactions which are to be added to the protocol's pending transaction pool, as well as an internal Result variable which represents the logical outcome of the mediated computation to support security modeling throughout the remainder of the phase.

After the mediated computation is executed internally, a validation check is done on the results to allow the protocol to constrain the adversary's Advice input to properly model behavioral assumptions about operator input. One might be tempted to run this check on the adversary's Advice directly, prior to running the mediated computation, but this runs afoul of the following issue: the adversary's advice representing the operation of the mediated computation may need to consist of a *strategy* (e.g. a program) whose validity can only be easily described in terms of its effects on computation involving intents or data fields to which the adversary does not have access. An important instance of this is the example of an "intent solver" based architecture, where the adversary might specify as input a solver strategy which is used to produce an "overall best operator solution" — but the solution should satisfy some constraints in terms of optimality or execution quality to model the assumption that honest operators will normally be able to find and submit such solutions. In this case, it is likely more effective to interrogate the solution produced by the adversary's strategy rather than attempting to derive properties of the solving strategy by direct inspection, and so this concrete solution can be recorded in the Result variable for reference in the validity check. (If a direct

$\mathcal{F}_{\mathsf{ThreeStageEpochModel}}$

```
Initialization
AppState := InitialApplicationState()
IntentSet := ∅
PendingT×s := ∅
ExecutedT×s := []
CorruptionState := StaticCorruptionState()
```

Phase 1: Independent computation

```
\begin{array}{l} \textbf{On input (Intent, Metadata) from party $\mathcal{P}$:} \\ & \text{add (Intent, Metadata) to IntentSet} \\ & \textbf{Leak $\mathcal{L}_{Submit}$(Intent, Metadata, CorruptionState) to $\mathcal{A}$} \\ & \textbf{On input (Intent, Metadata) from $\mathcal{A}$:} \\ & \text{add (Intent, Metadata) to IntentSet} \end{array}
```

Phase 2: Mediated computation

```
On input (Advice) from \mathcal{A}:
   Result, NewTxs := MediatedComputation(IntentSet, AppState, Advice)
  assert ComputationAdviceIsValid(Result, CorruptionState)
  add elements of NewTxs to PendingTxs
  If SatisfiesSecurityAssumptions(CorruptionState):
     \mathbf{Leak} \ \mathcal{L}_{\mathsf{Med}}^{\mathsf{Expected}}(\mathsf{Result},\mathsf{IntentSet},\mathsf{AppState},\mathsf{CorruptionState}) \ \mathrm{to} \ \mathcal{A}
  Else :
      \mathbf{Leak} \ \mathcal{L}_{\mathsf{Med}}^{\mathsf{Broken}}(\mathsf{Result},\mathsf{IntentSet},\mathsf{AppState},\mathsf{CorruptionState}) \ \mathrm{to} \ \mathcal{A}
Phase 3: Global computation
On input (Advice) from \mathcal{A}:
  {\sf Result}, \, {\sf Txs} := {\tt SequenceTransactions}({\sf PendingTxs}, {\sf AppState}, {\sf Advice})
  assert SequencingAdviceIsValid(Result, CorruptionState)
  For Tx in Txs:
      AppState = UpdateState(AppState, Tx)
     remove Tx from PendingTxs
      append Tx to ExecutedTxs
   Output GlobalView(AppState) to global communication channel
   For each party \mathcal{P}:
      Output PrivatePartyView(AppState, \mathcal{P}) to \mathcal{P}
  Leak PrivateAdversaryView(AppState) to \mathcal{A}
```

Table 1: Ideal functionality representing operation of the three-stage protocol model.

check on the input Advice is all that is required, this value can likewise be recorded in the Result variable for this purpose.) Once this validity check is passed, the list of new transactions produced by the mediated computation is added to the application's global pending transaction set.

As a last step, any information revealed to the adversary during the mediated computation phase is formally leaked. This leakage of course depends on the inputs and outputs of the mediated computation, but importantly, it is very sensitive to the corruption state of the protocol. We divide the leakage represented by this ideal functionality into two branches, depending on whether the corruption state satisfies what the protocol considers the "core security assumptions" of the mediated phase. For example, if privacy depends on a k of n threshold decryption step, then this would be something like "at most k - 1 operators are corrupted", or if the mediated computation takes place in a TEE, then it could be "the trusted hardware enclave is secure, and the adversary does not have access to hidden key material that would allow spoofing outputs". Since many cryptographic techniques for conducting secure computation over private data depend on such "make or break" assumptions, we model the leakage as two separate leakage functions, one for when the core security assumption is satisfied, and one for when it is not.

In Phase 3, we finalize the state updates that are applied to the application state and communicate public and private updates to all relevant parties. Again in this phase, the UC adversary provides Advice input to simulate the operation of protocol operators, in this case to model the selection and ordering of transactions which are to be applied to the application state. As in Phase 2, the adversary's input is provided to an internal function call which has access to potentially private application state (especially, the list of pending transactions, some or all of which may be hidden from the adversary depending on the protocol architecture), and the result of this internal function call is then inspected in order to constrain the adversary's Advice to properly model the sequencing provided by the protocol's operation. Once sequencing is executed and validated, the chosen transactions are executed and applied to the application state one by one, and are transferred from the set of pending transactions to the ordered list of executed transactions. When this is finished, a global view of the application state is output to a global communication channel, and appropriate private views are output to protocol participants and the adversary.

As a technical note, since protocol parties in UC style protocols are usually addressed dynamically, the For loop in this phase which provides private output to parties should be interpreted as a placeholder for a more detailed concrete implementation, which practically might involve either an interactive communication protocol with participants to validate identity in some way, or maintaining a cached array of known parties with which the application logic requires private communication.

3.3 Privacy

Various concepts of privacy can be formulated concretely in our computational framework in terms of data leaked in the ideal functionality. In particular, since the framework involves iterated repetition over multiple epochs, we aim to provide definitions which describe a notion of leakage which describes an upper bound on the information which may be exposed to an adversary as a result of providing input to the protocol.

Because of the generality of such a notion of leakage in the context of an iterated distributed protocol, it is useful to allow specialization to protocol execution which is constrained to having some specific properties which enable a higher level of privacy. This could be as simple as "the corruption state satisfies the base protocol security assumptions", or could include restrictions on the current or future application state, on the contents of intents submitted by other parties, or more granularly on the protocol instance's corruption state. We call such a specification an **operation mode**, and we incorporate an operation mode in our definitions for long-term privacy properties to increase expressiveness.

In the following, leakage is specified as a function of an intent input. We require that the format of an intent is specified by a fixed data schema, given as a list of fields, data types, and basic consistency conditions required by an input to the protocol. It is then assumed that an input intent is of the correct format when provided to the protocol by a party.

Let \mathcal{L}_M be a function on tuples (Intent, Metadata, AppState), and let \mathcal{O} be an operation mode. We say that a protocol satisfies **bounded leakage** \mathcal{L}_b with respect to operation mode \mathcal{O} if when a participant sends input (Intent, Metadata) in the individual computation phase of the protocol with current application state AppState, then under any evolution of the protocol which satisfies \mathcal{O} , the information leaked to the adversary about Intent can be efficiently computed from \mathcal{L}_b (Intent, Metadata, AppState). As a basic definition for privacy purposes, we say that intent submission of a protocol has **confidentiality** with respect to a set S of one or more fields of an intent input under operation mode \mathcal{O} if the protocol satisfies bounded leakage \mathcal{L}_b with respect to \mathcal{O} which is constant over the inputs in S.

It is worth reiterating that this notion of confidentiality is abstracted at the protocol level, and does not address out-of-band leakage which may occur due to the operation of the underlying concrete communication network (e.g. a traditional peer to peer communication network usually reveals the IP addresses of network participants). As such, a complete privacy analysis of a protocol must take into account the broader context in which the ideal functionality is implemented.

4 Analysis of Protocols

We now give a comparison of contemporary protocols which support aspects of programmable privacy. As an initial summary, we organize the protocols into two structured classification schemes: a comparison lattice describing the computational methodologies we observed for incorporating private data and computations into public network state updates, and a table summarizing other salient characteristics associated with these protocols. Following this summary, we will provide a more detailed discussion and analysis for a subset of protocols which we believe give a representative view of the techniques currently being employed.

4.1 Comparison of Techniques and Security Assumptions

Privacy-preserving computation on a public or distributed computational engine can approximately be divided into two phases, corresponding to the straightforward observation that "single-party computation is privacy-preserving". In the first phase, individual protocol participants execute computation on their own private data in isolation, and produce a convincing zero-knowledge proof of the outcome which can be consumed by the public network. This phase of computation is distinct because the single-party nature of the computation allows for techniques which can ignore privacy entirely except for in the final protocol output. The rapidly developing field of general-purpose zk-SNARKs provides a thoroughly vetted toolkit for providing functionality of this kind with strong privacy guarantees, but there are various paradigms which make trade-offs between different measures of efficiency and security, and of the degree of programmability that is revealed to smart contract developers.

Programmability Assumptions for privacy	Fully programmable	Specialized functionality
k-of- N security	Sunscreen darkpool [40]	ZSwap (Penumbra) [35] Renegade darkpool [7]
Hardware security	SecretSwap [31]	
Economic security	Anoma [19], AlphaSwap (Aleo) [3] Aztec [4], Mina [29]	ZSA Swap (Zcash) [24]

Table 2: Comparison lattice of protocols categorized according to their programmability and security assumptions for privacy in the mediated computation phase.

We dedicate an axis to this phase of computation, dividing the approaches into the categories of "fully programmable", and "specialized functionality". Fully programmable applications come in many flavors, but share the feature that developers are able to specify application logic to interact with users' private inputs. Specialized computation techniques encompass a variety of approaches to enabling private computation without accruing the sometimes severe overhead associated with general-purpose programs; for instance, in the case of zk-SNARKs, this may include targeted circuit designs which have high efficiency at the cost of significantly reduced flexibility (e.g. the Groth16 SNARK), or more specialized verifiable computation techniques representing non-generic computational tasks.

For a network protocol with multiple interacting parties, at some point it is necessary to address the question of how the private inputs of different participants are to be combined in a single computation to

Phase Protocol	Independent computation	Mediated computation	Global computation
ZSA Swap [24] / Anoma [19]	trading intent revealed to matcher/solver	corrupt matcher/solver can reveal trading intent	anonymity set revealed by anchor
renegade.fi darkpool [7]	"lit" fields revealed publicly	corrupt counterparty can hangup and learn match outcome without revealing their own order	anonymity set revealed by anchor
SecretSwap [31] (Secret Network)	sender revealed to network	individual trades can be revealed if trusted hardware is compromised	batch trade outcome
Penumbra ZSwap [35]	asset ids revealed to network	individual swap amounts can be decrypted if a quorum of validators collude	batch trade outcome

Table 3: Comparison lattice of protocols categorized according to their information leakage in the independent, mediated, and global phases of computation.

produce a verifiable output. If one has access to a trusted intermediary then this task becomes trivial: each participant may encrypt their inputs to the trusted party who will conduct the agreed upon computation, and each participant will receive the appropriate output and any shared state will be updated appropriately. Permissionless blockchain protocols, which allow by their design the participation of untrusted or even adversarial parties, are specifically interested in ways to reduce or partition this absolute trust assumption. In particular there are large trade-offs between efficiency and security in all known schemes providing privacy-preserving distributed computation with improved trust assumptions.

The second axis of our comparison lattice enumerates the categories of techniques employed by protocols to enable computation on the network which incorporates the private inputs of multiple participants. In practice this is usually accomplished via the operation of some class of "online" network participants such as proof of stake validators to conduct a privacy-preserving computation using inputs provided by transactions submitted by "offline" network participants, but in theory it would be possible to describe a protocol in which offline participants contribute to a common privacy-preserving computation, e.g. in some number of asynchronous phases or via a short online interaction. We have not, however, encountered protocols which implement such a scheme.

The techniques employed to enable multi-party privacy-preserving computations are more varied than those typically used for single-party processing, but can be aggregated into the following coarse categories. "Cryptographically secured" computation uses techniques such as FHE and SMPC to ensure with mathematical guarantees that privacy is maintained during the multi-party phase of computation. "Economically secured" computation makes use of economic design or physical engineering to impose a monetary cost on violating desired privacy assumptions. We include hardware solutions such as TEEs in this category since a dedicated adversary with access to sufficiently sophisticated hardware and expertise (at cost!) could break security properties attained by such hardware under standard operating conditions. "Socially secured" computation relies on some mechanism of social proof such as a reputation system, or even just relying on a trusted operator to maintain privacy without further protocol-level mitigations.

Our lattice of protocols categorized along these two axes is given in Table 2.

4.2 Comparison of Limit Order Applications

The goal here is to specify the limit order auction, using implementations in existing protocols as concrete examples rather than giving abstract notation. Since the focus is on the limit order functionality as a motivating application, we exclude additional actions such as: transferring account balances; depositing/withdrawing from public to private; voting; and more general capabilities. In Table 3, we compare the privacy leakage of each protocol across the phases of computation. We have selected a range of representative protocols that enable privacy-preserving computation in the mediated phase, and have at least a proof-of-concept implementation of a limit order auction application at the time of writing. This is not meant to be an exhaustive enumeration of available protocols, but a minimal covering set of the design space.

The first class of protocols, which include ZSA Swaps [24] and Anoma [19], makes use of only **zero-knowledge proofs** for privacy on the network level. This does not allow third parties to perform computations directly on the encodings of private data; instead, trading intents must be fully revealed to off-chain solvers/matchers. These intermediaries are incentivized to keep intents secret in order to remain competitive in the marketplace, and to avoid loss of market share in case of detection of wrongdoing. On the network level, no information about the orders is revealed besides some upper bound on the anonymity set in which their notes belong.

The second class of protocols, of which the **renegade.fi** darkpool [7] is a representative example, makes use of **multi-party computation** to jointly compute matches between private orders. Users may choose to "light up" certain fields in their order (e.g. the asset pair being traded) in order to facilitate counterparty discovery. Here, a corrupt counterparty can hangup the MPC protocol to learn the match outcome without revealing their own order. As in the first case, this match outcome is validated by the network using zeroknowledge proofs, but remains hidden from those not party to the MPC.

The third class of protocols, for which we use SecretSwap [31] as a representative example, makes use of **trusted hardware** in the mediated computation phase. The user's identity and input asset are publicly revealed to the network; however, the trade amount and output asset are encrypted to a network public key, and can only be decrypted within a network node containing the corresponding private key. If this private key is exfiltrated from the secure hardware, every trade would be decryptable. Otherwise, the network observes only the batch outcome of the trades in each asset pair.

The final class of protocols makes use of **threshold homomorphic encryption** to blindly aggregate individual swaps into a batch swap on-chain. Penumbra's ZSwap [35] is a representative example of this. The user reveals the asset pair involved in the order, but homomorphically encrypts the input amount to the network's threshold public key. A validator then homomorphically sums the inputs to each asset pair; following that, a quorum decrypts the batch trade and executes it against public liquidity positions. As in the previous class of protocols, the network observes the outcome of the batch trades; in other words, if some trade is the only one in its asset pair, the network can directly observe its original input amount. (NB: Newer protocols such as Zama [21] and Sunscreen [40] make use of advancements in FHE to support more generic functionality; however, these protocols are not in production and do not have limit order auction applications at the time of writing.)

In Appendix A, we detail UC-style definitions for both ideal functionalities and concrete implementations of these protocols; the rest of this section provides high-level summaries, highlighting key features of the protocols' state models; limit order auction implementations; and privacy properties.

4.2.1 ZSA Swap

The Zcash protocol maintains an append-only set of confidential notes, where each note can be anonymously spent only once. Most computation on the network is done offline by individual users, who produce and publish note commitments, nullifiers, and ownership proofs to the blockchain; the network nodes are tasked with verifying these proofs, and checking that the notes had not been previously spent.

The shielded portion of the Zcash blockchain state consists of the *note commitment tree* and the *nullifier* set. Each note commitment is a cryptographically binding and hiding encoding of a note, which records some quantity of value together with a spending capability describing the keypair controlling that value. The note commitment tree is a Merkle tree over the set of note commitments. The anonymity set for each note consists of all the note commitments corresponding to the anchor in the spend proof.

When spent, a note produces a unique nullifier. An unspent valid note, at a given point on the blockchain, is one for which the note commitment has been revealed on-chain previously, but the nullifier has not. Each spend of a note must be accompanied by an ownership proof, which attests to the spender's authority and to the note's membership in the note commitment tree. In practice, to reduce the risk of the membership proof being invalidated by a reorg, the tree root is picked from a few blocks deep.

With the introduction of Zcash Shielded Assets (ZSAs) [24], the Zcash protocol will be able to support

multiple asset types; this also introduces the possibility of asset swaps, as described in our protocol model for ZSA Swap (Table 5). Since note values and types are concealed, there is no way to directly construct an order-book on-chain. Instead, the swap protocol proceeds as follows:

- 1. **Independent computation.** The party to a swap spends her input note and pre-emptively creates an output note, specifying the desired value and asset type of the outcome.
- 2. Mediated computation. The party reveals the values and note types involved in the swap to an offline third-party *matcher*, who balances her spend with another party's complementary spend. Since parties do not reveal their spend authority to the matcher, the matcher is unable to forge or alter a spend, or to otherwise use it in a way does not fulfil the initial spend's desired outcome.
- 3. Global computation. The matcher sends the balanced transaction to the network. Each party is able to directly decrypt and spend her output note.

Privacy

- The choice of anchor leaks information about the notes being spent, namely that the notes in the transaction were created no later than the anchor's block height. In the original Zcash protocol, all transfers in the same transaction must use the same anchor; however, in the case of ZSA Swap, this condition is relaxed to allow the matcher to form transactions with transfers using different anchors.
- The note values and types involved in each swap are revealed to the matcher.

4.2.2 renegade.fi

renegade.fi [7] is a peer-to-peer matching protocol implemented using multi-party computation (MPC), with an off-chain gossip network as the underlying communication channel. Successful matches are submitted to and eventually settled on a blockchain. For simplicity of exposition, we ignore fees, deposits, and withdrawals, and consider only trading actions between funded wallets. We also fix the asset pair to some (id_1, id_2) .

The darkpool contract maintains a Merkle tree over a set of wallet commitments, and a nullifier set marking nullified wallets. The matching protocol proceeds in three stages (Table 6):

- 1. **Independent computation**. The wallet owner commits to her order using a privacy-preserving encoding, and computes a set of zero-knowledge proofs attesting that the order is well-formed. The order commitment, along with the proofs of well formation, are broadcast to the gossip network as a handshake.
- 2. Mediated computation. Upon receiving a handshake request from a potential counterparty, the party validates the proofs of well formation and initiates an MPC. They jointly compute the outcome of the match, and a zero-knowledge proof that the match is well-formed. Upon verifying the match, each party sends their secret-share of the outcome to the counterparty. Either party may then submit the match outcome and accompanying proofs to the blockchain.
- 3. Global computation. Upon validating the well-formation of a match and the orders involved, the network nodes nullify the spent wallets, and insert commitments to the updated wallets.

Privacy

- In practice, users delegate their wallets to service providers known as *relayers*, who consolidate orders and perform both internal matching and the MPC protocol as described above. In other words, relayers are able to see all details of users' wallets; however, they cannot create or execute orders other than those specified by the user.
- The MPC protocol scheme fails to have opening fairness: the first party to receive the other's shares can simply hang up the connection without replying with their corresponding shares. In this way, a malicious party could learn its counterparty's orders without revealing her own.

4.2.3 Secret Network

Secret Network is an instance of TEE-based smart contracts. This is an instance of the Ekiden architecture [12], where the execution layer involves enclave nodes, and relies on ciphertexts published to an ordinary blockchain as the messaging layer.

The network is associated with a public key. To send a transaction, the "calldata" of the transaction, which includes the bids, is encrypted to this public key. The mediated computation is then carried out by the validator nodes, which run the computations within TEEs. All of the TEEs on the network share a copy of a master decryption key.

Although the trade is encrypted, some information is leaked through the metadata of a Secret Network transaction. Because gas is collected through ordinary Cosmos modules, outside of the trusted hardware, the pseudonymous address of the sender is leaked. Additionally the token that is being offered in the trade (token A) is revealed. This is because a swap involves three main contract queries executed in a sequence, where only the "first" is visible before finalization: first a transfer of token A to the swap contract, then the swap itself, and finally a transfer of token B to the user.

In our protocol model (Table 4), we skip the bootstrapping steps and assume that keys are already distributed to each validator node. Validator nodes are practically limited to choosing the order of the transactions in a block, based on the visible metadata alone, which constitutes an adversary's influence over the mediated computation phase. The global application state is updated only once a block is finalized, after which each party is able to read their updated token balance.

Note that our model of Secret Network operation is somewhat optimistic in that it assumes fixes for replay attacks and freshness attacks have been applied, such as those discussed in [23].

4.2.4 Penumbra

The Penumbra protocol supports sealed-bid batch swaps. Users create a Swap intent that privately burns their input assets and homomorphically encrypts the amount to validators. Encrypted Swap intents are *batched* in a block and executed against an (effectively) global liquidity pool. Using their original Swaps and the public result of the batch swap, users can then claim the output of their swap.

The shielded part of the Penumbra state model is similar to Zcash's, with modifications to support additional functionality. The Penumbra state commitment tree supports arbitrary state fragments (such as swaps), instead of single-asset notes as in Zcash. Penumbra uses the shielded pool to its full effect when collecting orders. To capture this minimal leakage, the ideal functionality (Table 7) includes a Leak statement, that only includes a tag. It leaks that an order was placed, but it conceals the direction of the order, the quantity, and any context about the sender.

The batch swap protocol proceeds in the following stages:

- 1. **Individual computation.** Users submit swaps to the blockchain, with the asset pair in plaintext and the limit order amount encrypted to the network public key.
- 2. Mediated computation. After selecting a batch to process, a validator determines the order of swaps in the batch, and homomorphically sums the *total* amount of trades for each asset pair. Recall that Penumbra does not express reserve prices, or rather the reserve price is always zero.
- 3. Global computation. A quorum of nodes decrypts the total batch swap for each asset pair, and executes the swap against the public liquidity pools.

Privacy

- An individual Swap's amount is concealed behind the net flow across its trading pair in the batch. However, this provides no protection if the batch contains a single swap, and limited protection when there are few other swaps.
- A SwapClaim reveals the specific block and trading pair of the original Swap, thus limiting its anonymity set to swaps within a single block.

5 Analysis and Conclusions

In this work, we provide a generic three phase framework for comparing programmable privacy protocols on blockchains across various cryptographic techniques and security assumptions. When thinking about programmability, one imagines a program that "can represent any NP language". In a private setting, this does not seem to capture the full range of capabilities, nor allude to the existing trade-offs. Our comparison of limit order auctions across multiple programmable privacy protocols has revealed trade-offs in the design space of cryptographic primitives and security assumptions. We summarise these and discuss practical design choices corresponding to specific protocol requirements, arguing for a phase-restricted fully programmable privacy protocol to reach the optimal trade-off between privacy, security and programmability.

5.1 Tradeoffs Between Programmability, Privacy, and Security

Functionalities that have a need for computation on private inputs of multiple users – which include limitorder auctions, swaps, DEXes and more – require the *aggregation of, and computation on, private inputs*. We observe the clearest tradeoffs to have emerged along the axes of (1) programmability and privacy, as well as (2) programmability and application security.

5.1.1 Programmability and Privacy

The classes of security assumptions we rely on for privacy include: economic security, hardware security, and k-of-n security. Designs that rely solely on economic security make the largest compromise in privacy, revealing the plaintext inputs to the operator entity; the latter two obscure the input through a layer of indirection. Designs that rely on k-of-n security often introduce limitations on programmability, since complex and fully general functions are less efficient to thresholdize. In practice, these security assumptions can be layered for defense-in-depth: for instance, having each trusted hardware module be financially staked and slashable; or having each party in the threshold scheme hold its key share in a trusted hardware module.

5.1.2 Programmability and Application Security

We also observe that protocols with limited application programmability, such as ZSA Swap [24], Penumbra [35], and renegade.fi [7], limit the potential security vulnerabilities created by "external" developers when building applications on chain. Of course, users are limited in the type of applications they can interact within this case, and may need to either use a non-private protocol instead, or a different protocol that enables that functionality. On the other hand, some protocols provide exhaustive programming capabilities and allow users to build any type of application. Protocols that are closer to this side of the spectrum tend to focus more on developing the language and compiler design that enables a user to write a program in a well-known language and produce a cryptographic-based distributed application at the protocol level. Examples of this include Mina zkApps [29], Aztec [4], AlphaSwap (Aleo) [3], SecretSwap (Secret Network) [31], and Sunscreen darkpool [40].

In fact, we argue that today, there is no protocol that has designed and implemented an optimal trade-off. Yet, the phased computational framework we define in this work creates a paradigm shift for thinking about programmability and extensibility of functionality. By dividing application logic and protocol evolution into several carefully defined phases, we are able to encapsulate the specific processes of the protocol with their respective cryptographic techniques, allowing for a more secure design.

5.2 Future Research: Optimal Programmable Privacy Design

In this section, we aim to show that, in fact, private programmability is a broar endeavor and does not finish with the protocol design. We believe that there is still research to be done in this space to define the optimal private programmability technology stack.

5.2.1 Designing the Tech Stack

When thinking about designing cryptographic systems, especially with an aim on programmability and privacy, one needs to consider the interaction and dependencies across elements of the technology stack. Let's quickly review the full stack of the programmability technology:

- At the lowest level, we have the cryptographic schemes used for obtaining privacy. These include ZKP, MPC, FHE and TEEs, many of which are seeing increasingly practical and efficient implementations being deployed in production.
- Next, we have the specific representation of the NP statement to be used within the cryptographic scheme. Usually this is called the circuit, and it can have many representations (i.e. for ZKPs there are R1CS, PlonK, AIR, and more; for FHE and MPC, boolean and arithmetic circuits).
- A level higher, we have the compiler, which takes in a regular program, describing some functionality, and maps it into the circuit representation.
- Further up, we have the actual domain-specific language (DSL) used to write the programs. The language needs to be specifically tailored to recognise the underlying cryptographic challenges and to minimize the potential for vulnerabilities introduced by the developer.
- Finally, we encounter the protocol itself, which in the context of blockchain applications, is what allows the user interaction with the applications, the data and other users. It also defines the execution model in which application logic takes place.

Beyond the security requirements of the protocol, while designing the DSL and compilers within a privacypreserving distributed system, it is important to provide abstractions for developers to insulate application designs from the cryptographic complexity of the underlying system. This is accomplished well in a number of privacy protocols such as Mina [29] and Aztec [4] which provide languages which compile from an understandable high-level syntax to suitable low-level representations in terms of zk-SNARK proofs and protocol execution logic — but which also omit a built-in mediated computation phase from their architectures.

On the other hand, among protocols providing programmable mediated computation, it did not seem that any provided a high-level language that sufficiently abstracts the subtleties involved in the mediated computation design. As an example, Secret Network [31], which provides mediated computation using Intel SGX hardware enclaves, documents a significant selection of security considerations [32] involving subtleties of the privacy provided by the trusted execution platform that contract developers need to understand when writing applications. In such cases, it would be better for the front-end language to encapsulate these design "best-practices" by default, without relying on developers to internalize complex and subtle cryptographic considerations. The way in which front-end languages can accomplish this kind of cryptographic abstraction for different methods of secure mediated computation does not appear to be well explored, and may be a fruitful direction for future research and design experimentation.

Finally, a common thread running through most of the protocols considered is the composition of multiple cryptographic primitives in a hybrid framework. We highlight the importance of an ergonomic frontend that allows for sufficient programmability while safeguarding the user from potential privacy leaks resulting from side-channel attacks exposed in this composition.

5.2.2 Best Practices

After reviewing dozens of protocols claiming to support programmable privacy, we highlight the importance of defining a common language and set of standard practices across projects. This would enable proper communication, education and R&D practices. Some of the ones we have observed work well, or are missing, include:

- Projects should first and foremost have well written, extensive documentation of the protocol that includes privacy goals, security assumptions, concrete design choices and their motivation.
- Explicit analysis of the privacy leakage and security model of the concrete implementation, which can differ from the theoretical exposition.

- Do not expose the low-level cryptographic functions to the users developing applications as this could create security vulnerabilities.
- We recommend projects use the framework described in this work to base the exposition of the privacy and security, and the extent of the programmability of functionality.

In the light of this analysis, a possible path for designing a programmability protocol with optimal trade-off could be to build what we call a *phase-based programmability* protocol. This design involves using generically programmable cryptographic schemes with high security assurance (e.g. ZKPs, FHE and MPC) that suit the different phases of independent computation and mediated computation, as we have seen. The protocol should provide boundaries around the phases, so that the program compilation is restricted and maps the different steps of the functionality directly into the corresponding phases and cryptographic schemes. Furthermore, the user should develop the applications using a highly typed and structured language that abstracts all the cryptographic components, to avoid unintended security lapses from their composition.

Acknowledgements

This work was commissioned and partly funded by the Zcash Foundation, for which we greatly appreciate the support. We would also like to thank Chris Bender and Joey Kraut for useful conversations about the Renegade dark pool MPC protocol, as well as Pablo Kogan for feedback on our framework and helpful information on the design of the ZSA Swap functionality.

References

- [1] Aleo. The leo programming language, 2024. https://developer.aleo.org/leo.
- [2] G. Almashaqbeh and R. Solomon. Sok: privacy-preserving computing in the blockchain era. In 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P), pages 124–139. IEEE, 2022.
- [3] AlphaSwap. Alphaswap first dex on aleo, 2024. https://alphaswap.pro/.
- [4] Aztec. Aztec protocol documentation, 2023. https://docs.aztec.network/aztec/overview.
- [5] C. Baum, J. H.-y. Chiang, B. David, and T. K. Frederiksen. Sok: Privacy-enhancing technologies in finance. *Cryptology ePrint Archive*, 2023.
- [6] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing*, 2022.
- [7] C. Bender and J. Kraut. Renegade whitepaper, protocol specification, v0.6, 2023. https://www. renegade.fi/whitepaper.pdf.
- [8] D. Boneh, R. Gennaro, S. Goldfeder, A. Jain, S. Kim, P. M. Rasmussen, and A. Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Advances in Cryptology-CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38, pages 565–596. Springer, 2018.
- [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), pages 947–964. IEEE, 2020.
- [10] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. Journal of computer and system sciences, 37(2):156–189, 1988.
- [11] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings* 42nd IEEE Symposium on Foundations of Computer Science, pages 136–145, 2001.

- [12] 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), pages 185–200. IEEE, 2019.
- [13] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.
- [14] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap. Concrete: Concrete operates on ciphertexts rapidly by extending tfhe. In WAHC 2020-8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, 2020.
- [15] R. Cohen, I. Haitner, E. Omri, and L. Rotem. From fairness to full security in multiparty computation. In *International Conference on Security and Cryptography for Networks*, pages 216–234. Springer, 2018.
- [16] T. F. Collective. Suave protocol specifications, 2024. https://github.com/flashbots/suave-specs.
- [17] E. C. Company. The halo2 book, 2024. https://zcash.github.io/halo2/index.html.
- [18] C. contributors. Circom circuit compiler, 2024. https://docs.circom.io/.
- [19] C. Goes, A. S. Yin, and A. Brink. Anoma: a unified architecture for full-stack decentralised applications. 2022.
- [20] L. Goldberg, S. Papini, and M. Riabzev. Cairo—a turing-complete stark-friendly cpu architecture. Cryptology ePrint Archive, 2021.
- [21] R. Hindi. Zama fhe evm demo, 2023. https://www.youtube.com/watch?v=C8uVqUFS1X0.
- [22] D. Hopwood, S. Bowe, T. Hornby, N. Wilcox, et al. Zcash protocol specification. GitHub: San Francisco, CA, USA, 4(220):32, 2016.
- [23] N. Jean-Louis, Y. Li, Y. Ji, H. Malvai, T. Yurek, S. Bellemare, and A. Miller. SGXonerate: Finding (and Partially Fixing) Privacy Flaws in TEE-based Smart Contract Platforms Without Breaking the TEE. Proceedings on Privacy Enhancing Technologies, pages 617–634, Jan. 2024.
- [24] P. Kogan, V. Arte, D.-E. Hopwood, and J. Grigg. Zip 227: Issuance of zcash shielded assets, 2022. https://zips.z.cash/zip-0227.
- [25] O. Labs. oljs compiler, 2024. https://docs.minaprotocol.com/zkapps/oljs-reference.
- [26] R. Li, Q. Wang, Q. Wang, D. Galindo, and M. Ryan. Sok: Tee-assisted confidential smart contract. arXiv preprint arXiv:2203.08548, 2022.
- [27] Y. Lindell. How to simulate it a tutorial on the simulation proof technique. In Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich, pages 277–346. Springer International Publishing, Cham, 2017.
- [28] U. Maurer. Constructive Cryptography A New Paradigm for Security Definitions and Proofs. In S. Mödersheim and C. Palamidessi, editors, *Theory of Security and Applications*, Lecture Notes in Computer Science, pages 33–56. Springer, 2012.
- [29] Mina. Mina docs, 2023. https://docs.minaprotocol.com/.
- [30] M. Naor. Bit commitment using pseudorandomness. Journal of cryptology, 4:151–158, 1991.
- [31] S. Network. Secret markets: Crypto front running prevention for automated market makers (amms), 2020. https://scrt.network/blog/secret-markets-front-running-prevention/.
- [32] S. Network. Secret network potential contract level attacks, 2023. https://docs.scrt. network/secret-network-documentation/overview-ecosystem-and-technology/techstack/ privacy-technology/theoretical-attacks/contract-level.

- [33] A. Ozdemir and D. Boneh. Experimenting with collaborative {zk-SNARKs}:{Zero-Knowledge} proofs for distributed secrets. In 31st USENIX Security Symposium (USENIX Security 22), pages 4291–4308, 2022.
- [34] A. Ozdemir, F. Brown, and R. S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In 2022 IEEE Symposium on Security and Privacy (SP), pages 2248–2266. IEEE, 2022.
- [35] Penumbra. The penumbra protocol, 2023. https://protocol.penumbra.zone/main/index.html.
- [36] O. P. Project. An implementation of ekiden on the oasis network, 2021. https://oasisprotocol.org/ technology#papers.
- [37] A. Protocol. The noir programming language, 2024. https://noir-lang.org/docs/dev/.
- [38] A. Rondelet and Q. Kilbourn. Threshold encrypted mempools: Limitations and considerations. arXiv preprint arXiv:2307.10878, 2023.
- [39] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, Jan. 2023. Microsoft Research, Redmond, WA.
- [40] R. Solomon. Building a truly dark dark pool, 2024. https://blog.sunscreen.tech/ building-a-truly-dark-dark-pool-2/.
- [41] Sunscreen. Sunscreen documentation, 2024. https://docs.sunscreen.tech/.
- [42] N. van Saberhagen. Monero whitepaper: Cryptonote v2.0., 2013. https://github.com/ monero-project/research-lab/blob/master/whitepaper/whitepaper.pdf.
- [43] S. van Schaik, A. Seto, T. Yurek, A. Batori, B. AlBassam, C. Garman, D. Genkin, A. Miller, E. Ronen, and Y. Yarom. Sok: Sgx.fail: How stuff gets exposed, 2022.
- [44] Zama. What is concrete?, 2024. https://docs.zama.ai/concrete/.

A UC-Style Protocol Definitions

$\mathcal{F}_{\text{SecretSwap}}(\text{id}^{\text{A}}, \text{id}^{\text{B}})$
Initialization Positions: a list of liquidity positions, where each entry is of the form $pos := ((R^A, p^A), (R^B, p^B))$ Tokens: a set of token contracts, each specified by an asset id and containing list of balances: Token _{assetId} := assetId, balances : [\mathcal{P} , \$val] PendingSwaps: a list of submitted swaps, where each entry is of the form swap := { \mathcal{P} , assetId ⁱⁿ , assetId ^{out} , \$val}, initially [] IncludedSwaps: a list of swaps checked to be valid, where each entry is of the form swap := { \mathcal{P} , assetId ⁱⁿ , assetId ^{out} , \$val}, initially [] Phase 1: Independent computation
<pre>On input (swap, swap := {P, assetIdⁱⁿ, assetId^{out}, \$val}) from P or A: Leak(P, assetIdⁱⁿ) to A append swap to PendingSwaps Phase 2: Mediated computation On input (trade, PendingSwaps): for swap_i ∈ PendingSwaps:</pre>
If \$outputVal := ExecuteSwap(swap _i , Positions) succeeds: append swap _i to IncludedSwaps Phase 3: Global computation On input (execute, IncludedSwaps): for swap _i ∈ IncludedSwaps: If \$outputVal := ExecuteSwap(swap _i , Positions) succeeds:
update the input token balance Token _{swapi.assetld} ⁱⁿ .balances[\mathcal{P}] -= swapi.\$val update the output token balance Token _{swapi.assetld} ^{out} .balances[\mathcal{P}] += \$outputVal
$\Pi_{SecretSwap}(id^A,id^B)$
Initialization Each secure enclave in the network has a copy of the master decryption key. Positions: a list of liquidity positions, where each entry is of the form $pos := ((R^A, p^A), (R^B, p^B))$ Tokens: a set of token contracts, each specified by an asset id and containing list of balances: Token _{assetId} := assetId, balances : $[\mathcal{P}, \$val]$ PendingSwaps: a list of submitted swaps, where each entry is of the form $swap := \{\mathcal{P}, assetId^{in}, Enc(assetId^{out}), Enc(\$val)\}$, initially [] IncludedSwaps: a list of swaps checked to be valid, where each entry is of the form $swap := \{\mathcal{P}, assetId^{in}, assetId^{out}, \$val\}$, initially [] Phase 1: Independent computation On input (swap, swap := $\{\mathcal{P}, assetId^{in}, assetId^{out}, \$val\}$) from \mathcal{P} or \mathcal{A} : encrypt Enc(assetId^{out}), Enc(\\$val) to the secure enclave; replace the plaintext values in swap with the ciphertexts append swap to PendingSwaps Phase 2: Mediated computation On input (trade, PendingSwaps): reorder PendingSwaps based on the leaked values of (swap. \mathcal{P} , swap.assetId ⁱⁿ) for swap _i \in PendingSwaps: decrypt the swap's output token assetId ^{out} and value $\$val$
If \$outputVal := ExecuteSwap(swapi, Positions) succeeds: append swapi to IncludedSwaps Phase 3: Global computation On input (execute, IncludedSwaps): for swapi ∈ IncludedSwaps: decrypt \$\mathcal{P}\$'s input token balance \$bal ⁱⁿ := Dec(Token _{swapi,assetIdⁱⁿ} .balances[\$\mathcal{P}\$]) update \$\mathcal{P}\$'s input token balance \$bal ⁱⁿ -= swapi,\$val re-encrypt \$\mathcal{P}\$'s input token balance \$bal ⁱⁿ := Enc(\$bal ⁱⁿ) decrypt \$\mathcal{P}\$'s output token balance \$bal ^{out} := Dec(Token _{swapi,assetId^{out}.balances[\$\mathcal{P}\$]) update \$\mathcal{P}\$'s output token balance \$bal^{out} := Dec(Token_{swapi,assetId^{out}.balances[\$\mathcal{P}\$]) update \$\mathcal{P}\$'s output token balance \$bal^{out} := Enc(\$bal^{out}.balances[\$\mathcal{P}\$]) update \$\mathcal{P}\$'s output token balance \$bal^{out} := Enc(\$bal^{out}.balances[\$\mathcal{P}\$]) update \$\mathcal{P}\$'s output token balance \$bal^{out} := Dec(Token_swapi,assetId^{out}.balances[\$\mathcal{P}\$]) update \$\mathcal{P}\$'s output token balance \$bal^{out} := Enc(\$bal^{out}.balances[\$\mathcal{P}\$]) update \$\mathcal{P}\$'s output token balance \$bal^{out} := Enc(\$bal^{out})}}

Table 4: SecretSwap ideal functionality in UC style, and description of concrete implementation.

 $\mathcal{F}_{\mathsf{ZSA-Swap}}(\mathsf{id}^\mathsf{A},\mathsf{id}^\mathsf{B})$

Initialization \mathcal{F}_{ZS}	A_{-Swap} interacts	with parties $\{\mathcal{P}_{i}\}$	$_{i}$ and matcher \mathcal{M} .
-----------------------------------	-----------------------	------------------------------------	------------------------------------

 $\mathsf{Notes:} \ a \ \mathsf{list} \ of \ unspent \ \mathsf{notes}, \ \mathsf{each} \ of \ \mathsf{the} \ \mathsf{form} \ \mathsf{note} := \{\mathsf{owner}: \mathcal{P}, \mathsf{assetValue}: (\mathsf{assetId}, \$val)\}$

PendingSwaps: a list of pending swap orders maintained by the matcher \mathcal{M} , each of the form swap := {input : note, output : note} PendingMatches: a list of pending matches, each of the form match := (swap₁, swap₂)

Phase 1: Independent computation

 $\begin{array}{l} \mathbf{On\ input}\ (\mathsf{swap}, \mathsf{input}: \{\mathsf{owner}: \mathcal{P}, \mathsf{assetValue}: (\mathsf{assetId}^{\mathsf{in}}, \$val^{\mathsf{in}}) \}, \mathsf{output}: \{\mathsf{owner}: \mathcal{P}, \mathsf{assetValue}: (\mathsf{assetId}^{\mathsf{out}}, \$val^{\mathsf{out}}) \} \ \mathrm{from}\ \mathcal{P} \ \mathrm{or}\ \mathcal{A}: \\ \mathsf{send}\ \mathsf{swap}:= \{\mathsf{input}, \mathsf{output}\} \ \mathrm{to}\ \mathsf{PendingSwaps}, \ \mathrm{maintained}\ \mathrm{by}\ \mathcal{M} \end{array}$

Phase 2: Mediated computation

$\mathrm{for}~\mathsf{swap}_i \in \mathsf{PendingSwaps}:$

If $\mathcal M \mbox{ is corrupt: } \mathbf{Leak} \mbox{ swap}_i.\mbox{assetValue to } \mathcal A$

 $\mathbf{If} \; \mathsf{swap}_i.\mathsf{input} \notin \mathsf{Notes}: \mathsf{skip} \; \mathsf{swap}_i$

 $\mathrm{for}~\mathsf{swap}_j \in \mathsf{PendingSwaps}:$

 $\mathbf{If} \; \mathsf{swap}_j.\mathsf{input} \notin \mathsf{Notes}: \mathsf{skip} \; \mathsf{swap}_j$

 $\mathbf{If} \; (\mathsf{swap}_i.\mathsf{input.assetValue} == \mathsf{swap}_j.\mathsf{output.assetValue}) \land (\mathsf{swap}_i.\mathsf{output.assetValue} == \mathsf{swap}_j.\mathsf{input.assetValue}) :$

 $\mathrm{remove}\;[\mathsf{swap}_i,\mathsf{swap}_j]\;\mathrm{from}\;\mathsf{PendingSwaps}$

append match := $(swap_i, swap_j)$ to PendingMatches

Phase 3: Global computation

for match $:= (swap_1, swap_2) \in \mathsf{PendingMatches}$:

If swap_1.input \notin Notes OR swap_2.input \notin Notes : skip match

remove [swap1.input, swap2.input] from Notes; append [swap1.output, swap2.output] to Notes

$\Pi_{\mathsf{ZSA}-\mathsf{Swap}}(\mathsf{id}^\mathsf{A},\mathsf{id}^\mathsf{B})$

Initialization $\Pi_{ZSA-Swap}$ interacts with parties $\{\mathcal{P}_i\}$ and matcher \mathcal{M} .					
NoteCommitments: a list of commitments to notes, where each commitment has the form					
$cm := Commit(note) := Commit(\{owner: \mathcal{P}, assetValue: (assetId, \$val)\})$					
Nullifiers: a set of nullifiers marking spent notes					
PendingSwaps: a list of pending swap orders maintained by the matcher \mathcal{M} , each of the form					
$swap := \{input: assetValue, output: assetValue, actionStatement: (\pi_{action}, nf, cm^{in}, cm^{out}), cvs: (cv^{in}, cv^{out}), bsk\}$					
PendingMatches: a list of pending matches maintained by the matcher \mathcal{M} , each of the form					
$match := (swap_1.actionStatement, swap_1.cvs, swap_2.actionStatement, swap_2.cvs, \sigma)$					
Phase 1: Independent computation					
$ \textbf{On input} (swap, input : \{owner : \mathcal{P}, assetValue : (assetId^{in}, \$val^{in})\}, output : \{owner : \mathcal{P}, assetValue : (assetId^{out}, \$val^{out})\} from \mathcal{P}: $					
commit to input note $cm^{in} := Commit(input)$; commit to output note $cm^{out} := Commit(output)$					
compute the nullifier nf for the input note					
compute an action proof π_{action} that nf, cm ^{out} are well-formed, and cm ⁱⁿ \in NoteCommitments					
sample a random r^{in} and use it to commit to input value $cv^{in} := CommitAssetValue(r^{in}, input.assetValue)$					
sample a random r^{out} and use it to commit to output value $cv^{out} := CommitAssetValue(r^{out}, output.assetValue)$					
compute $bsk := r^{in} - r^{out}$ and $bvk := cv^{in} - cv^{out}$					
$construct \ swap := \{input : input.assetValue, output : output.assetValue, actionStatement : (\pi_{action}, nf, cm^{out}), cvs : (cv^{in}, cv^{out}), bsk\}$					
send swap to matcher \mathcal{M} , who appends it to PendingSwaps					
Phase 2: Mediated computation					
for swap, \in PendingSwaps:					
$\mathbf{If} \; swap_{i}.actionStatement.nf \in Nullifiers \; OR \; !SNARK.Verify(swap_{i}.actionStatement.\pi_{action}) : skip \; swap_{i}$					
$\mathbf{For} \ swap_{j} \in PendingSwaps$					
$\mathbf{If} \; swap_{j}.actionStatement.nf \in Nullifiers \; OR \; !SNARK.Verify(swap_{j}.actionStatement.\pi_{action}) : skip \; swap_{j}$					
$\mathbf{If} \ bvk \neq swap_i.cvs.cv^{in} - swap_i.cvs.cv^{out} + swap_j.cvs.cv^{in} - swap_j.cvs.cv^{out} : skip \ swap_j$					
$\mathbf{If} \; (swap_i.input.assetValue == swap_j.output.assetValue) \land (swap_i.output.assetValue == swap_j.input.assetValue):$					
remove [swap _i , swap _j] from PendingSwaps					
$ ext{compute bsk} := ext{swap}_i. ext{bsk} + ext{swap}_i. ext{bsk}, ext{bvk} := [ext{bsk}]\mathcal{R}$					
$compute \ \sigma := {\sf Signature.Sign}_{\sf ssk}({\sf sigHash}), \ where \ {\sf sigHash} := {\sf Hash}({\sf swap}_i.{\sf actionStatement} {\sf swap}_i.{\sf cvs} {\sf swap}_j.{\sf actionStatement} {\sf swap}_j.{\sf cvs})$					
$append \ match := (swap_i.actionStatement, swap_i.cvs, swap_j.actionStatement, swap_j.cvs, \sigma) \ \mathrm{to} \ PendingMatches$					
Phase 3: Global computation					
$\text{for match} := (swap_1.actionStatement, swap_1.cvs, swap_2.actionStatement, swap_2.cvs, \sigma) \in PendingMatches:$					
assert swap ₁ .actionStatement.nf \notin Nullifiers \land swap ₂ .actionStatement.nf \notin Nullifiers					
assert SNARK.Verify(swap1.actionStatement. π_{action}) \land SNARK.Verify(swap2.actionStatement. π_{action})					
compute a hash of public information in the swaps, sigHash := Hash(swapi.actionStatement swapi.cvs swapi.actionStatement swapj.cvs)					
assert Signature. Validate(σ , sigHash, bvk), where bvk := swap ₁ .cvs.cv ⁱⁱⁱ - swap ₁ .cvs.cv ^{out} + swap ₂ .cvs.cv ⁱⁱⁱ - swap ₂ .cvs.cv ^{out}					
insert {swap1.actionStatement.nf, swap2.actionStatement.nf} into Nullifiers; insert {swap1.cvs.cm ^{out} , swap2.cvs.cm ^{out} } into					
NoteCommitments					

Table 5: ZSA-Swap ideal functionality in UC style, and description of concrete implementation.

```
\mathcal{F}_{\mathsf{Simplified}-\mathsf{Match}-\mathsf{Renegade}}(\mathsf{id}^\mathsf{A},\mathsf{id}^\mathsf{B})
  Initialization \mathcal{F}_{\mathsf{Simplified}-\mathsf{Match}-\mathsf{Renegade}} interacts with a party \mathcal{P}_1 and counterparty \mathcal{P}_2.
              \mathsf{Wallets:} \ a \ set \ of \ wallets, \ each \ of \ the \ form \ wallet := (\mathcal{P}, \mathsf{B} := (\$bal^\mathsf{A}, \$bal^\mathsf{B}), \\ \mathsf{order} := (\mathsf{side}, \$limit\_price, \$amt, \min\_fill))
 Phase 1: Independent computation
 \mathbf{On\ input\ (wallet:=\{\mathcal{P}, \mathsf{B}:=(\$bal^{\mathsf{A}},\$bal^{\mathsf{B}}), \mathsf{order}:=(\mathsf{side},\$\mathsf{limit\_price},\$\mathsf{amt},\mathsf{min\_fill})\}), \$price)\ from\ \mathcal{P}:(\mathsf{A},\mathsf{A}) \in \mathsf{A}, \mathsf{A}) \in
              If \mathcal{P} is corrupt: Leak order to \mathcal{A}
              if side == buy: assert (bal^B \ge bal^B \ge bal^A = bal^A \ge bal^A \ge bal^A \ge bal^A \ge bal^A \ge bal^A = bal^
              broadcast (order, $price)
 Phase 2: Mediated computation
 On input (match, (order<sub>1</sub>, order<sub>2</sub>)) and ($price<sub>1</sub>, $price<sub>2</sub>):
               assert(side_1 \neq side_2) \land assert(\$price_1 == \$price_2)
              set $price := $price<sub>1</sub>; compute match := ComputeMatch(order<sub>1</sub>, order<sub>2</sub>, $price)
              If \mathcal{P}_i, i \in \{1, 2\} is corrupt: Leak match to \mathcal{A}
              \mathbf{Else} : store match
 Phase 3: Global computation
\mathbf{On~input}~(\texttt{settle}, \texttt{match} := (\$ \mathsf{val}^\mathsf{A}, \$ \mathsf{val}^\mathsf{B}, \mathsf{dir})):
              if dir == 0 (\mathcal{P}_1 is buying id<sup>A</sup>, \mathcal{P}_2 is buying id<sup>B</sup>):
                              increase wallet<sub>1</sub>.B.bal^A += val^A, decrease wallet<sub>1</sub>.B.bal^B -= val^B
                              increase wallet<sub>2</sub>.B.bal^{B} += val^{B}, decrease wallet<sub>2</sub>.B.bal^{A} -= val^{A}
              else if dir == 1 (\mathcal{P}_1 is buying id<sup>B</sup>, \mathcal{P}_2 is buying id<sup>A</sup>):
                             increase wallet<sub>1</sub>.B.bal^{B} += val^{B}, decrease wallet<sub>1</sub>.B.bal^{A} -= val^{A}
                             increase wallet<sub>2</sub>.B.bal^A += val^A, decrease wallet<sub>2</sub>.B.bal^B -= val^B
              remove order_1 from wallet<sub>1</sub>; remove order_2 from wallet<sub>2</sub>
```

 $\Pi_{\mathsf{Simplified}-\mathsf{Match}-\mathsf{Renegade}}(\mathsf{id}^\mathsf{A},\mathsf{id}^\mathsf{B})$

Initialization $\Pi_{\mathsf{Simplified}-\mathsf{Match}-\mathsf{Renegade}}$ interacts with a party \mathcal{P}_1 and counterparty \mathcal{P}_2 . Note that the notation share is used to represent secret shares of a value, with the superscripts [[]^{pub}, [[]^{priv} denoting, respectively, public secret shares which are published to the network, and private secret shares which remain known only to the user. ${cm_i^{wallet}}$. Commitments: \mathbf{a} set of wallet commitments where each wallet is of the form $\mathsf{wallet} := \{\mathsf{B} := (\$\mathsf{bal}^\mathsf{A}, \$\mathsf{bal}^\mathsf{B}), \mathsf{order} := (\mathsf{side}, \$\mathsf{limit_price}, \$\mathsf{amt}, \mathsf{min_fill}, \mathsf{r})\}$ Nullifiers: a set of nullifiers marking spent wallets $\{nf_i\}$ Phase 1: Independent computation $\mathbf{On~input}~(\mathsf{wallet}:=\{\mathcal{P},\mathsf{B}:=(\$\mathsf{bal}^\mathsf{A},\$\mathsf{bal}^\mathsf{B}),\mathsf{order}:=(\mathsf{idx},\mathsf{side},\$\mathsf{limit_price},\$\mathsf{amt},\mathsf{min_fill},\mathsf{r}),\$\mathsf{price})~\mathrm{from}~\mathcal{P}:$ $\text{if side} == \texttt{buy: set \$bal} := \$\texttt{bal}^{\mathsf{B}} \text{; else if side} == \texttt{sell: set \$bal} := \$\texttt{bal}^{\mathsf{A}}$ $\text{compute nullifier} = \mathsf{H}(\mathsf{wallet} \mid\mid \mathsf{r}); \ \text{compute } (\pi^{\mathsf{comm}},\mathsf{H}_{\$\mathsf{bal}},\mathsf{H}_{\mathsf{order}},\mathsf{H}_{\mathsf{r}}) := \mathsf{OrderCommitments}(\$\mathsf{bal},\mathsf{order});$ $broadcast \ \mathsf{handshake} = \big((\mathsf{idx}, \llbracket \mathsf{nullifier} \rrbracket^{\mathsf{pub}}, \pi^{\mathsf{comm}}, \mathsf{H}_{\$\mathsf{bal}}, \mathsf{H}_{\mathsf{order}}, \mathsf{H}_{\mathsf{r}}), \$\mathsf{price}\big)$ Phase 2: Mediated computation **On input** (match, ((handshake₁, \$price₁), (handshake₂, \$price₂))): for $\mathcal{P}_i, \mathcal{P}_j, i, j \in \{1, 2\}, i \neq j$: if Handshake(handshake_1, $price_1$, handshake_2, $price_2$) == false: Abort ; else: set $price_1$ == $price_2$ $\mathrm{jointly\ compute\ }([\![\mathsf{match}]\!], [\![\mathsf{H}_{\mathsf{match}}]\!], [\![\mathsf{witness}]\!]) := \mathsf{Match}\mathsf{MPC}([\![\mathsf{amt}_1]\!], [\![\mathsf{amt}_2]\!], \$price), \mathrm{where}$ match := (val^A , val^B , dir); H_{match} := H(match|| \hat{r}), \hat{r} := H_{r1} + H_{r2} compute $[\pi^{\text{match}}] := \text{MatchValidityMPC}([witness])$, a proof that the MPC was correctly run each party sends shares of $[\pi^{match}]$ to the other; reconstruct and assert SNARK.Verify (π^{match}) each party sends shares of [match] to the other using the reconstructed match := (val^A , val^B , dir), update the public shares $[vallet'_1]^{pub}$, $[vallet'_2]^{pub}$ of each wallet to reflect: the updated balances of each wallet; and the removal of fulfilled orders from each wallet. Phase 3: Global computation $\mathbf{On~input}~(\texttt{settle}, \pi^{\texttt{match}}, \pi_1^{\texttt{comm}}, \pi_2^{\texttt{comm}}, [\![\texttt{wallet}_1']\!]^{\texttt{pub}}, [\![\texttt{wallet}_2']\!]^{\texttt{pub}}, \texttt{nullifier}_1, \texttt{nullifier}_2):$ assert SNARK.Verify(π_{match}) \land SNARK.Verify(π_{1}^{comm}) \land SNARK.Verify(π_{2}^{comm}) assert {nullifier₁, nullifier₂} \notin Nullifiers insert {nullifier1, nullifier2} into Nullifiers compute commitments to the updated wallet public shares $C_1^W := Comm(\llbracket wallet'_1 \rrbracket^{pub}), C_2^W := Comm(\llbracket wallet'_1 \rrbracket^{pub})$ insert C_1^W, C_2^W into Commitments

Table 6: Simplified-Match-Renegade ideal functionality in UC style, and description of concrete implementation. For simplicity of exposition, we omit protocol fees, wallet keychains, blinding/reblinding, and partial fills; we also fix a single asset pair and a single order per wallet.

```
\mathcal{F}_{\textbf{BatchAuction-Penumbra}}^{(t,n)}(\mathsf{id}^{\mathsf{A}},\mathsf{id}^{\mathsf{B}})
\mathcal{F}_{\mathsf{BatchAuction-Penumbra}} interacts with a set of parties \mathcal{P}_i, a set of validators \mathcal{V} = \{\mathcal{V}_i\}, and a block proposer \mathcal{V}^{\star} \in \mathcal{V}.
Initialization
   Coins: a multiset of coins, each of the form coin := (\mathcal{P}, id, \$val)
   Positions: a list of liquidity positions, where each entry is of the form pos := ((R^A, p^A), (R^B, p^B))
   PendingSwaps: a list of submitted swaps, where each entry is of the form swap := (coin^{A}, coin^{B}), initially []
  IncludedSwaps: a list of swaps included in the batch, where each entry is of the form swap := (coin^{A}, coin^{B}), initially []
Phase 1: Independent computation
On input (swap, swap := (coin^A, coin^B)) from party \mathcal{P} or \mathcal{A}:
  send (swap, (id^A, id^B)) to \mathcal{A} and append swap to PendingSwaps
Phase 2: Mediated computation
On input (batch, PendingSwaps) from A:
  If at least t validators are corrupt: Leak PendingSwaps to A
  If \mathcal{V}^* is not corrupt: assert(SwapSubset == PendingSwaps)
  initialize TotalBuysSells = (0, 0)
  for each (coin<sup>A</sup>, coin<sup>B</sup>) in SwapSubset:
     TotalBuysSells = TotalBuysSells + (coin<sup>A</sup>.$val, coin<sup>B</sup>.$val)
     remove (coin<sup>A</sup>, coin<sup>B</sup>) from PendingSwaps and append it to IncludedSwaps
  send TotalBuysSells to \mathcal{A}
Phase 3: Global computation
On input (execute, IncludedSwaps, TotalBuysSells := (val^A, val^B) from A:
   for each (coin^A, coin^B) tuple in IncludedSwaps: remove \{coin^A, coin^B\} from Coins
  compute (BatchOutputs, Unfilled) = ExecuteBatchSwap(TotalBuysSells, Positions)
  send (BatchOutputs, Unfilled) to \mathcal{A}
  \Pi_{\mathsf{BatchAuction-Penumbra}}^{(t,n)}(D,\mathsf{id}^\mathsf{A},\mathsf{id}^\mathsf{B})
Initialization
  StateCommitments: a list of blinded state commitments, which could consist of coin commitments {cm<sub>i</sub><sup>coin</sup>} or swap commitments
   \{cm_i^{swap}\}
  Nullifiers: a set of nullifiers marking spent coins \{nf_i\}
  Positions: a list of concentrated liquidity positions, where each entry is of the form pos := ((R^A, p^A), (R^B, p^B))
   PendingSwaps: metadata about submitted swaps {(\pi^{\text{spend}^A}, \pi^{\text{spend}^B}, \text{Enc}(\$val^A), \text{Enc}(\$val^B))}; initialized to \emptyset
  IncludedSwaps: metadata about swaps in the batch {(\pi^{swap}, \pi^{spend^A}, \pi^{spend^B}, Enc(sval^A), Enc(sval^B)}; initialized to \emptyset at the beginning of
  each block
Phase 1: Independent computation
On input (swap, coin<sup>A</sup> := ((\mathcal{P}, id<sup>A</sup>, $val<sup>A</sup>), coin<sup>B</sup> := (\mathcal{P}, id<sup>B</sup>, $val<sup>B</sup>))) from party \mathcal{P}:
  for id \in \{A, B\}:
     commit to cm^{id} := Commit(\mathcal{P}, id, \$val)
     homomorphically encrypt Enc_D(\$val^{id}) to the public threshold key
     compute a spend proof \pi^{\text{spend}^{\text{id}}} for coin<sup>id</sup> (if \text{sval}^{\text{id}} == 0, \pi^{\text{spend}^{\text{id}}} is a trivially valid "dummy proof")
  compute a swap proof \pi^{swap} and swap commitment cm^{swap} committing to the swap ids, values, and claim address, among other information
  append (\pi^{\text{swap}}, \pi^{\text{spend}^{\mathsf{A}}}, \pi^{\text{spend}^{\mathsf{B}}}, \text{Enc}(\$val^{\mathsf{A}}), \text{Enc}(\$val^{\mathsf{B}})) to PendingSwaps
Phase 2: Mediated computation
On input (batch, PendingSwaps):
   initialize TotalBuysSells := (Enc_D(0), Enc_D(0))
  for swap in PendingSwaps:
     assert SNARK.Verify(swap.\pi^{swap}) \land SNARK.Verify(swap.\pi^{spend^{A}}) \land SNARK.Verify(swap.\pi^{spend^{B}})
```

```
assert {swap.\pi^{\text{spend}^{A}}.nf, swap.\pi^{\text{spend}^{B}}.nf} \notin Nullifiers
```

 $update TotalBuysSells = TotalBuysSells + (swap.Enc_D($val^A), swap.Enc_D($val^B))$

remove swap from $\mathsf{PendingSwaps}$ and append swap to $\mathsf{ExecutedSwaps}$

decrypt all encrypted values in $\mathsf{Dec}_\mathsf{D}(\mathsf{TotalBuysSells});$ store $\mathsf{Dec}_\mathsf{D}(\mathsf{TotalBuysSells})$

Phase 3: Global computation On input (execute, IncludedSwaps, $Dec_D(TotalBuysSells) := (\$val^A, \$val^B))$:

 $compute \ ({\sf BatchOutputs}, {\sf Unfilled}) = {\sf ExecuteBatchSwap}({\sf TotalBuysSells}, {\sf Positions})$

for swap in IncludedSwaps: insert {swap. $\pi^{spend^{A}}$.nf, swap. $\pi^{spend^{B}}$.nf} into Nullifiers

insert swap.cm^{swap} into StateCommitments

store (BatchOutputs, Unfilled)

reset IncludedSwaps := \emptyset

Table 7: Penumbra BatchSwap ideal functionality in UC style, and description of concrete implementation. The protocol is simplified to ignore fees, deposits, and withdrawals, and considers only trading actions between funded wallets and for a single asset pair.