Jigsaw: Doubly Private Smart Contracts

Sanjam Garg[*] Aarushi Goel[†] Dimitris Kolonelos[‡] Rohit Sinha[§]

Abstract

Privacy is a growing concern for smart contracts on public ledgers. In recent years, we have seen several practical systems for privacy-preserving smart contracts, but they only target privacy of on-chain data, and rely on trusted off-chain parties with user data – for instance, a decentralized finance application (e.g. exchange) relies on an off-chain matching engine to process client orders that get settled on-chain, where privacy only applies to the on-chain data. Privacy conscious users demand stronger notions of privacy, for their identity and their data, from all other parties in the ecosystem.

We propose a novel framework for smart contracts that ensures *doubly private* execution, addressing both on-chain and off-chain privacy requirements. In our framework, clients submit their requests in a privacy-preserving manner to a group of (potentially mutually untrusting) servers. These servers collaboratively match client requests without learning any information about the data or identities of the clients.

We then present *Jigsaw*, an efficient cryptographic realization of our proposed framework. *Jigsaw* builds on the ZEXE architecture (Bowe et al., S&P 2020), which leverages zkSNARKs, and extends Collaborative zkSNARKs (Ozdemir and Boneh, USENIX 2022) to enable proof generation by a group of servers.

In Jigsaw, we introduce a novel collaborative zkSNARK construction that achieves low latency and reduced proving time, and showcase these advantages over sample applications ranging from trading in a decentralized exchange to auctions and voting. Our experiments demonstrate that *Jigsaw* is roughly $40 - 50$x faster in proof generation and uses orders-of-magnitude less bandwidth than the naive approach of using off-the-shelf Collaborative zkSNARKs.

[*]UC Berkeley and Exponential Science Foundation sanjamg@berkeley.edu
[†]Rutgers University aarushi.goel@rutgers.edu
[‡]UC Berkeley dimitris.kolonelos@berkeley.edu
[§]Swirlds Labs sinharo@gmail.com

Contents

1 Introduction **3**
- 1.1 Our Framework . . . 3
- 1.2 Our Contributions . . . 4
- 1.3 Related Work . . . 5

2 Technical Overview **6**
- 2.1 Core Challenges . . . 7
- 2.2 Our Solution . . . 8
- 2.3 Additional Technical Subtleties . . . 12

3 Preliminaries **12**
- 3.1 Zero Knowledge Succinct Arguments of Knowledge (zkSNARKs) . . . 12
- 3.2 Collaborative zkSNARKs . . . 14

4 Doubly Private Smart Contracts **15**
- 4.1 Syntax and Algorithms . . . 15
- 4.2 Security . . . 17

5 Jigsaw: Our Construction **20**
- 5.1 Design Choices . . . 20
- 5.2 Building Blocks . . . 21
- 5.3 zkSNARK Relations . . . 21
- 5.4 Protocol Description . . . 23
- 5.5 Instantiation and optimizations . . . 27

6 Applications **29**
- 6.1 Atomic Swap . . . 29
- 6.2 Sealed-bid Auctions . . . 31
- 6.3 Lotteries . . . 31
- 6.4 Voting . . . 32

7 Implementation and Evaluation **33**
- 7.1 Local Proving . . . 34
- 7.2 Collaborative Proving . . . 36
- 7.3 Verification: Time, Gas Fee, and Proof Size . . . 36
- 7.4 Discussion and Comparison . . . 37

A Security Proof **41**

B Commit-and-Prove PLONK **43**
- B.1 Argument for Linking KZG Committed Values . . . 43
- B.2 CP-PLONK . . . 44

1 Introduction

Smart contracts are user-defined programs that execute automatically on public blockchains when predefined conditions are met. Their success stems from ensuring integrity without relying on any single trusted party. Instead correctness is enforced by the blockchain's consensus mechanism. Platforms like Ethereum [Woo14] provide a powerful environment where all participants share a common data and computation history. This transparency, paired with strong integrity guarantees, enables a wide range of applications. However, the public nature of blockchain computations poses a challenge to privacy—especially for sensitive decentralized applications such as Decentralized Finance (DeFi), one of today's most prominent blockchain use cases.

To illustrate, consider a trading scenario involving two users, Alice and Bob. Alice holds 100 tokens of asset A and wishes to exchange those for 200 tokens of asset B. Conversely, Bob has 200 tokens of asset B and wishes to acquire 100 tokens of asset A. For privacy, several details must remain hidden when executing this trade: (1) the origin of tokens involved; (2) the rate at which tokens were exchanged; (3) quantity of tokens bought or sold.

Another example of a decentralized application involving sensitive data are sealed-bid auctions, where bidders can anonymously submit confidential bids. Here, it is crucial to keep the following details private: (1) the bid amounts; (2) the source of bidding amounts; (3) identities of bidders. Ensuring privacy in such applications while maintaining correctness of execution is critical.

To address this, several frameworks [KMS+16, KGC+18, BCG+20, CZK+19, SBG+19, BCT21, SBBV22, SBV22, BT22, SWA23, XCZ+23, ByCDF23] have been proposed to enable *privacy-preserving* smart contract execution, with zkSNARK-based approaches [KMS+16, BCG+20, SBG+19, SBV22, XCZ+23] being among the most popular. These approaches ensure that the information stored on the blockchain neither reveals the input-output data nor, in some cases, the program logic [BCG+20], all that while guaranteeing computational correctness through the blockchain's consensus mechanism. In essence, they offer integrity, anonymity and confidentiality guarantees while incurring minimal on-chain costs.

While prior solutions prevent on-chain exposure of sensitive data, they introduce significant off-chain privacy risks.[1] All existing privacy-preserving smart contract systems rely on a helper node or server to manage computations. For example, in the trading scenario, Alice and Bob each submit their requests to a server, who is responsible for determining that their trades are compatible, executing the trade and posting the final transaction on-chain. Similarly, in a sealed-bid auction, the server collects bids, determines the winner, finalizes the transaction, and submits it on-chain. This centralized model gives the server full visibility into client identities and data.

This raises an important question: Can we design smart contracts that are *doubly* private, i.e., ones that ensure complete confidentiality on-chain and during *off-chain counterparty discovery and computation*? Our goal is to achieve this while retaining the benefits of existing privacy-preserving smart contract frameworks.

1.1 Our Framework

We introduce a new framework for doubly private smart contracts that leverages a group of potentially mutually distrusting servers. These servers consist of independent off-chain entities distinct from blockchain validators. The clients start by sharing their private data with the servers in a privacy-preserving manner, who then collaboratively handle computations. This framework has the following features:

[1] This applies to all mainstream – i.e., widely deployed – approaches to private smart contracts. Some recent works achieve partial off-chain privacy but sacrifice other key properties discussed in Section 1.3.

- **Integrity:** As in any smart contract framework, through the blockchain consensus mechanism, we ensure that the conditions of the exchange are enforced, preventing fund locking or unauthorized creation of assets. As a result, in our framework, this guarantee remains intact even if all servers collude, preserving integrity of the system.
- **Fire-and-Forget:** All existing private smart contract solutions offer a highly desirable "fire-and-forget" feature, allowing clients to submit their data and leave. No further interaction is required between the clients, or between the clients and the server. This fire-and-forget feature is intended to mimic traditional markets, where parties can issue requests and exchange securities without knowledge of their counter-parties. Our framework preserves this fire-and-forget feature, where the servers are responsible for: (1) collecting the data of the clients (in a privacy-preserving manner); (2) identifying compatible requests; (3) executing the smart contracts and (4) posting the transactions on-chain.
- **On-Chain and Off-Chain Data Privacy:** In addition to on-chain data privacy guaranteed by existing smart contract solutions, our framework additionally guarantees off-chain data privacy. In particular, as long as at least one of the servers remains honest, the data of the smart contracts[2] are not revealed to *any* other party at *any* stage of the computation.
- **Hiding Client Identity:** Finally, in contrast to existing solutions, as long as at least one of the servers remains honest, the identity of the client remains hidden.[3]

Our framework is especially valuable for decentralized applications involving sensitive data of multiple users. The most common examples of such applications are in DeFi. In context of DeFi applications, our framework effectively replicates the capabilities of traditional market exchanges in a decentralized setting, while ensuring end-to-end transaction privacy.[4]

1.2 Our Contributions

We propose an efficient cryptographic approach to realize our framework, provide a proof-of-concept implementation demonstrating that it is orders of magnitude more efficient than naïve alternatives, and show the utility of our framework through a discussion of potential applications that could benefit from it.

Jigsaw. To realize our framework, we build on the widely adopted architectures of Zcash [BSCG+14] and ZEXE [BCG+20], both of which leverage zkSNARKs to ensure privacy. We begin by observing that a straightforward approach to realize our framework would involve using an off-the-shelf construction of collaborative zkSNARKs [OB22a]. Recall that collaborative SNARKs, introduced by Ozdemir and Boneh, allow a group of mutually distrusting servers to jointly compute a zkSNARK using their combined witnesses with privacy.

Indeed, collaborative zkSNARKs have been successfully used in private trading contexts, as demonstrated in Renegade [BK] and Taceo [tc24]. However, a key limitation of this approach is the significant

[2] Unlike some prior works [BCG+20], we do not focus on keeping the program logic of the smart contract hidden. See Section 1.3 for a detailed discussion.

[3] Note that the networking mechanism can reveal identifying information (e.g. ip address). Therefore, we must assume that the client is using anonymous communication systems (e.g. Tor) to interact with the servers. The exact mechanism for establishing such channels is out of our scope.

[4] As is typical in privacy-preserving systems, there are signals that could potentially be used to track users–e.g. timing correlations between transactions or transport layer information (such as IP address, packet sizes, etc.)–though we stress that they must be addressed separately, as they fall outside the scope of our work. In this work we focus on the cryptographic aspects of data privacy.

communication overhead it requires between participants. This overhead can significantly increase computation time, making them especially challenging to employ in blockchain settings.

To address this limitation, we introduce a novel primitive called *Jigsaw*, which offers a more efficient way to implement collaborative zkSNARKs. Specifically, *Jigsaw* retains the collaborative nature of zkSNARKs while reducing redundant work among participants, minimizing communication and thus decreasing the overall time required to generate the zkSNARK. In essence, *Jigsaw* can be viewed as an efficient replacement for traditional collaborative zkSNARKs. We anticipate that this approach will be useful not only for private smart contracts but also in other applications that rely on collaborative zkSNARKs.

Implementation. We implement *Jigsaw* in Rust using a combination of open source libraries – namely MP-SPDZ [Kel20], TurboPlonk [GW20] and collaborative Plonk [OB22b] (as implemented in [tc24]).

Applications. We demonstrate the broad applicability of *Jigsaw*, by using it to implement some popular real-world smart contract functionalities: Atomic Swaps, Sealed-bid Auctions, Lotteries and Voting. The cryptographic details of *Jigsaw* within these applications is described in Section 6, and performance testing is reported in Section 7. For instance, consider the atomic swap, an application with real-time performance needs. Our local proof computation takes about 1 second, while the collaborative prover takes 1.4 secs in a LAN setting or 2.2 secs in a WAN setting. Compare that to the baseline approach – running the entire relation within a collaborative SNARK protocol – which would take upwards of 60 secs, even with a 3 gigabit LAN connection; on lower bandwidth WAN connections, we can expect the proof generation to take upwards of 170 secs. Furthermore, we have an order of magnitude improvement in communication efficiency, thus providing not only a better user experience but also lower cloud computing costs.

1.3 Related Work

Privacy-Preserving Smart Contract Frameworks. Private smart contract frameworks often rely on zkSNARKs, starting with HAWK [KMS+16], which ensures data privacy but depends on a single trusted dealer and lacks off-chain privacy. ZEXE [BCG+20] extends this to hide contract functions, yet similarly requires a trusted entity with full off-chain visibility. VeriZEXE [XCZ+23] improves ZEXE's efficiency but inherits the same privacy limitations. Other zkSNARK-based frameworks like zkay [SBG+19] and Zapper [SBV22] offer usability but still leak off-chain privacy; zkay additionally lacks anonymity.

To address trust assumptions, zkHawk [BCT21] and V-zkHawk [BT22] replace the dealer with MPC among clients, requiring all to be online and interactive—impractical for blockchain settings. Eagle [ByCDF23] outsources MPC to a committee, achieving off-chain privacy and non-interactive clients. However, it introduces new trust assumptions: MPC servers can break blockchain rules, e.g., create unauthorized coins. Though mitigated via crypto-economic mechanisms, these servers become points of failure. In contrast, our framework preserves the blockchain's native trust model and consensus integrity—even with fully malicious MPC nodes.

Zeestar [SBBV22] and smartFHE [SWA23] use FHE (and zkSNARKs) but leak client identities. smartFHE supports multi-user contracts only via multi-key FHE, which to-date remains highly impractical. Approaches like Arbitrum [KGC+18] and Ekiden [CZK+19] follow a different paradigm with stronger trust assumptions than cryptographic solutions, the former relying on a trusted manager and the latter on TEE environments (trusted hardware).

On Function Hiding. Function privacy, unlike in ZEXE [BCG+20] and VeriZEXE [XCZ+23], is not our goal. While technically feasible, it introduces unnecessary overhead. We consider revealing the function on-chain a reasonable tradeoff, as done in prior works (e.g., [SBV22]), while prioritizing off-chain privacy as essential. Function privacy is left for future work.

Privacy for specific Blockchain Applications. Several works target privacy for specific use-cases, such as anonymous payments [MGGR13, BSCG+14, BAZB20], decentralized exchanges [BDF21], or MEV resistance [LSH+24], but lack general-purpose support.

MPC as a Computational Service. Recent works [CLMZ23, GGJ+23, LZW+24, YCW+24] use MPC for outsourcing zkSNARK computation. In contrast, we use MPC for privacy. These two use cases are orthogonal, as further discussed in Section 5.1.

2 Technical Overview

We use a decentralized exchange application to illustrate our techniques. Consider a simple smart contract that allows two parties to perform a swap. Specifically, client 1 holds v_1 units of token_1 (e.g. ETH) and wishes to sell them in exchange for $\widetilde{\mathsf{token}}_1$ (e.g. BTC) at rate_1 (i.e., client 1 wants $1/\mathsf{rate}_1$ $\widetilde{\mathsf{token}}_1$'s for every unit of token_1). Similarly, client 2 possesses v_2 units of token_2 and wishes to exchange it with $\widetilde{\mathsf{token}}_2$ at rate_2. We want to design a system where these clients can submit their requests and an exchange occurs when the following conditions are satisfied: (1) $\widetilde{\mathsf{token}}_1 = \mathsf{token}_2$, (2) $\widetilde{\mathsf{token}}_2 = \mathsf{token}_1$ and (3) $\mathsf{rate}_1 = 1/\mathsf{rate}_2$.[5]

Consider this trade. Say client 1 has 2 units of BTC and wants to exchange them for ETH at a rate of $1/30$; similarly, client 2 has 70 units of ETH and wishes to exchange them with BTC at a rate $30/1$ (allowing for leftover change). So, they submit the following orders:

- Client 1: $\mathsf{order}_1 = \{\ 2\ \mathsf{BTC}, \mathsf{BTC} \leftrightarrow \mathsf{ETH}, 1{:}30\ \}$.
- Client 2: $\mathsf{order}_2 = \{\ 70\ \mathsf{ETH}, \mathsf{ETH} \leftrightarrow \mathsf{BTC}, 30{:}1\ \}$.

As discussed in the previous section, existing privacy-preserving smart-contract systems [KMS+16, KGC+18, BCG+20, CZK+19, SBG+19, SBBV22, SBV22, SWA23, XCZ+23] rely on a single entity who receives all the orders, executes the smart contract and then posts a zero-knowledge proof certifying validity of the transaction on-chain. Zero-knowledge property of the proof ensures that the posted transaction does not leak any private information about the exchange. However, the trusted entity who performs all the computations of the exchange off-chain, inevitably learns all the information about clients' requests.[6]

To prevent such off-chain privacy leakage, in our framework, we replace a single trusted entity with a cluster of m servers who are *collectively* responsible for performing all the computations. Privacy in our framework is preserved even if all-but-one of the servers are malicious. Crucially, integrity (and non-interactivity) is preserved even when all servers are malicious. As noted in Section 1.3, we do not aim for function-privacy. In other words, the fact that a swap happened, does not remain hidden in our framework.

Data Structures. We build on the ZEXE [BCG+20] (which in turn builds on the Zerocash [BCG+14]) architecture. In short, our data is organized into records $\boldsymbol{r}$. Each record contains an address public key apk, a payload pd which contains the actual data of the record, a nonce ρ, randomness $\underline{\boldsymbol{r}}$ and a hiding commitment $\mathsf{cm} = \mathsf{Com}(\mathsf{apk}, \mathsf{pd}, \rho; \underline{\boldsymbol{r}})$: $\boldsymbol{r} = (\mathsf{cm}, \mathsf{apk}, \mathsf{pd}, \rho, \underline{\boldsymbol{r}})$. The record owner possesses the corresponding address secret key ask, such that apk is the output of a (pre-defined) pseudorandom function computed with key ask, i.e., $\mathsf{apk} = \mathsf{PRF}_{\mathsf{ask}}(0)$.

[5] Note that, in reality, it is more common to allow the exchange to take place at rate_1 even when $\mathsf{rate}_1 \geq 1/\mathsf{rate}_2$. For simplicity, we consider $\mathsf{rate}_1 = 1/\mathsf{rate}_2$ throughout this overview.

[6] Even if the trusted entity was one of the two clients of the DEX, say client 1, valuable information would still be leaked. For example the DEX could fail if there was no match in the orders but client 1 still gets to learn the order of client 2.

In our DEX application, the clients' input records are:

$$\boldsymbol{r}_1 = (\mathsf{cm}_1, \mathsf{apk}_1, (2, \mathsf{BTC}), \rho_1, \underline{\boldsymbol{r}_1})$$
$$\boldsymbol{r}_2 = (\mathsf{cm}_2, \mathsf{apk}_2, (70, \mathsf{ETH}), \rho_2, \underline{\boldsymbol{r}_2})$$

and the output records of the clients will be:

$$\widetilde{\boldsymbol{r}}_{1,1} = (\widetilde{\mathsf{cm}}_{1,1}, \widetilde{\mathsf{apk}}_{1,1}, (60, \mathsf{ETH}), \widetilde{\rho}_{1,1}, \underline{\widetilde{\boldsymbol{r}}_{1,1}}),$$
$$\widetilde{\boldsymbol{r}}_{1,2} = (\widetilde{\mathsf{cm}}_{1,2}, \widetilde{\mathsf{apk}}_{1,2}, (0, \mathsf{BTC}), \widetilde{\rho}_{1,2}, \underline{\widetilde{\boldsymbol{r}}_{1,2}})$$
$$\widetilde{\boldsymbol{r}}_{2,1} = (\widetilde{\mathsf{cm}}_{2,1}, \widetilde{\mathsf{apk}}_{2,1}, (2, \mathsf{BTC}), \widetilde{\rho}_{2,1}, \underline{\widetilde{\boldsymbol{r}}_{2,1}}),$$
$$\widetilde{\boldsymbol{r}}_{2,2} = (\widetilde{\mathsf{cm}}_{2,2}, \widetilde{\mathsf{apk}}_{2,2}, (10, \mathsf{ETH}), \widetilde{\rho}_{2,2}, \underline{\widetilde{\boldsymbol{r}}_{2,2}}).$$

Here, $\widetilde{\boldsymbol{r}}_{1,1}$, $\widetilde{\boldsymbol{r}}_{2,1}$ are the records used in the actual exchange and $\widetilde{\boldsymbol{r}}_{1,2}$, $\widetilde{\boldsymbol{r}}_{2,2}$ are for the leftover change. For privacy, we want both clients to receive a record corresponding to the leftover change, even if it has 0 value. Observe that this is necessary to avoid leaking to the adversary the fact that one client did not receive any change.

Whenever a new record is generated, its commitment cm is posted on-chain. We use a Merkle tree to store the set of all record commitments, such that the root denotes the current blockchain state. Similarly, whenever a record is consumed, its "nullifier" $\mathsf{sn} = \mathsf{PRF}_{\mathsf{ask}}(\rho)$ goes on-chain. A privacy-preserving transaction for a smart contract predicate f over the payloads associated with some records comprises of: (1) nullifiers of all the consumed records ($\mathsf{sn}_1, \mathsf{sn}_2$ in our case); (2) commitments of all the newly generated records ($\widetilde{\mathsf{cm}}_{1,1}$, $\widetilde{\mathsf{cm}}_{1,2}$, $\widetilde{\mathsf{cm}}_{2,1}$, $\widetilde{\mathsf{cm}}_{2,2}$ in our case); and (3) a zkSNARK proof π of correct execution of f, i.e.[7]

$$f(\mathsf{pd}_1, \mathsf{pd}_2, \widetilde{\mathsf{pd}}_{1,1}, \widetilde{\mathsf{pd}}_{1,2}, \widetilde{\mathsf{pd}}_{2,1}, \widetilde{\mathsf{pd}}_{2,2}, \mathsf{order}_1, \mathsf{order}_2) = 1.$$

We refer to Section 4 for a detailed description.

We must now design a privacy-preserving mechanism to: (1) execute f on the inputs of the clients and (2) compute the zkSNARK proof π. Importantly, all of this should be done without relying solely on *one* trusted entity.

2.1 Core Challenges

The naïve approach to implement this would be to let the clients participate in a secure multi-party computation (MPC) protocol to compute (1) and (2) above in a privacy-preserving manner. However, that has two issues:

- **Issue-1:** it clearly violates our fire-and-forget goal; i.e., we need clients to be completely non-interactive;
- **Issue-2:** using an off-the-shelf MPC for computing a zkSNARK proof π would be tremendously expensive.

[7] Unlike ZEXE (see [BCG+20, Section 6.2]) we choose not to include the order as a field of the payload in the record. Instead we allow the client to choose the order after the creation of the record and send the order (in a privacy-preserving manner) externally to the MPC. This offers more flexibility to the system. Generally, in our system the smart-contract f is allowed to take external inputs from each client that we call auxiliary inputs aux_i.

MPC Service. Our first idea is to therefore rely on a *committee* of m servers who offer *MPC-as-a-service*. The service runs an MPC protocol in place of the clients. The clients can submit their requests (including their input records and their orders) to this MPC service, who then runs an MPC protocol to compute f, the output records and the corresponding zkSNARK proof π. While this seems to mitigate **Issue-1**, i.e., the need for any interaction amongst the clients, as we will see shortly, it doesn't yet fully give us a "fire-and-forget" style solution.

The clients can secret share all of their sensitive information (e.g. their records, their exchange intents etc.) amongst the MPC servers, i.e., server j only gets the j-th share of the data. This essentially means that, if we use an m-out-of-m secret sharing scheme to compute these shares, then as long as at least one of the MPC servers is honest and non-colluding (and even if all the other servers collude), none of the servers learn the client's inputs.

Collaborative zkSNARKs. To address **Issue-2**, i.e., computing the zkSNARK proof π using an MPC protocol, our next idea is to rely on the seminal work of Ozdemir and Boneh on Collaborative zkSNARKs [OB22a]. In this work the authors provide custom MPC protocols for efficient distributed computation of zkSNARK proofs. In our DEX example, the zkSNARK that we want the servers to compute using an MPC protocol is:

- **Statement:** $(\mathsf{root}, \mathsf{sn}_1, \mathsf{sn}_2, \widetilde{\mathsf{cm}}_{1,1}, \widetilde{\mathsf{cm}}_{1,2}, \widetilde{\mathsf{cm}}_{2,1}, \widetilde{\mathsf{cm}}_{2,2}, f)$
- **Witness:** $(\mathsf{ask}_1, \mathsf{path}_1, \boldsymbol{r}_1), (\mathsf{ask}_2, \mathsf{path}_2, \boldsymbol{r}_2), \widetilde{\boldsymbol{r}}_{1,1}, \widetilde{\boldsymbol{r}}_{1,2}, \widetilde{\boldsymbol{r}}_{2,1}, \widetilde{\boldsymbol{r}}_{2,2}, \mathsf{order}_1, \mathsf{order}_2$
- **NP Relation:**

  (i) $\mathsf{F}(\mathsf{root}, \mathsf{sn}_1, \mathsf{ask}_1, \mathsf{path}_1, \boldsymbol{r}_1) = 1 \wedge \mathsf{F}(\mathsf{root}, \mathsf{sn}_2, \mathsf{ask}_2, \mathsf{path}_2, \boldsymbol{r}_2) = 1$

  (ii) $\widetilde{\mathsf{F}}(\widetilde{\mathsf{cm}}_{1,1}, \widetilde{\boldsymbol{r}}_{1,1}) = 1 \wedge \widetilde{\mathsf{F}}(\widetilde{\mathsf{cm}}_{1,2}, \widetilde{\boldsymbol{r}}_{1,2}) = 1 \wedge \widetilde{\mathsf{F}}(\widetilde{\mathsf{cm}}_{2,1}, \widetilde{\boldsymbol{r}}_{2,1}) = 1 \wedge \widetilde{\mathsf{F}}(\widetilde{\mathsf{cm}}_{2,2}, \widetilde{\boldsymbol{r}}_{2,2}) = 1$

  (iii) $f(\mathsf{pd}_1, \mathsf{pd}_2, \widetilde{\mathsf{pd}}_{1,1}, \widetilde{\mathsf{pd}}_{1,2}, \widetilde{\mathsf{pd}}_{2,1}, \widetilde{\mathsf{pd}}_{2,2}, \mathsf{order}_1, \mathsf{order}_2) = 1$

Here, F and $\widetilde{\mathsf{F}}$ are the predicates that check well-formedness of the nullifiers in old records (i.e., the records that are going to be consumed in this transaction) and the commitments of the new records that will be generated, respectively. We defer the details of F and $\widetilde{\mathsf{F}}$ to the technical sections. What is important to highlight here is that these predicates check cryptographic relationship between the inputs – for instance, F contains many sequential hash computations for asserting membership in the Merkle tree of commitments.

Recall that we want the MPC servers to collaboratively compute such zkSNARKs. Distributed computation of zkSNARKs for such large relations using off-the-shelf constructions of collaborative zkSNARKs [OB22a, tc24] poses a significant challenge for us. That is, all existing collaborative zkSNARK solutions require very large communication (proportional to the size of the relation) between the participants and, more importantly, requires them to interact over several rounds. The number of rounds grows with the multiplicative depth of the zkSNARK circuit; for example, the Merkle tree verification alone results in thousands of communication rounds (even for favorable choices of hash functions). Running such high communication protocols over low-bandwidth, high-latency networks is impractical.

2.2 Our Solution

The discussion so far highlights that the primary efficiency bottleneck in realizing this approach with MPC servers lies in the computation of collaborative zkSNARKs. To address this, we propose a novel primitive called *Jigsaw*, which enables a more efficient implementation of collaborative zkSNARKs.

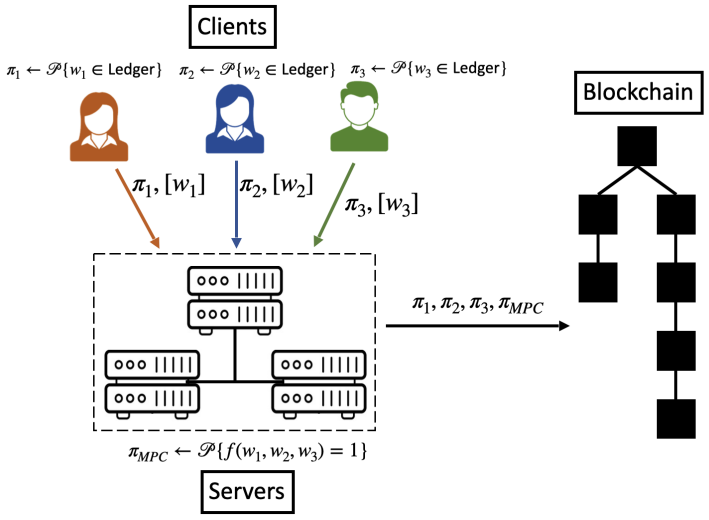

Figure 1: System Architecture of Jigsaw

Core Observation. A key insight is that part of the NP relation for this zkSNARK depends solely on "local data" held by individual clients. Specifically, the predicate F, which verifies the well-formedness of an input record, can be computed entirely by the client. Our first key idea is to delegate this portion of the computation to the client.

To avoid back-and-forth interaction between clients and servers, we propose "splitting" the NP relation into multiple sub-relations and computing separate zkSNARKs for each:

- *Local zkSNARKs:* These zkSNARKs validate relations that depend exclusively on a single client's input. Each client independently computes their local zkSNARK before submitting their request to the MPC service.
- *Collaborative zkSNARK:* This zkSNARK validates the combined relation that depends on all clients' inputs. It is computed collaboratively by the MPC servers.

As a result, instead of a single zkSNARK certifying the final transaction's validity (for on-chain publication), we use a set of zkSNARKs: one local zkSNARK per client and a collaborative zkSNARK. For instance, in the example of DEX swap, the set of proofs is $\pi := (\pi_1, \pi_2, \pi_{\mathsf{MPC}})$. Notably, the computationally intensive portion of the relation is offloaded to the local zkSNARKs, leaving a smaller, more efficient relation for the collaborative zkSNARK, which significantly reduces the communication overhead for MPC servers. The end-to-end system is illustrated in Figure 1.

A reader might question whether this set of independently generated zkSNARKs can safely replace a single zkSNARK proving the entire relation. Indeed, the sub-relations proved by the constituent zkSNARKs are interdependent. Thus, it is essential to ensure consistency among them.

For instance, $\mathsf{F}(\mathsf{root}, \mathsf{sn}_1, \mathsf{ask}_1, \mathsf{path}_1, \boldsymbol{r}_1) = 1$ and $f(\mathsf{pd}_1, \mathsf{pd}_2, \widetilde{\mathsf{pd}}_{1,1}, \widetilde{\mathsf{pd}}_{1,2}, \widetilde{\mathsf{pd}}_{2,1}, \widetilde{\mathsf{pd}}_{2,2}, \mathsf{order}_1, \mathsf{order}_2) = 1$ both take pd_1 as input. To ensure correctness, we must guarantee that the same pd_1 is used in both zkSNARKs. This requires a mechanism to "bind" the proofs together.

Commit-and-Prove zkSNARKs. To achieve this, we employ the well-known commit-and-prove technique [CLOS02] for zkSNARK composition [CFQ19]. This approach securely links zkSNARKs operating

on shared data by: (1) committing to the shared data in $\mathsf{cm}_{\mathsf{cp}}$, and (2) opening $\mathsf{cm}_{\mathsf{cp}}$ within each zkSNARK. Using this approach, we can be assured that the same pd_1 is used in both F and f.

With these cryptographic tools in place, we proceed to describe our construction in detail.

2.2.1 A Straw-man Approach

Before describing our actual construction, we find it instructive to first present a baseline approach and discuss its limitations.

In this approach, each client i computes their local zkSNARK proof π_i to prove the well-formedness of their record. To enable commit-and-prove style composition, commitments to the data shared between multiple zkSNARKs are also included, highlighted in blue below.

Local zkSNARK for Client 1:

- Statement: $(\mathsf{root}, \mathsf{sn}_1, \mathsf{cm}_{\mathsf{cp},1})$
- Witness: $(\mathsf{ask}_1, \mathsf{path}_1, \boldsymbol{r}_1, \underline{\mathsf{cp}_1})$
- NP Relation: (i-a) $\mathsf{F}(\mathsf{root}, \mathsf{sn}_1, \mathsf{ask}_1, \mathsf{path}_1, \boldsymbol{r}_1) = 1 \wedge$ (iv-a) $\mathsf{cm}_{\mathsf{cp},1} = \mathsf{Com}(\mathsf{pd}_1; \underline{\mathsf{cp}_1})$

Local zkSNARK for Client 2:

- Statement: $(\mathsf{root}, \mathsf{sn}_2, \mathsf{cm}_{\mathsf{cp},2})$
- Witness: $(\mathsf{ask}_2, \mathsf{path}_2, \boldsymbol{r}_2, \underline{\mathsf{cp}_2})$
- NP Relation: (i-b) $\mathsf{F}(\mathsf{root}, \mathsf{sn}_2, \mathsf{ask}_2, \mathsf{path}_2, \boldsymbol{r}_2) = 1 \wedge$ (iv-b) $\mathsf{cm}_{\mathsf{cp},2} = \mathsf{Com}(\mathsf{pd}_2; \underline{\mathsf{cp}_2})$

Both clients send their local zkSNARKs, their nullifiers, CP commitments, and secret shares of their records and orders to the MPC servers.

Given these inputs pd_1, pd_2, order_1, order_2, the MPC service first computes the output records as follows:

1. Computes $\widetilde{\mathsf{pd}}_{1,1}, \widetilde{\mathsf{pd}}_{1,2}, \widetilde{\mathsf{pd}}_{2,1}, \widetilde{\mathsf{pd}}_{2,2}$ such that $f(\mathsf{pd}_1, \mathsf{pd}_2, \widetilde{\mathsf{pd}}_{1,1}, \widetilde{\mathsf{pd}}_{1,2}, \widetilde{\mathsf{pd}}_{2,1}, \widetilde{\mathsf{pd}}_{2,2}, \mathsf{order}_1, \mathsf{order}_2) = 1$.
2. For $i, j \in \{1, 2\}$: Samples $\widetilde{\mathsf{ask}}_{i,j}$, $\widetilde{\rho}_{i,j}$, $\underline{\widetilde{\boldsymbol{r}}_{i,j}}$, computes $\widetilde{\mathsf{apk}}_{i,j} = \mathsf{PRF}_{\widetilde{\mathsf{ask}}_{i,j}}(0)$, $\widetilde{\mathsf{cm}}_{i,j} = \mathsf{Com}(\widetilde{\mathsf{apk}}_{i,j}, \widetilde{\mathsf{pd}}_{i,j}, \widetilde{\rho}_{i,j}; \underline{\widetilde{\boldsymbol{r}}_{i,j}})$ and sets $\widetilde{\boldsymbol{r}}_{i,j} = (\widetilde{\mathsf{cm}}_{i,j}, \widetilde{\mathsf{apk}}_1, \widetilde{\mathsf{pd}}_{i,j}, \widetilde{\rho}_{i,j}, \underline{\widetilde{\boldsymbol{r}}_{i,j}})$.

After that the service computes a collaborative zkSNARK proof π_{MPC} for the following relation:

Collaborative zkSNARK:

- Statement: $(\mathsf{root}, \widetilde{\mathsf{cm}}_{1,1}, \widetilde{\mathsf{cm}}_{1,2}, \widetilde{\mathsf{cm}}_{2,1}, \widetilde{\mathsf{cm}}_{2,2}, f, \mathsf{cm}_{\mathsf{cp},1}, \mathsf{cm}_{\mathsf{cp},2})$
- Witness: $\widetilde{\boldsymbol{r}}_{1,1}, \widetilde{\boldsymbol{r}}_{1,2}, \widetilde{\boldsymbol{r}}_{2,1}, \widetilde{\boldsymbol{r}}_{2,2}, \mathsf{order}_1, \mathsf{order}_2, \underline{\mathsf{cp}_1}, \underline{\mathsf{cp}_2}$
- NP Relation:

(ii) $\widetilde{\mathsf{F}}(\widetilde{\mathsf{cm}}_{1,1}, \widetilde{\boldsymbol{r}}_{1,1}) = 1 \wedge \widetilde{\mathsf{F}}(\widetilde{\mathsf{cm}}_{1,2}, \widetilde{\boldsymbol{r}}_{1,2}) = 1 \wedge \widetilde{\mathsf{F}}(\widetilde{\mathsf{cm}}_{2,1}, \widetilde{\boldsymbol{r}}_{2,1}) = 1 \wedge \widetilde{\mathsf{F}}(\widetilde{\mathsf{cm}}_{2,2}, \widetilde{\boldsymbol{r}}_{2,2}) = 1$

(iii) $f(\mathsf{pd}_1, \mathsf{pd}_2, \widetilde{\mathsf{pd}}_{1,1}, \widetilde{\mathsf{pd}}_{1,2}, \widetilde{\mathsf{pd}}_{2,1}, \widetilde{\mathsf{pd}}_{2,2}, \mathsf{order}_1, \mathsf{order}_2) = 1$

(iv) $\mathsf{cm}_{\mathsf{cp},1} = \mathsf{Com}(\mathsf{pd}_1; \underline{\mathsf{cp}_1}) \wedge \mathsf{cm}_{\mathsf{cp},2} = \mathsf{Com}(\mathsf{pd}_2; \underline{\mathsf{cp}_2})$

Finally, the service outputs $\pi_{\mathsf{MPC}}, \widetilde{\mathsf{cm}}_{1,1}, \widetilde{\mathsf{cm}}_{1,2}, \widetilde{\mathsf{cm}}_{2,1}, \widetilde{\mathsf{cm}}_{2,2}$.

This approach fails to meet our goals for several reasons. First, allowing the MPC service to sample secret keys for the output records risks compromising security if all MPC servers collude. Instead, as discussed in Section 1.1, we want to settle for the guarantee where, if *all* MPC servers collude, they can violate privacy of the system, but never its integrity. The second issue with this approach is that after the servers are done with the computation, the clients must return to collect their secret keys from the MPC service, violating the desired "fire-and-forget" property. In summary, we cannot let the MPC servers generate output records for the smart contract *from scratch* on behalf of the clients.

2.2.2 Our Approach

Our idea is to let the clients generate new records with uniformly random payloads; we refer to these as "tentative records". These tentative records with uniformly random payloads are secret shared with the MPC servers at the time of submitting their requests. Upon receiving requests from all clients, the MPC servers compute the actual payloads for the output records and use the additive homomorphic property of the commitment scheme update the payloads of the output commitments in the tentative records. Importantly, (unlike the "Strawman Approach" above), this approach prevents the MPC servers from learning the secret keys of the output records even if they all collude.

In more detail, each client $i \in \{1,2\}$ now samples $\widetilde{\mathsf{ask}}_{i,1}$, $\widetilde{\rho}_{i,1}$, $\underline{\widetilde{\boldsymbol{r}}_{i,1}}$, and an additionally uniformly random $\widetilde{\mathsf{pd}}^*_{i,1}$, computes $\widetilde{\mathsf{apk}}_{i,1} = \mathsf{PRF}_{\widetilde{\mathsf{ask}}_{i,1}}(0)$ and sets the tentative record commitment to be $\widetilde{\mathsf{cm}}^*_{i,1} = \mathsf{Com}(\widetilde{\mathsf{apk}}_{i,1}, \widetilde{\mathsf{pd}}^*_{i,1}, \widetilde{\rho}_{i,1}; \underline{\widetilde{\boldsymbol{r}}_{i,1}})$. The tentative record is defined as $\widetilde{\boldsymbol{r}}^*_{i,1} = (\widetilde{\mathsf{cm}}^*_{i,j}, \widetilde{\mathsf{apk}}_{i,1}, \widetilde{\mathsf{pd}}^*_{i,1}, \widetilde{\rho}_{i,1}, \underline{\widetilde{\boldsymbol{r}}_{i,1}})$. They repeat this process for their second output record $\widetilde{\boldsymbol{r}}^*_{i,2} = (\widetilde{\mathsf{cm}}^*_{i,2}, \widetilde{\mathsf{apk}}_{i,2}, \widetilde{\mathsf{pd}}^*_{i,2}, \widetilde{\rho}_{i,2}, \underline{\widetilde{\boldsymbol{r}}_{i,2}})$.

As before, the clients send their nullifiers as well as secret shares of the input records to the MPC servers. Additionally, they also send their tentative output commitments and secret shares of the corresponding output records to the MPC servers. The MPC servers then proceed as follows:

1. Compute $\widetilde{\mathsf{pd}}_{1,1}, \widetilde{\mathsf{pd}}_{1,2}, \widetilde{\mathsf{pd}}_{2,1}, \widetilde{\mathsf{pd}}_{2,2}$ such that $f(\mathsf{pd}_1, \mathsf{pd}_2, \widetilde{\mathsf{pd}}_{1,1}, \widetilde{\mathsf{pd}}_{1,2}, \widetilde{\mathsf{pd}}_{2,1}, \widetilde{\mathsf{pd}}_{2,2}, \mathsf{order}_1, \mathsf{order}_2) = 1$.
2. For $i,j = \{1,2\}$: Compute $\delta_{i,j} = \widetilde{\mathsf{pd}}_{i,j} - \widetilde{\mathsf{pd}}^*_{i,j}$ and set $\widetilde{\mathsf{cm}}_{i,j}$ as $\widetilde{\mathsf{cm}}_{i,j} = \widetilde{\mathsf{cm}}^*_{i,j} + \mathsf{Com}(0, \delta_{i,j}, 0; 0)$.[8]

Finally, the servers compute a collaborative zkSNARK π_{MPC} and post the resulting transaction on-chain. Note that since $\delta_{i,j}$ is uniformly distributed, posting it publicly does not affect privacy.

Preventing Misuse of π_i. As a final note, we need to make sure that the MPC is actually using the input records of the clients as specified by the clients, i.e., they are used for the correct f and the correct input-order order_i. In more detail, we want to prevent the servers from maliciously using the π_i and $\boldsymbol{r}_i$ for other purposes. For this, we want to find a way to "bind" π_i to f and order_i.

Since f is public, we can simply let client i sign it together with π_i. However, we do not want to let the client generate a fixed signature public key since this would violate anonymity. Therefore, we make use of *signatures of knowledge* [CL06] that allow users to sign using knowledge of the witness corresponding to a zkSNARK relation and the corresponding statement acts as the signature public key (see Section 5.2). In practice, the local zkSNARK can also be used to implement a signature of knowledge (as long as it is simulation extractable [CL06]); resulting in zero overhead (see Section 5.5).

[8] For simplicity we use $+$ to denote the homomorphic operation over commitments. However, depending on the commitment scheme, this can be replaced with the appropriate operation. For instance, in Pedersen commitments, two commitments can be multiplied to get a commitment to a sum of the values committed inside the original commitments.

For order_i, however, the situation is slightly more complicated since it is not a public value. Therefore, to bind order_i with π_i, we choose to include it in the CP commitment $\mathsf{cm}_{\mathrm{cp},i}$ and open it inside both the local and collaborative zkSNARKs.

2.3 Additional Technical Subtleties

In the overview, we presented the core technical ideas of our protocol through a simplified example. However, to generalize the scheme for other real-world applications, several additional challenges must be addressed. These subtleties are discussed in detail in the technical sections:

- *Asynchronous orders*: Practically speaking, we cannot expect all clients to synchronize and generate local proofs with respect to the same blockchain state (specifically, the Merkle root). This challenge and its resolution is detailed in Section 5.1.
- *Sabotaging Attacks*: To prevent malicious clients from disrupting smart contract executions, we ensure that the secret shares provided by clients are consistent with the values used in their local zkSNARK proofs (see Section 5.4.2 and Section 5.5.3).
- *Commit-and-Prove PLONK*: For our concrete instantiation, we require a commit-and-prove variant of the PLONK zkSNARK [GWC19]. This involves designing an efficient opening proof for KZG-based commitments [KZG10] within PLONK. We address this by constructing a "linking proof" between KZG commitments, enabling secure composition. Further details are provided in Section 5.5.2 and Appendix B.

3 Preliminaries

Notation. Throughout this work, unless explicitly stated otherwise, we will denote a random value associated with a cryptographic operation (e.g. a commitment on x) underlined. For example $\underline{\hat{\boldsymbol{y}}_i}$ would denote the randomness corresponding to $\hat{\boldsymbol{y}}_i$.

Notation for basic Cryptographic Primitives. In our work we make use of basic cryptographic primitives such as Commitment schemes, Pseudorandom Functions and Merkle trees. We write $\mathsf{cm} = \mathsf{Com}(x_1, \ldots x_n; \underline{x})$ for a (multi-)commitment of $x_1, \ldots, x_n$ with randomness $\underline{x}$. We write $\mathsf{PRF}_{\mathsf{sk}}(x) = y$ for a pseudorandom function with secret key sk applied on x and giving output y. We denote a Merkle tree root as root, a path as path and the verification of a leaf x_i w.r.t. the root as $\mathsf{MerkleVerify}(\mathsf{root}, \mathsf{path}_i, x_i)$. $[x]$ in general refers to a secret share of x and $[x]_j$ in concrete to the j-th share.

Remark 1. *For the sake of simplicity, throughout the paper we will usually omit explicitly writing the public parameters of the cryptographic schemes, unless crucial for the description. For example in the above description we omitted the commitment key from the Com algorithm, i.e. instead of writing Com*$(\mathsf{ck}, x; \underline{x})$ *we wrote Com*$(x; \underline{x})$.

3.1 Zero Knowledge Succinct Arguments of Knowledge (zkSNARKs)

We recall the definition of zero-knowledge succinct arguments of knowledge (zkSNARKs).

Definition 1 (zkSNARKs)**.** *A zero-knowledge succinct non-interactive argument of knowledge for a family of relations R_λ, is a tuple of three algorithms* $\mathsf{zkSNARK} = (\mathsf{Setup}, \mathcal{P}, \mathcal{V})$*, defined as follows:*

$\mathsf{Setup}(\mathcal{R}) \to \mathsf{srs}$: *On input a relation $\mathcal{R} \in R_\lambda$ the setup algorithm outputs a structured reference string* srs.

$\mathcal{P}(\mathsf{srs}, \mathbb{x}, \mathbb{w}) \to \pi$: *On input the structured reference string* srs*, a statement* $\mathbb{x}$ *and a witness* $\mathbb{w}$ *the prover algorithm outputs a proof* π.

$\mathcal{V}(\mathsf{srs}, \mathbb{x}, \pi) \to 0/1$: *On input the structured reference string* srs*, a statement* $\mathbb{x}$ *and a proof* π *the verifier algorithm outputs either* 1 *for accept or* 0 *for reject.*

It is further required that the following properties hold.

Correctness. *For each $\lambda \in \mathbb{N}$, each relation $\mathcal{R} \in R_\lambda$, and every statement-witness pair $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$:*

$$\Pr\left[\mathcal{V}(\mathsf{srs}, \mathbb{x}, \pi) = 1 \;:\; \begin{array}{c} \mathsf{srs} \leftarrow \mathsf{Setup}(\mathcal{R}) \\ \pi \leftarrow \mathcal{P}(\mathsf{srs}, \mathbb{x}, \mathbb{w}) \end{array} \right] = 1$$

Knowledge Soundness. *For every PPT adversarial prover P^*, there exists a PPT extractor $\mathcal{E}_{P^*}$ such that for every security parameter $\lambda \in \mathbb{N}$, every auxiliary input $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$, and every relation $\mathcal{R} \in R_\lambda$:*

$$\Pr\left[\begin{array}{l} \mathcal{V}(\mathsf{srs}, \mathbb{x}, \pi) = 1 \\ \wedge (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} \end{array} \;:\; \begin{array}{c} \mathsf{srs} \leftarrow \mathsf{Setup}(\mathcal{R}) \\ (\mathbb{x}, \pi)^* \leftarrow \mathcal{P}^*(\mathsf{srs}, \mathsf{aux}) \\ w \leftarrow \mathcal{E}_{P^*}(\mathsf{srs}, \mathsf{aux}) \end{array} \right] = 1$$

Succinctness. *There exists a universal polynomial $p(\cdot)$ such that, for every security parameter $\lambda \in \mathbb{N}$, every relation $\mathcal{R} \in R_\lambda$, and every statement-witness pair $(\mathbb{x}, \mathbb{w})$:*

- *An honestly generated proof π has size $p(\lambda + \log |\mathbb{w}|)$.*
- *The verifier algorithm $\mathcal{V}(\mathsf{srs}, x, \pi)$ runs in time $p(\lambda + |\mathbb{x}| + \log |\mathbb{w}|)$.*

(Perfect) Zero-Knowledge. *For every security parameter $\lambda \in \mathbb{N}$ and every relation $(\mathcal{R}, \mathsf{aux}_R) \leftarrow R_\lambda$, there exists a simulator $\mathcal{S}$ such that, for every statement-witness pair $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ and for every computationally unbounded adversary $\mathcal{A}$:*

$$\begin{aligned} &\Pr\left[\mathcal{A}(R, \mathsf{aux}_R, \mathsf{srs}, \pi) = 1 \;:\; \begin{array}{c} \mathsf{srs} \leftarrow \mathsf{Setup}(\mathcal{R}) \\ \pi \leftarrow \mathcal{P}(\mathsf{srs}, \mathbb{x}, \mathbb{w}) \end{array} \right] \\ =\ &\Pr\left[\mathcal{A}(R, \mathsf{aux}_R, \mathsf{srs}, \pi) = 1 \;:\; (\mathsf{srs}, \pi) \leftarrow \mathcal{S}(x, \mathcal{R}) \right] \end{aligned}$$

3.1.1 Commit-and-Prove zkSNARKs

In this work we rely on the commit-and-prove framework for zkSNARKs [CFQ19]. A commit-and-prove zkSNARK is a zkSNARK for a relation of the type $R_{\mathrm{CP}}(\mathsf{cm}, x; \omega, u, o) = 1$[9] (i.e. $\mathbb{x} := (\mathsf{cm}, x)$ and $\mathbb{w} := (\omega, u, o)$) and the relation holds iff:

$$R_{\mathrm{CP}}(\mathsf{cm}, x; \omega, u, o) = 1 \iff \mathsf{cm} = \mathsf{Com}(u; o) \wedge \tilde{R}(x; \omega, u) = 1.$$

In plain words, the CP-zkSNARK is first 'opening' a commitment to the witness and then proves a property that involves the committed value. Observe that the witness should not necessarily be committed in its entirety (e.g. ω is not committed).

[9] Formally the relation is also parametrized by the commitment key of the commitment scheme.

Modular Composition of CP-zkSNARKs. We can modularly compose different CP-SNARK proofs for different relations in a secure manner as long as they refer to common committed witnesses. For instance, let the relation:

$$R(x_1, x_2; \omega_1, \omega_2, u) = 1 \iff \tilde{R}_1(x_1; \omega_1, u) = 1 \wedge \tilde{R}_2(x_2; \omega_2, u) = 1$$

and two different CP-zkSNARKs:

1. $\mathsf{zkSNARK}_1$ for $R_1(x_1; \omega_1, u, o) = \big(\mathsf{cm} = \mathsf{Com}(u; o)\big) \wedge \tilde{R}_1(x_1; \omega_1, u)$
2. $\mathsf{zkSNARK}_2$ for $R_2(x_2; \omega_2, u, o) = \big(\mathsf{cm} = \mathsf{Com}(u; o)\big) \wedge \tilde{R}_2(x_2; \omega_1, u)$

that generate the corresponding zkSNARK proofs π_1 and π_2. Then, according to the composition theorem of [CFQ19], $\pi := (\mathsf{cm}, \pi_1, \pi_2)$ is a zkSNARK proof for $R(x_1, x_2; \omega_1, \omega_2, u) = 1$.

In plain words, we can prove two distinct properties of u with two different CP-zkSNARKs that share a common commitment of u.

3.2 Collaborative zkSNARKs

A *collaborative zk-SNARK* is a non-interactive proof system in which multiple mutually distrustful provers compute a proof from their combined witnesses in a way that preserves zero-knowledge against an adversarial subset of $\leq t$ provers. Unless otherwise indicated, the definitions in this section come almost verbatim from [OB22b].

Relation. For any language L in NP, a **relation** R_L for L is a set of instance-witness pairs $(\mathcal{X}, \mathsf{wtn})$ such that $\mathcal{X} \in L$, and wtn is a witness for $\mathcal{X}$. Furthermore, there is a polynomial-time function V_R to verify the witness. For all $(\mathcal{X}, \mathsf{wtn}) \in R$: $V_R(\mathcal{X}, \mathsf{wtn}) = 1$. And for all $\mathcal{X} \notin L$, there does not exist a wtn such that $V_R(\mathcal{X}, \mathsf{wtn}) = 1$. Finally, the size of the relation, $|R|$, is the time needed to compute $V_R(\mathcal{X}, \mathsf{wtn})$ (or the size of the circuit that computes it).

Random oracle. Let $\mathsf{poly}(\lambda)$ be a polynomial upper bound on the total communication of the proof system. Let $\mathcal{U}(\lambda) := \{H | H : \{0,1\}^{\leq \mathsf{poly}(\lambda)} \to \{0,1\}^{\lambda}\}$. A random oracle H is a function sampled uniformly at random from $\mathcal{U}(\lambda)$. H can be reprogrammed on certain inputs. The reprogrammed oracle is written as $H[\mu]$, where μ specifies the reprogrammed points. Specifically, μ is a partial function whose domain is a subset of Domain(H). $H[\mu]$ maps x to $\mu(x)$ if $x \in$ Domain(μ) and maps x to $H(x)$ if $x \notin$ Domain(μ).

Protocol Syntax. A collaborative zk-SNARK in the random oracle model is a protocol among N provers $\boldsymbol{\mathcal{P}} = (\mathcal{P}_1, \ldots, \mathcal{P}_N)$ and a verifier $\mathcal{V}$. Each prover $\mathcal{P}_i$ has a witness fragment wtn_i, and the witness fragments combine to form the witness: $\mathsf{wtn} = (\mathsf{wtn}_1, \ldots, \mathsf{wtn}_N)$. The collaborative zk-SNARK comprises the following three sub-protocols (each of them can query RO H):

- $\mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \to \mathsf{pp}$: Generates the public parameters pp.
- $\mathsf{Prove}^H(\mathsf{pp}, \mathcal{X}, \mathsf{wtn}) \to \{\pi, \perp\}$: If $(\mathcal{X}, \mathsf{wtn}) \in R$, then it outputs a proof π. Otherwise, it outputs $\perp$. Prove is run by the provers.
- $\mathsf{Verify}^H(\mathsf{pp}, \mathcal{X}, \pi) \to \{0,1\}$: Verifies π. It outputs 1 to indicate acceptance and 0 to indicate rejection. Verify is run by the verifier.

Definition 2. *[OB22a] In random oracle model, a collaborative zk-SNARK for a relation $\mathfrak{R}$ secure against t malicious provers is a collaborative proof $(\mathsf{Setup}^H, \mathsf{Prove}^H, \mathsf{Verify}^H)$ satisfying the following properties:*

Completeness: *For any $(\mathcal{X}, \mathsf{wtn}) \in R$, the verifier will accept the proof with overwhelming probability. In other words, for any $(\mathcal{X}, \mathsf{wtn}) \in R$, there is a negligible function $\varepsilon(\lambda)$ such that:*

$$\Pr\left[\mathsf{Verify}^H(\mathsf{pp}, \mathcal{X}, \pi) = 0 : \begin{array}{c} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ \pi \leftarrow \mathsf{Prove}^H(\mathsf{pp}, \mathcal{X}, \mathsf{wtn}) \end{array}\right] = \varepsilon(\lambda)$$

Knowledge Soundness: *Informally, knowledge soundness says that if the provers can convince the verifier to accept with some probability, then there is an extractor that can extract from the provers a valid witness wtn, with similar probability.*

Formally, the protocol has knowledge soundness if for any $\mathcal{X}$ and any efficient provers $\boldsymbol{\mathcal{P}} = (\mathcal{P}_1, \ldots, \mathcal{P}_N)$, there exists an efficient extractor Ext with the following properties: Ext gets oracle access to H. Ext can run the provers multiple times, reprogramming H each time, and Ext receives only the provers' final output from each run. This is denoted as $\mathsf{Ext}^{H, \vec{\mathcal{P}}^H}$. Finally, there exists a negligible function $\varepsilon(\lambda)$ such that:

$$\Pr\left[(\mathcal{X}, \mathsf{wtn}) \in R : \begin{array}{c} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ \mathsf{wtn} \leftarrow \mathsf{Ext}^{H, \vec{\mathcal{P}}^H}(\mathsf{pp}, \mathcal{X}) \end{array}\right] \geq \Pr\left[\mathsf{Verify}^H(\mathsf{pp}, \mathcal{X}, \pi) = 1 : \begin{array}{c} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ \pi \leftarrow \vec{\mathcal{P}}^H(\mathsf{pp}, \mathcal{X}) \end{array}\right] - \varepsilon(\lambda)$$

Succinctness: *The size of π and the runtime of $\mathsf{Verify}^H(\mathsf{pp}, \mathcal{X}, \pi)$ are $o(|R|)$.*

t-Zero-Knowledge: *t-zero-knowledge ensures that no subset of t parties or fewer will learn anything about the witnesses of the other parties, except whether the combined witness is valid. Formally, a collaborative zk-SNARK satisfies t-zero-knowledge if for any PPT malicious adversary $\mathcal{A}$ controlling $k \leq t$ provers $(\mathcal{P}_{i_1}, \ldots, \mathcal{P}_{i_k})$, there exists an efficient simulator Sim such that for all $(\mathcal{X}, \mathsf{wtn})$, and for all efficient distinguishers D, there is a negligible function $\varepsilon(\lambda)$ such that:*

$$\left|\Pr\left[D^{H[\mu]}(\mathsf{tr}) = 1 : \begin{array}{c} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ b \leftarrow V_{\mathfrak{R}}(\mathcal{X}, \mathsf{wtn}) \in \{0,1\} \\ (\mathsf{tr}, \mu) \leftarrow \mathsf{Sim}^H(\mathsf{pp}, \mathcal{X}, \mathsf{wtn}_{i_1}, \ldots, \mathsf{wtn}_{i_k}, b) \end{array}\right] - \Pr\left[D^H(\mathsf{tr}) = 1 : \begin{array}{c} H \xleftarrow{\$} \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathfrak{R}) \\ \mathsf{tr} \leftarrow \mathsf{View}_{\mathcal{A}}^H[\mathcal{X}, \mathsf{wtn}] \end{array}\right]\right| = \varepsilon(\lambda)$$

Here, tr is the protocol transcript, which lists the messages sent among the parties. And $\mathsf{View}_{\mathcal{A}}^H[\mathcal{X}, \mathsf{wtn}]$ is $\mathcal{A}$'s view of the protocol.

4 Doubly Private Smart Contracts

In this section, we present a formal definition of our framework for doubly private smart contracts. Our definitions are adapted from the definition of decentralized private computation from [BCG+20]. Consequently, much of the text in this section is taken verbatim from [BCG+20].

4.1 Syntax and Algorithms

Data Structures. As in any smart contract, we consider three main data structures: records, transactions and the ledger.

- **Records:** A record, denoted by the symbol $\boldsymbol{r}$, is a data structure representing a unit of data. Records can be created or consumed, and these events denote state changes in the system. $\boldsymbol{r}$ has the following attributes:

  1. A commitment cm which binds together all other attributes of $\boldsymbol{r}$ while hiding all information about them.
  2. An address public key apk, which specifies the record's owner.
  3. A payload pd containing arbitrary application-dependent information.
  4. Any other construction-specific information.

  Informally speaking, the record $\boldsymbol{r}$ is created when its commitment cm is posted on the ledger as part of the transaction. The record $\boldsymbol{r}$ is consumed when its serial number sn appears on the ledger as part of a later transaction. To consume $\boldsymbol{r}$, one must also know the address secret key ask corresponding to $\boldsymbol{r}$'s address public key apk because the serial number sn to be revealed can only be computed from $\boldsymbol{r}$ and ask.

  $$\boldsymbol{r} = (\mathsf{cm}, \mathsf{apk}, \mathsf{pd}, \rho, \underline{\boldsymbol{r}})$$

  where

  $$\mathsf{cm} = \mathsf{Com}(\mathsf{apk}, \mathsf{pd}, \rho; \underline{\boldsymbol{r}}),\ \mathsf{apk} = \mathsf{PRF}_{\mathsf{ask}}(0)\ \&\ \mathsf{sn} = \mathsf{PRF}_{\mathsf{ask}}(\rho)$$

- **Transactions:** A transaction, denoted by the symbol tx, is a data structure representing a state change that involves the consumption and creation of records. It has the following attributes:

  1. $(\mathsf{sn}_i)_{i=1}^{\ell}$ is the list of serial numbers of the ℓ old records.
  2. $(\mathsf{cm}_i)_{i=1}^{k}$ is the list of commitments of the k new records.
  3. memo is an arbitrary string associated with the transaction.
  4. $\star$ is other construction-specific information.

  Anyone can assemble a transaction and append it to the ledger, provided that it is well-formed. All transactions reveal the number of old records ℓ and the number of new records k.

- **Ledger:** We consider a model where all parties have access to an append-only ledger, denoted $\mathbf{L}$, that stores all published transactions. Only a valid transaction can be appended to the ledger. The ledger exposes the following interface:

  1. $\mathbf{L}.\mathsf{Len}$: Return the number of transactions currently on the ledger.
  2. $\mathbf{L}.\mathsf{Push}(\mathsf{tx})$: Append a (valid) transaction tx to the ledger.
  3. $\mathbf{L}.\mathsf{Digest} \to \mathsf{st}_{\mathbf{L}}$: Return a (short) digest of the current state of the ledger.
  4. $\mathbf{L}.\mathsf{ValidateDigest}(\mathsf{st}_{\mathbf{L}}) \to b$: Check that $\mathsf{st}_{\mathbf{L}}$ is a valid digest for some (past) ledger state.
  5. $\mathbf{L}.\mathsf{Contains}(tx) \to b$: Determine if tx (or a subcomponent thereof) appears on the ledger or not.
  6. $\mathbf{L}.\mathsf{Prove}(\mathsf{tx}) \to \mathbb{W}_{\mathbf{L}}$: If a transaction tx (or a subcomponent thereof) appears on the ledger, return a proof of membership $\mathbb{W}_{\mathbf{L}}$ for it. If there are duplicates, return a proof for the lexicographically first one.
  7. $\mathbf{L}.\mathsf{Verify}(\mathsf{st}_{\mathbf{L}}, \mathsf{tx}, \mathbb{W}_{\mathbf{L}}) \to b$: Check that $\mathbb{W}_{\mathbf{L}}$ certifies that tx (or a subcomponent thereof) is in a ledger with digest $\mathsf{st}_{\mathbf{L}}$.

Only valid transactions can be appended to the ledger. The commitments and serial numbers in a transaction do not already appear on the ledger.

Algorithms/Protocols. Let $C_1 \ldots, C_n$ be a set of clients/users and $S_1, \ldots, S_m$ be the set of servers. A doubly private smart contract comprises of a tuple of the following algorithms/protocols (some of which may read information from the ledger $\mathbf{L}$):

$$\left(\mathsf{Setup}, \mathsf{GenAddress}, f, \Pi^{f,\mathbf{L}}_{\mathsf{Execute}}, \mathsf{Verify}^{f,\mathbf{L}}\right)$$

- $\mathsf{Setup}(1^\lambda) \rightarrow \mathsf{pp}$: On input the security parameter 1^λ, Setup outputs public parameters pp for the system. A trusted third party runs this algorithm once and then publishes its output; afterwards the trusted party is not needed anymore.
- $\mathsf{GenAddress}(\mathsf{pp}) \rightarrow \mathsf{apk}, \mathsf{ask}$: On input public parameters pp, GenAddress outputs an address key pair apk, ask, where apk is the address public key and ask is the corresponding secret key. Any client may run this algorithm to create an address key pair. Each record is bound to an address public key and the corresponding secret to consume it.
- $f\left((\mathsf{pd}_i)_{i=1}^{\ell}, \left(\widetilde{\mathsf{pd}}_j\right)_{j=1}^{k}, \mathsf{aux}\right) \rightarrow b$ Given as input a list of payloads $(\mathsf{pd}_i)_{i=1}^{\ell}$ from the consumed records, a list of payloads $\left(\widetilde{\mathsf{pd}}_j\right)_{j=1}^{k}$ for the new records, and private auxiliary input, the smart contract predicate f outputs a bit $b \in \{0, 1\}$.
- $\Pi^{f,\mathbf{L}}_{\mathsf{Execute}}\left(\mathsf{pp}, (\boldsymbol{r}_i)_{i=1}^{\ell}, (\mathsf{ask}_i)_{i=1}^{\ell}, \left(\widetilde{\mathsf{apk}}_j\right)_{j=1}^{k}, \left(\widetilde{\mathsf{pd}}_j\right)_{j=1}^{k}, \mathsf{aux}, \mathsf{memo}\right) \rightarrow (\boldsymbol{r}_j)_{j=1}^{k}, \mathsf{tx}$: Given as input a list of old records $(\boldsymbol{r}_i)_{i=1}^{\ell}$ with corresponding secret keys $(\mathsf{ask}_i)_{i=1}^{\ell}$, attributes for new records, private auxiliary input aux, and an arbitrary transaction memorandum memo, this protocol produces new records $(\widetilde{\boldsymbol{r}}_j)_{j=1}^{k}$ and a transaction tx. The transaction attests that f on input $\left(\widetilde{\mathsf{pd}}_j\right)_{j=1}^{k}$, the payloads in $(\boldsymbol{r}_i)_{i=1}^{\ell}$ and aux outputs 1. The servers subsequently post tx on the ledger by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx})$. This protocol is executed between the clients (associated with the input and output records) and the servers $S_1, \ldots, S_m$. Each client in this protocol sends a single message to every server, after which the servers interact among themselves to collaboratively compute the output records and the transaction.
- $\mathsf{Verify}^{f,\mathbf{L}}(\mathsf{pp}, \mathsf{tx}) \rightarrow b$: On input public parameters pp and a transaction tx, and given oracle access to the ledger $\mathbf{L}$, this algorithm outputs a bit b denoting whether the transaction tx is valid relative to the smart contract predicate f and the ledger $\mathbf{L}$.

4.2 Security

At a high level, a doubly private smart contracts scheme must satisfy the following security properties:

- *Execution Correctness:* Malicious clients and servers cannot create valid transactions if the consumed and created records do not satisfy the predicate f.
- *Execution Privacy:* Assuming that at least one of the servers is honest, the transactions reveal only the information revealed in the memorandum field, a bound on the number of consumed records, a bound on the number of created records and predicates of all involved records. All other information including the payloads remains hidden. For example, putting aside the information revealed in the memorandum

(which is arbitrary), one cannot link a transaction that consumes a record with the prior transaction that created it.

- *Consumability:* Every record can be consumed at least once and at most once by the parties that know its secrets. Thus, a malicious party cannot create two valid records for another party such that only one of them can be consumed.
- *Transaction Non-Malleability:* Malicious clients cannot modify a transaction "in flight" to the ledger.

Formally, we prove standalone security against static corruptions, in a model where every client has private anonymous communication channels to all servers. In particular, we capture security of a doubly private smart contract scheme via a simulation-based security definition.

Definition 3. *A doubly private smart contract scheme DPSC is secure if for every efficient real-world adversary $\mathcal{A}$, there exists an efficient ideal-world simulator* Sim *such that for every efficient environment $\mathcal{E}$, the following are computationally indistinguishable:*

- *The output of $\mathcal{E}$ when interacting with the adversary $\mathcal{A}$ in a real-world execution of DPSC in a model where clients can communicate with the servers via private anonymous channels; and*
- *the output of $\mathcal{E}$ when interacting with the simulator $\mathcal{S}$ in an ideal-world execution with the ideal functionality $\mathcal{F}_{\mathsf{DPSC}}$ described below.*

Ideal Data Structures. The ideal functionality $\mathcal{F}_{\mathsf{DPSC}}$ uses ideal counterparts of a doubly private smart contract scheme's data structures. An address public key apk denotes the owner of an ideal record r, which is a tuple $(\mathsf{cm}, \mathsf{apk}, \mathsf{pd})$, where cm is its commitment, apk is its address public key and pd is its payload. The record is also associated with a unique identifier sn. $\mathsf{apk}, \mathsf{cm}, \mathsf{sn}$ must be globally unique.

Before the ideal execution begins, $\mathcal{S}$ specifies three functions $(\mathsf{SampleAddrPk}, \mathsf{SampleCm}, \mathsf{SampleSn})$ that, on input a random string, sample $(\mathsf{apk}, \mathsf{cm}, \mathsf{sn})$ respectively. When $\mathcal{F}_{\mathsf{DPSC}}$ needs to sample one of these, it invoked the respective functions.

Internal State. The ideal functionality $\mathcal{F}_{\mathsf{DPSC}}$ maintains several internal tables.

- Addr, which stores address public keys.
- $\mathsf{AddrUsers}$, which maps an address public key to the set of parties that are authorized to use it.
- $\mathsf{Records}$, which maps a record's commitment to that record's information (address public key, payload).
- $\mathsf{RecUsers}$, which maps a record's commitment to the set of parties that are authorized to consume it.
- $\mathsf{SerialNumbers}$, which maps a record's commitment to that record's unique serial number.
- State, which maps a record's commitment to that record's state, either `alive` or `dead`.

Ideal Algorithms. The ideal functionality $\mathcal{F}_{\mathsf{DPSC}}$ provides the following interface to parties.

- **Address Generation:** $\mathcal{F}_{\mathsf{DPSC}}.\mathsf{GenAddress}$ outputs a new address public key apk. $\mathcal{F}_{\mathsf{DPSC}}.\mathsf{GenAddress}[C](\mathsf{apk})$:

  1. Sample randomness r for generating address public key.
  2. If $\mathsf{apk} = \perp$ then $\mathsf{apk} \leftarrow \mathsf{SampleAddrPk}(r)$.

3. Check that apk is unique: $\mathsf{Addr}[\mathsf{apk}] = \perp$.
4. Set $\mathsf{Addr}[\mathsf{apk}] = r$.
5. If C is corrupted: set S to be the set of corrupted parties.
6. If C is honest: set $S = \{C\}$.
7. Set $\mathsf{AddrUsers}[\mathsf{apk}] = \mathsf{AddrUsers}[\mathsf{apk}] \cup S$.
8. Send to C : address public key apk.

- **Execution:** $\mathcal{F}_{\mathsf{DPSC}}$.Execute performs an execution that consumes old records and creates new records. All parties are notified that an execution has occurred, and learn the serial numbers of input records, commitments of output records, and the transaction memorandum memo. Concurrent $\mathcal{F}_{\mathsf{DPSC}}$.Execute calls are serialized arbitrarily.

$$\mathcal{F}_{\mathsf{DPSC}}.\mathsf{Execute}[\{S_j\}_{j=1}^m]\left(f, (\boldsymbol{r}_i, \mathsf{sn}_i)_{i=1}^{\ell}, \left(\widetilde{\mathsf{cm}}_j, \widetilde{\mathsf{apk}}_j, \widetilde{\mathsf{pd}}_j\right)_{j=1}^{k}, \mathsf{aux}, \mathsf{memo}\right)$$

1. For each $i \in [k]$:
   - Sample randomness r_i.
   - If $\mathsf{sn}_i = \perp$ then generate serial number $\mathsf{sn}_i \leftarrow \mathsf{SampleSn}(r_i)$.
   - Check that sn_i is unique: $\mathsf{SerialNumbers}[\mathsf{sn}_i] = \perp$.
2. For each $j \in [k]$:
   - Sample randomness r_j.
   - If $\widetilde{\mathsf{cm}}_j = \perp$ then generate commitment: $\widetilde{\mathsf{cm}}_j \leftarrow \mathsf{SampleCm}(r_j)$.
   - Check that $\widetilde{\mathsf{cm}}_j$ is unique: $\mathsf{Records}[\widetilde{\mathsf{cm}}_\mathsf{j}] = \perp$.
   - Construct record $\widetilde{\boldsymbol{r}}_j = \left(\widetilde{\mathsf{cm}}_j, \widetilde{\mathsf{apk}}_j, \widetilde{\mathsf{pd}}_j\right)$.
3. Define the local data $\mathsf{ldata} = ((\boldsymbol{r}_i)_{i=1}^{\ell}, (\mathsf{sn}_i)_{i=1}^{\ell}, (\widetilde{\boldsymbol{r}}_j)_{j=1}^{k}, \mathsf{aux}, \mathsf{memo})$
4. For each $i \in [\ell]$:
   - Parse $\boldsymbol{r}_i$ as $(\mathsf{cm}_j, \mathsf{apk}_i, \mathsf{pd}_i)$.
   - Check that, for some randomness $\underline{\boldsymbol{r}_i}$, old record $\boldsymbol{r}_i$ exists: $((\mathsf{cm}_j, \mathsf{apk}_i, \mathsf{pd}_i), r_i) = \mathsf{Records}[\mathsf{cm}_i]$.
   - Check that the client C is authorized to use apk_i: $C \in \mathsf{AddrUsers}[\mathsf{apk}_i]$.
5. If the function $f\left((\mathsf{pd}_i)_{i=1}^{\ell}, \left(\widetilde{\mathsf{pd}}_j\right)_{j=1}^{k}, \mathsf{aux}\right) = 1$:
   - For each $i \in [\ell]$, mark $\boldsymbol{r}_i$ as consumed: $\mathsf{State}[\mathsf{cm}_i] = \text{dead}$.
   - For each $j \in [K]$, Insert new record $\widetilde{\boldsymbol{r}}_j$: $\mathsf{Records}[\widetilde{\mathsf{cm}}_j] = ((\widetilde{\mathsf{apk}}_j, \widetilde{\mathsf{pd}}_j), \underline{\widetilde{\boldsymbol{r}}}_j)$ and mark it as unconsumed: $\mathsf{State}[\widetilde{\mathsf{cm}}_j] = \text{alive}$. Send $\widetilde{\boldsymbol{r}}_j$ to the owner of the record. Also compute secret shares of $\widetilde{\mathsf{pd}}_j$ and send these shares to the servers.
6. If all servers $(S_i)_{i=1}^m$ are corrupted, send $\left((\boldsymbol{r}_i, \mathsf{sn}_i)_{i=1}^{\ell}, \left(\widetilde{\mathsf{cm}}_j, \widetilde{\mathsf{apk}}_j, \widetilde{\mathsf{pd}}_j\right)_{j=1}^{k}, \mathsf{aux}, \mathsf{memo}\right)$ to the adversary.
7. If all servers $(S_i)^m$ are honest, send $(f, \mathsf{Execute}, (\mathsf{sn}_i)_{i=1}^{\ell}, (\mathsf{cm}_j)_{j=1}^{k}, \mathsf{memo})$ to all servers and clients. Else, send $(f, \mathsf{Execute}, (\mathsf{sn}_i)^{\ell}, (\mathsf{cm}_j)^{k}, \mathsf{memo})$ to the adversary, if the adversary returns abort, then send $(f, \perp)$ to all parties, otherwise, send $(f, \mathsf{Execute}, (\mathsf{sn}_i)_{i=1}^{\ell}, (\mathsf{cm}_j)_{j=1}^{k}, \mathsf{memo})$ to all honest servers and clients.

5 Jigsaw: Our Construction

In this section we formally present our doubly private smart contract protocol, *Jigsaw*. For a high-level description we refer to Section 2.

Our protocol builds on the design of of Zcash [BSCG+14] and ZEXE [BCG+20]. Specifically, the ledger state is represented as the root of a Merkle tree, denoted root, which is derived by Merkle hashing all active commitments cm of records. Merkle paths path are used as proofs of membership within the ledger. For brevity, we do not restate the ledger data structure or other architectural details previously discussed in Section 4. Instead, we focus on the $\pi^{f,\mathbf{L}}_{\mathsf{Execute}}$ and $\mathsf{Verify}^{f,\mathbf{L}}$ algorithms, which constitute our core contributions.

Our system consists of n clients and a cluster of m servers that offer MPC as a service. The clients want to execute a smart contract where, in principle, each user has an arbitrary number of input and output records. However, allowing for arbitrary inputs and outputs quickly overwhelms the notation. For simplicity, we assume that each client has one input and one output. Generalizations to arbitrary numbers of input and output records are straightforward.

5.1 Design Choices

We outline some general design choices made in *Jigsaw*.

MPC vs Client Computation Tradeoff. Communication is often the primary bottleneck in high-latency, low-bandwidth environments like blockchains. To optimize the overall runtime, our approach minimizes communication between MPC servers, even at the cost of increased local computation by clients. This tradeoff enables deploying MPC services at scale, focusing primarily on privacy rather than outsourcing computation. While alternate tradeoffs – such as designing a more complex MPC service capable of both privacy and computational outsourcing – could be explored, this is beyond our scope and makes for interesting future research.

Asynchronous Clients. Our system supports asynchronous client orders, meaning a client's order can be matched with another order arriving later. Since the ledger state (i.e., the Merkle tree root) may change between these orders, we allow orders corresponding to different roots to be included in the same transaction, provided all roots have been validated previously. For example, the zkSNARK proof of client 1 may use root_1 as its statement, and client 2's proof may use root_2. Similar to zCash, our on-chain verifier is responsible for storing a sliding window of historical roots. The application can specify the size of the set.

On Pending Orders. In applications such as exchanges, an order (and its asset) may remain pending for an extended period of time. We let the application determine how to treat such orders, considering some potential options below:

- client submits an out-of-band request asking the MPC servers to cancel (i.e. delete) the previous order;
- application adopts a deterministic strategy, wherein an output record returns the asset back to the client.[10]

Note that these options have different privacy-efficiency tradeoffs that is application-dependent, and out of scope for the general framework. Moreover, in all of these options, the client relies on the servers for availability (i.e., service), which is simply inherent in our framework.

[10] For instance, an auction can terminate with output records that return assets back to all losing bidders; similarly, an exchange can periodically cancel pending orders and return assets back to the clients.

5.2 Building Blocks

Recall from Section 4 that we need the following building blocks for describing records and transactions:

- Com_1: An additively homomorphic commitment scheme.
- PRF: A pseudorandom function.
- $\mathsf{MerkleTree}$: A Merkle tree based commitment scheme.

Additionally, our construction will make use of the following cryptographic primitives:

- $\mathsf{zkSNARK}$: A Commit-and-Prove zkSNARK (see Section 3.1).
- NIZK: A non-interactive zero-knowledge proof, which is essentially a zkSNARK without the succintness property.
- $\mathsf{colzkSNARK}$: A Commit-and-Prove collaborative zkSNARK (see Section 3.2).
- An m-out-of-m secret sharing scheme. We use $\mathsf{SS}(x) = ([x]_1, \ldots, [x]_m)$ to denote the sharing algorithm. Here $[x]_i$ denotes the share that will be sent to the ith party. We will sometimes use $[x]$ to denote the entire set of shares $([x]_1, \ldots, [x]_m)$.
- MPC: A secure multi-party computation protocol that achieve security abort against malicious corruptions of all but one parties.
- Com_2: A commitment scheme (that doesn't necessarily have to be the same as the one used in the records).
- SoK: A Signature of Knowledge scheme [CL06]for NP relations is essentially a signature scheme designed to work over a relation $R(\mathbb{x}, \mathbb{w})$. In this scheme, any statement $\mathbb{x}$ in the relation R serves as the public key, while the corresponding witness $\mathbb{w}$ acts as the secret key. We denote the signing operation as $\sigma = \mathsf{SoK.Sign}_{\mathbb{w}}(m)$, where m is the message being signed and $\mathbb{w}$ is witness/secret key. Verification of the signature is written as $\mathsf{SoK}.\mathcal{V}_{\mathbb{x}}(\sigma, m)$ which uses the statement $\mathbb{x}$ as the public key, ensuring $R(\mathbb{x}, \mathbb{w}) = 1$.

  We will additionally require the signature of knowledge to satisfy succinctness and simulation extractability [GM17].

5.3 zkSNARK Relations

In this section, we describe the relevant NP relations that will be used in our construction. We start by identifying the overall statement that must be satisfied to ensure the validity of a transaction. This relation is summarized below:

- $$R\Big(\big\{\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i\big\}_{i=1}^n, f\ ;\ \big\{\mathsf{path}_i, \mathsf{cm}_i, \mathsf{apk}_i, \mathsf{ask}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i}, \widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i, \widetilde{\rho}_i, \underline{\widetilde{\boldsymbol{r}}_i}, \mathsf{aux}_i\big\}_{i=1}^n\Big) = 1 \iff$$
  $$\Big\{\mathsf{MerkleVerify}(\mathsf{root}_i, \mathsf{path}_i, \mathsf{cm}_i) = 1\Big\}_{i=1}^n$$
  $$\wedge\ \Big\{\mathsf{cm}_i = \mathsf{Com}_1(\mathsf{apk}_i, \mathsf{pd}_i, \rho_i; \underline{\boldsymbol{r}_i})\ \wedge\ \mathsf{apk}_i = \mathsf{PRF}_{\mathsf{ask}_i}(0)\ \wedge\ \mathsf{sn}_i = \mathsf{PRF}_{\mathsf{ask}_i}(\rho_i)\Big\}_{i=1}^n$$
  $$\wedge\ \Big\{\widetilde{\mathsf{cm}}_i = \mathsf{Com}_1(\widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i, \widetilde{\rho}_i; \underline{\widetilde{\boldsymbol{r}}_i})\Big\}_{i=1}^n$$
  $$\wedge\ f\Big((\mathsf{pd}_i)_{i=1}^n, (\mathsf{aux}_i)_{i=1}^n, (\widetilde{\mathsf{pd}}_i)_{i=1}^n\Big) = 1$$

Here $\left\{\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i\right\}_{i=1}^n$, f corresponds to the statement, while $\left\{\mathsf{path}_i, \mathsf{cm}_i, \mathsf{apk}_i, \mathsf{ask}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i}, \widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i, \widetilde{\rho}_i, \underline{\widetilde{\boldsymbol{r}}_i}, \mathsf{aux}_i\right\}_{i=1}^n$ corresponds to the witness. Intuitively, this relation checks validity of the following four predicates: (i) Merkle tree inclusion, (ii) well-formedness of input-coins (iii) well-formedness of output-coins and (iv) whether the input and output coins satisfy predicate f.

As discussed in Section 2, our goal is to carefully split this relation into n local relations $R_1, \ldots, R_n$, one for each client, and one collaborative relation R_{MPC} for the collaborative zkSNARK. In other words R must be equivalent to $R_{\mathsf{MPC}} \bigwedge_{i=1}^n R_i$. While there are many different ways to split this relation that offer varying trade-offs, our objective is to minimize the work of the collaborative provers. For this, we choose to put the maximum possible amount of computation in the "local" R_i's.

Observe that the only information not known a priori to the client i (before submitting their inquiry to the MPC service) is the payload of their output record, $\widetilde{\mathsf{pd}}i$, and consequently, the commitment $\widetilde{\mathsf{cm}}i$.

The client i generates a tentative record beforehand, so they know $\widetilde{\mathsf{apk}}i$, $\widetilde{\rho}i$, and $\underline{\widetilde{\boldsymbol{r}}_i}$. However, the actual payload $\widetilde{\mathsf{pd}}_i$ of the output record is determined later by the MPC. Given this, we can include in R_i all computations that do not require knowledge of $\widetilde{\mathsf{pd}}i$ and $\widetilde{\mathsf{cm}}i$. This includes the first two predicates of the relation: Merkle tree inclusion and input-coin well-formedness. Conversely, the third and fourth predicates—output-coin well-formedness and function evaluation—should be deferred to the MPC relation. To further minimize the MPC's work, we use the following idea.

$$\begin{aligned}
\widetilde{\mathsf{cm}}_i &= \mathsf{Com}_1(\widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i, \widetilde{\rho}_i; \underline{\widetilde{\boldsymbol{r}}_i}) \\
&= \mathsf{Com}_1(\widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i; \underline{\widetilde{\boldsymbol{r}}_i}) + \mathsf{Com}_1(0, \delta_i, 0; 0) \\
&= \widetilde{\mathsf{cm}}_i^* + \mathsf{Com}_1(0, \delta_i, 0; 0)
\end{aligned}$$

where $\delta_i = \widetilde{\mathsf{pd}}_i - \widetilde{\mathsf{pd}}_i^*$. For simplicity we use $+$ to denote the homomorphic operation over commitments. However, depending on the commitment scheme, this can be replaced with the appropriate operation.

We note that proving the field operation $\delta_i = \widetilde{\mathsf{pd}}_i - \widetilde{\mathsf{pd}}_i^*$ within the collaborative zkSNARK is significantly less expensive than proving the cryptographic operation $\widetilde{\mathsf{cm}}_i = \mathsf{Com}_1(\widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i, \widetilde{\rho}_i; \underline{\widetilde{\boldsymbol{r}}_i})$. Consequently, we can design the collaborative zkSNARK to operate over $\widetilde{\mathsf{cm}}_i^*, \delta_i$ instead of directly over $\widetilde{\mathsf{cm}}_i$.

This approach allows the local prover to handle the cryptographic operation $\widetilde{\mathsf{cm}}_i^* = \mathsf{Com}_1(\widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i; \underline{\widetilde{\boldsymbol{r}}_i})$ within the local zkSNARK, leaving the collaborative zkSNARK responsible solely for computing the field operation δ_i. Importantly, δ_i reveals no information about $\widetilde{\mathsf{pd}}_i$, since $\widetilde{\mathsf{pd}}_i^*$ is uniformly random and, thus, can be safely published.

This modification leads us to define an equivalent alternative relation R', which is functionally identical to R:

- $R'\Big(\left\{\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \delta_i\right\}_{i=1}^n, f\ ;$
  $\left\{\mathsf{path}_i, \mathsf{cm}_i, \mathsf{apk}_i, \mathsf{ask}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i}, \widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\mathsf{pd}}_i, \widetilde{\rho}_i, \underline{\widetilde{\boldsymbol{r}}_i}, \mathsf{aux}_i\right\}_{i=1}^n\Big) = 1 \iff$

  $$\begin{aligned}
  &\left\{\mathsf{MerkleVerify}(\mathsf{root}_i, \mathsf{path}_i, \mathsf{cm}_i) = 1\right\}_{i=1}^n \\
  \wedge\ &\left\{\mathsf{cm}_i = \mathsf{Com}_1(\mathsf{apk}_i, \mathsf{pd}_i, \rho_i; \underline{\boldsymbol{r}_i}) \ \wedge\ \mathsf{apk}_i = \mathsf{PRF}_{\mathsf{ask}_i}(0) \ \wedge\ \mathsf{sn}_i = \mathsf{PRF}_{\mathsf{ask}_i}(\rho_i)\right\}_{i=1}^n \\
  \wedge\ &\left\{\widetilde{\mathsf{cm}}_i^* = \mathsf{Com}_1(\widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i; \underline{\widetilde{\boldsymbol{r}}_i})\right\}_{i=1}^n \\
  \wedge\ &\left\{\delta_i = \widetilde{\mathsf{pd}}_i - \widetilde{\mathsf{pd}}_i^*\right\}_{i=1}^n
  \end{aligned}$$

$$\wedge\ f\Big((\mathsf{pd}_i)_{i=1}^n, (\mathsf{aux}_i)_{i=1}^n, (\widetilde{\mathsf{pd}}_i)_{i=1}^n\Big) = 1$$

The first three predicates can be computed locally by the clients and the MPC is only left with proving that the function f outputs 1 and that the δ_is were computed honestly.

Finally, since we will ultimately compose the zkSNARKs for the R_i's and R_{MPC} we define the commit-and-prove version of these relations, where all the shared witnesses between the relations are committed(see Section 3.1.1). Below is the definition of the local R_i and the collaborative R_{MPC}. We highlight in blue, the information that is added specifically for the linking the relations.

- $R_i\Big(\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathsf{cp},i}\ ;\ \mathsf{path}_i, \mathsf{cm}_i, \mathsf{apk}_i, \mathsf{ask}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i}, \widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i, \underline{\widetilde{\boldsymbol{r}}_i}, \mathsf{aux}_i, \underline{\mathsf{cp}_i}\Big) = 1 \iff$

  $\mathsf{MerkleVerify}(\mathsf{root}_i, \mathsf{path}_i, \mathsf{cm}_i) = 1$

  $\wedge\ \mathsf{cm}_i = \mathsf{Com}_1(\mathsf{apk}_i, \mathsf{pd}_i, \rho_i; \underline{\boldsymbol{r}_i})\ \wedge\ \mathsf{apk}_i = \mathsf{PRF}_{\mathsf{ask}_i}(0)\ \wedge\ \mathsf{sn}_i = \mathsf{PRF}_{\mathsf{ask}_i}(\rho_i)$

  $\wedge\ \widetilde{\mathsf{cm}}_i^* = \mathsf{Com}_1(\widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i; \underline{\widetilde{\boldsymbol{r}}_i})$

  $\wedge\ \mathsf{cm}_{\mathsf{cp},i} = \mathsf{Com}_2(\mathsf{aux}_i, \mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*; \underline{\mathsf{cp}_i})$

- $R_{\mathsf{MPC}}\Big(\{\delta_i\}_{i=1}^n, f, \{\mathsf{cm}_{\mathsf{cp},i}\}_{i=1}^n\ ;\ \{\mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\mathsf{pd}}_i, \mathsf{aux}_i\}_{i=1}^n, \{\underline{\mathsf{cp}_i}\}_{i=1}^n\Big) = 1 \iff$

  $\wedge\ \Big\{\delta_i = \widetilde{\mathsf{pd}}_i - \widetilde{\mathsf{pd}}_i^*\Big\}_{i=1}^n$

  $\wedge\ f\Big((\mathsf{pd}_i)_{i=1}^n, (\mathsf{aux}_i)_{i=1}^n, (\widetilde{\mathsf{pd}}_i)_{i=1}^n\Big) = 1$

  $\wedge\ \Big\{\mathsf{cm}_{\mathsf{cp},i} = \mathsf{Com}_2(\mathsf{aux}_i, \mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*; \underline{\mathsf{cp}_i})\Big\}_{i=1}^n$

We additionally include aux_i in $\mathsf{cm}_{\mathsf{cp},i}$ in order to "bind" aux_i with π_i; this prevents use of other aux_i as input to f.

5.4 Protocol Description

In this section, we provide a formal description of the $\pi_{\mathsf{Execute}}^{f,\mathbf{L}}$ and $\mathsf{Verify}^{f,\mathbf{L}}$ algorithms of our protocol, *Jigsaw*. We begin by detailing the main MPC protocol. Subsequently, we address a potential malicious behavior by clients that could lead to undesirable aborts and update the protocol with a cryptographic countermeasure to mitigate such risks. A concise summary of our protocol is shown in Figure 2.

5.4.1 Main Protocol

Our main protocol works as follows (the main building blocks used in this protocol are outlined in Section 5.2):

Local Computation – Clients. Each client C_i takes as input a record to be consumed, the last root of the ledger, a valid Merkle path, an auxiliary input and the predicate f that they wish to be applied to the record's payload: $\boldsymbol{r} = (\mathsf{cm}_i, \mathsf{apk}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i})$, ask_i, $\mathsf{sn}_i = \mathsf{PRF}_{\mathsf{ask}_i}(\rho)$, root_i, path_i, aux_i, f. She proceeds as follows:

1. **Tentative record generation:** Samples a new uniformly random record $\widetilde{\mathsf{pd}}_i^*$, $\underline{\widetilde{\boldsymbol{r}}_i}$, $\widetilde{\mathsf{ask}}_i$, $\widetilde{\rho}_i$ and computes its commitment $\mathsf{cm}_i^* = \mathsf{Com}_1(\widetilde{\mathsf{apk}}, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i; \underline{\widetilde{\boldsymbol{r}}_i})$. This forms the new tentative record $\widetilde{\boldsymbol{r}}_i^* = (\widetilde{\mathsf{cm}}_i^*, \widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i, \underline{\widetilde{\boldsymbol{r}}_i})$.

2. **zkSNARK proof computation:** Samples a random $\underline{\mathsf{cp}_i}$ and computes the linking commitment $\mathsf{cm}_{\mathrm{cp},i} = \mathsf{Com}(\mathsf{aux}_i, \mathsf{pd}_i, \widetilde{\rho}_i; \underline{\mathsf{cp}_i})$. Then proceeds to the zkSNARK proof computation for R_i, $\pi_i \leftarrow \mathsf{zkSNARK}.\mathcal{P}\big((\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathrm{cp},i}), (\mathsf{path}_i, \mathsf{cm}_i, \mathsf{apk}_i, \mathsf{ask}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i}, \widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i, \underline{\widetilde{\boldsymbol{r}}_i}, \mathsf{aux}_i, \underline{\mathsf{cp}_i})\big)$.
3. **Secret sharing of secret values:** Computes secret shares $[\mathsf{pd}_i]$, $[\widetilde{\mathsf{pd}}_i^*]$, $[\underline{\mathsf{cp}_i}]$ of the witness-inputs $\mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*$ and $\underline{\mathsf{cp}_i}$ to R_{MPC}.
4. **Signing of the function:** Computes a signature of knowledge on f and π_i. The secret key used for this signature is $\mathbb{w}_i = (\mathsf{path}_i, \mathsf{cm}_i, \mathsf{apk}_i, \mathsf{ask}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i}, \widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i, \underline{\widetilde{\boldsymbol{r}}_i}, \mathsf{aux}_i, \underline{\mathsf{cp}_i})$ and public key is $\mathbb{x}_i = (\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathrm{cp},i})$. The client computes $\sigma_i \leftarrow \mathsf{SoK}.\mathsf{Sign}_{\mathbb{w}}(f, \pi_i)$.

Finally, the client sends to each MPC Server j, the public values/statement of R_i together with the proof π_i, their corresponding share of the secret values/witnesses of R_{MPC} and the signature σ_i, i.e.,

$$f, \mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathrm{cp},i}, \pi_i, [\mathsf{pd}_i]_j, [\widetilde{\mathsf{pd}}_i^*]_j, [\mathsf{aux}_i]_j, [\underline{\mathsf{cp}_i}]_j, \sigma_i$$

MPC Service – Servers' Computation. Each server S_j receives the following from each client C_i: $\{f, \mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathrm{cp},i}, \pi_i, [\mathsf{pd}_i]_j, [\widetilde{\mathsf{pd}}_i^*]_j, [\mathsf{aux}_i]_j, [\underline{\mathsf{cp}_i}]_j, \sigma_i\}$ and then verifies the following for each client C_i:

1. **Verify signatures of knowledge:** Checks if $\mathsf{SoK}.\mathcal{V}_{\mathbb{x}}(\sigma_i, (f, \pi_i))$ outputs 1, where $\mathbb{x}_i = (\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathrm{cp},i})$.
2. **Verify validity of Merkle roots:** Checks if $\mathbf{L}.\mathsf{ValidateDigest}(\mathsf{root}_i)$ outputs 1.
3. **Verify zkSNARK:** Checks if $\mathsf{zkSNARK}.\mathcal{V}(\pi_i, \mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathrm{cp},i})$ outputs 1.

If all of the above checks pass, then the servers proceed to collectively compute the following:

1. Run an MPC protocol on inputs $\{[\mathsf{pd}_i], [\widetilde{\mathsf{pd}}_i^*], [\mathsf{aux}_i]\}_{i=1}^n$ to compute $\{\widetilde{\mathsf{pd}}_i\}_{i=1}^n$ such that $f\Big((\mathsf{pd}_i)_{i=1}^n, (\mathsf{aux}_i)_{i=1}^n, (\widetilde{\mathsf{pd}}_i)_{i=1}^n\Big) = 1$ and then compute $\{\delta_i = \widetilde{\mathsf{pd}}_i - \widetilde{\mathsf{pd}}_i^*\}_{i=1}^n$. At the end of this MPC protocol, each server S_j learns $\{\delta_i\}_{i=1}^n$ and shares $\{[\widetilde{\mathsf{pd}}_i]_j\}_{i=1}^n$.
2. Compute a collaborative zkSNARK for the relation R_{MPC}, $\pi_{\mathsf{MPC}} \leftarrow \mathsf{colzkSNARK}.\mathcal{P}\Big(\Big(\{\delta_i\}_{i=1}^n, f, \{\mathsf{cm}_{\mathrm{cp},i}\}_{i=1}^n\Big)\ ;\ \Big(\{\mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\mathsf{pd}}_i, \mathsf{aux}_i\}_{i=1}^n, \{\underline{\mathsf{cp}_i}\}_{i=1}^n\Big)\Big)$.

Finally the servers output the following transaction:

$$\mathsf{tx} = \{\mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*\}_{i=1}^n, \mathsf{memo} = \bot, \star = \big(\{\mathsf{root}_i, \mathsf{cm}_{\mathrm{cp},i}, \pi_i, \delta_i, \sigma_i\}_{i=1}^n, \pi_{\mathsf{MPC}}, f\big)$$

Smart Contract – Verification. Given a transaction tx, the verifier parses $\mathsf{tx} = \{\mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*\}_{i=1}^n, \mathsf{memo} = \bot, \star = \big(\{\mathsf{root}_i, \mathsf{cm}_{\mathrm{cp},i}, \pi_i, \delta_i, \sigma_i\}_{i=1}^n, \pi_{\mathsf{MPC}}, f\big)$ and then accepts (outputs 1) if all of the following conditions are met:

1. For each $i = [n]$: $\mathsf{zkSNARK}.\mathcal{V}(\pi_i, \mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathrm{cp},i}) = 1$.
2. For each $i = [n]$: $\mathsf{SoK}.\mathcal{V}_{\mathbb{x}_i}(\sigma, f, \pi_i) = 1$, where $\mathbb{x}_i = (\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathrm{cp},i})$.
3. $\mathsf{colzkSNARK}.\mathcal{V}(\pi_{\mathsf{MPC}}, \{\delta_i\}_{i=1}^n, f, \{\mathsf{cm}_{\mathrm{cp},i}\}_{i=1}^n) = 1$.
4. $\mathbf{L}.\mathsf{ValidateDigest}(\mathsf{root}_i) = 1$

5.4.2 Ensuring Correctness of Secret Sharing

To complete our construction, we address a subtle caveat. Each client i computes a local proof π_i for (among other values) $\mathsf{aux}_i, \mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*, \underline{\mathsf{cp}_i}$, then secret shares these values and sends the shares $[\mathsf{pd}_i], [\widetilde{\mathsf{pd}}_i^*], [\mathsf{aux}_i], [\underline{\mathsf{cp}_i}]$ to the servers. However, a malicious client could compute the proof using one set of values (e.g., aux_i) but secret share different ones (e.g., aux_i').

While this doesn't compromise the security of the main protocol—since the final proof π_{MPC} will be invalid (as aux_i' doesn't correctly open $\mathsf{cm}_{\mathrm{cp},i} = \mathsf{Com}_2(\mathsf{aux}_i, \mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*; \underline{\mathsf{cp}_i})$)—it allows a client to sabotage the procedure f without being detected. For instance, in an auction, a malicious client could disrupt the outcome undetected.

To prevent such sabotage, we augment *Jigsaw* with a proof of correct secret sharing: clients additionally prove to the servers that their shares match the values used in π_i. Specifically, each client i commits to the secret shares of $\mathsf{aux}_i, \mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*, \underline{\mathsf{cp}_i}$ using a third commitment scheme Com_3:

$$\mathsf{cm}_{\mathsf{SS},i} = \mathsf{Com}_3\left(\left([\mathsf{aux}_i]_k, [\mathsf{pd}_i]_k, [\widetilde{\mathsf{pd}}_i^*]_k, [\underline{\mathsf{cp}_i}]_k\right)_{k=1}^m ; \underline{\mathsf{SS}_i}\right)$$

Then, the client computes a NIZK proof for each server, showing that their share is consistent with both $\mathsf{cm}_{\mathrm{cp},i}$ and $\mathsf{cm}_{\mathsf{SS},i}$. Formally, the NIZK relation is:

- $R_{i\to j}\Big(\mathsf{cm}_{\mathrm{cp},i},\ \mathsf{cm}_{\mathsf{SS},i},\ [\mathsf{aux}_i]_j,\ [\mathsf{pd}_i]_j,\ [\widetilde{\mathsf{pd}}_i^*]_j,\ [\underline{\mathsf{cp}_i}]_j\ ;\ \mathsf{aux}_i,\ \mathsf{pd}_i,\ \widetilde{\mathsf{pd}}_i^*,\ \underline{\mathsf{cp}_i},\ \big\{[\mathsf{aux}_i]_k,\ [\mathsf{pd}_i]_k,\ [\widetilde{\mathsf{pd}}_i^*]_k,\ [\underline{\mathsf{cp}_i}]_k\big\}_{k\neq j}, \underline{\mathsf{SS}_i}\Big) = 1 \iff$

  $\mathsf{cm}_{\mathrm{cp},i} = \mathsf{Com}_2(\mathsf{aux}_i, \mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*; \underline{\mathsf{cp}_i})$

  $\wedge\ \mathsf{cm}_{\mathsf{SS},i} = \mathsf{Com}_3(([\mathsf{aux}_i]_k, [\mathsf{pd}_i]_k, [\widetilde{\mathsf{pd}}_i^*]_k, [\underline{\mathsf{cp}_i}]_k)_{k=1}^m; \underline{\mathsf{SS}_i})$

  $\wedge\ \left([\mathsf{aux}_i]_k\right)_{k=1}^m = \mathsf{SS}(\mathsf{aux}_i) \wedge \left([\mathsf{pd}_i]_k\right)_{k=1}^m = \mathsf{SS}(\mathsf{pd}_i) \wedge \left([\widetilde{\mathsf{pd}}_i^*]_k\right)_{k=1}^m = \mathsf{SS}(\widetilde{\mathsf{pd}}_i^*) \wedge \left([\underline{\mathsf{cp}_i}]_k\right)_{k=1}^m = \mathsf{SS}(\underline{\mathsf{cp}_i})$

5.4.3 On Accessibility of Assets

Previous attacks, such as those in [HBHW22], show that allowing an adversary to create another user's records and freely choose ρ can lead to non-consumable records. In our protocol, however, each client generates records exclusively for themselves, making this attack inapplicable. We allow ρ to be chosen uniformly at random by the client in the honest case, as they have no incentive to "burn" their own records.

5.4.4 Handling Multiple Smart Contract Interactions

For simplicity, we described the protocol for a single application / smart-contract, implemented by the function denoted so far as f. In practice, a smart contract platform will want to support multiple such applications, and allow users to use their assets across any application of their choosing – for instance, a user may wish to trade their (wrapped) BTC for (wrapped) ETH using an atomic swap application, and then use that ETH to place a sealed bid in an auction, etc.

Handling multiple functions is a delicate task when function privacy is required[11]. We instead choose to leak identity of which functions are being executed. That is, any observer can determine that an atomic

[11] ZEXE [BCG+20] gives a shared execution environment for function privacy.

Client$_i\big(\mathsf{cm}_i, \mathsf{apk}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i}, \mathsf{ask}_i, \mathsf{sn}_i, \mathsf{root}_i, \mathsf{path}_i, \mathsf{aux}_i, f\big)$:

1. Sample $\widetilde{\mathsf{pd}}_i^*, \widetilde{\underline{\boldsymbol{r}}}_i, \widetilde{\mathsf{ask}}_i, \widetilde{\rho}_i$; set $\mathsf{cm}_i^* = \mathsf{Com}_1(\widetilde{\mathsf{apk}}, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i; \widetilde{\underline{\boldsymbol{r}}}_i)$.
2. Sample $\underline{\mathsf{cp}_i}$; set $\mathsf{cm}_{\mathsf{cp},i} = \mathsf{Com}_2(\mathsf{aux}_i, \mathsf{pd}_i, \widetilde{\rho}_i; \underline{\mathsf{cp}_i})$ and $\pi_i = \mathsf{zkSNARK}.\mathcal{P}\big(\mathsf{srs}, \mathbb{x}_i, \mathbb{w}_i\big)\big)$, where

$$\mathbb{x}_i = (\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathsf{cp},i}),\ \mathbb{w}_i = (\mathsf{path}_i, \mathsf{cm}_i, \mathsf{apk}_i, \mathsf{ask}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i}, \widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\rho}_i, \widetilde{\underline{\boldsymbol{r}}}_i, \mathsf{aux}_i, \underline{\mathsf{cp}_i})$$

3. Compute $[\mathsf{pd}_i]$, $[\widetilde{\mathsf{pd}}_i^*]$, $[\underline{\mathsf{cp}_i}]$.
4. Set $\sigma_i \leftarrow \mathsf{SoK.Sign}_{\mathbb{w}_i}(f, \pi_i)$.
5. Sample $\underline{\mathsf{SS}_i}$; set $\mathsf{cm}_{\mathsf{SS},i} = \mathsf{Com}_3(([\mathsf{aux}_i]_k, [\mathsf{pd}_i]_k, [\widetilde{\mathsf{pd}}_i^*]_k, [\underline{\mathsf{cp}_i}]_k)_{k=1}^m, \underline{\mathsf{SS}_i})$ and $\pi_{i\to j} = \mathsf{NIZK}.\mathcal{P}(\mathbb{x}_{i\to j}, \mathbb{w}_{i\to j})$, where

$$\mathbb{x}_{i\to j} = (\mathsf{cm}_{\mathsf{cp},i}, \mathsf{cm}_{\mathsf{SS}}, [\mathsf{aux}_i]_j, [\mathsf{pd}_i]_j, [\widetilde{\mathsf{pd}}_i^*]_j, [\underline{\mathsf{cp}_i}]_j),$$

$$\mathbb{w}_{i\to j} = (\mathsf{aux}_i, \mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*, \underline{\mathsf{cp}_i}, \big\{[\mathsf{aux}_i]_k, [\mathsf{pd}_i]_k, [\widetilde{\mathsf{pd}}_i^*]_k, [\underline{\mathsf{cp}_i}]_k\big\}_{k\neq j}, \underline{\mathsf{SS}_i})$$

Output to each $j = 1, \ldots, m$: $f, \mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathsf{cp},i}, \pi_i, [\mathsf{pd}_i]_j, [\widetilde{\mathsf{pd}}_i^*]_j, [\mathsf{aux}_i]_j, [\underline{\mathsf{cp}_i}]_j, \sigma_i, \mathsf{cm}_{\mathsf{SS},i}, \pi_{i\to j}$

MPC Service : Each S_j: For every $i = 1, \ldots, n$, if $\pi_i, \sigma_i, \pi_{i\to j}$ do not verify exclude user i (and set $n \leftarrow n-1$ and client labels $i+1 \leftarrow i$, $i+2 \leftarrow i+1$, etc.).

$S_j\big(\{[\mathsf{pd}_i]_j, [\widetilde{\mathsf{pd}}_i^*]_j, [\mathsf{aux}_i]_j, [\underline{\mathsf{cp}_i}]_j\}_{i=1}^n\big)$, for $j = 1, \ldots, m$ involve in a protocol that computes:

1. $\{\delta_i = \widetilde{\mathsf{pd}}_i - \widetilde{\mathsf{pd}}_i^*\}_{i=1}^n$ such that $f\big((\mathsf{pd}_i)_{i=1}^n, (\mathsf{aux}_i)_{i=1}^n, (\widetilde{\mathsf{pd}}_i)_{i=1}^n\big) = 1$.
2. $\pi_{\mathsf{MPC}} = \mathsf{colzkSNARK}.\mathcal{P}(\big\{\delta_i\big\}_{i=1}^n, f, \{\mathsf{cm}_{\mathsf{cp},i}\}_{i=1}^n\ ;\ \big\{\mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\mathsf{pd}}_i, \mathsf{aux}_i\big\}_{i=1}^n, \big\{\underline{\mathsf{cp}_i}\big\}_{i=1}^n)$.

Output: $\{\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathsf{cp},i}, \pi_i, \delta_i, \sigma_i\}_{i=1}^n, \pi_{\mathsf{MPC}}, f$

Verification$(\{\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathbf{cp},i}, \pi_i, \delta_i, \sigma_i\}_{i=1}^n, \pi_{\mathsf{MPC}}, f)$: Output 1 iff:

1. $\big\{\mathsf{zkSNARK}.\mathcal{V}(\pi_i, \mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathsf{cp},i}) = 1\big\}_{i=1}^n$.
2. $\big\{\mathsf{SoK}.\mathcal{V}_{\mathbb{x}_i}(\sigma, f, \pi_i) = 1\big\}_{i=1}^n$.
3. $\mathsf{colzkSNARK}.\mathcal{V}(\pi_{\mathsf{MPC}}, \{\delta_i\}_{i=1}^n, f, \{\mathsf{cm}_{\mathsf{cp},i}\}_{i=1}^n) = 1$.

Figure 2: Our Protocol, *Jigsaw*.

swap or an auction occurred, but not necessarily the identities of the participants (assuming anonymous communication mechanisms) or the assets being used – so in our example above, an adversary cannot infer from the on-chain transaction data that the output UTXO of the atomic swap is used as input to the

sealed-bid auction[12]. Not surprisingly, this leakage provides an added performance benefit that we do not need to introduce "dummy" UTXOs to pad all function executions to an upper bound number of input and output UTXOs.

One way to handle multiple applications is to use a shared settlement layer, which, by checking the uniqueness of spent nullifiers, would ensure safe usage of assets across multiple applications. This has the drawback that a careless or malicious user can submit the same UTXO to multiple applications (MPC instances) concurrently, and even though only one will successfully settle, it will lead to wasted computation or likely denial-of-service to other users.

Instead, we propose a form of "UTXO-locking" where the client issues a transaction converting a UTXO to another equivalent-value UTXO locked to a specific application. To that end, we reserve a field within the UTXO payload pd for the application identifier, which when set to 0 indicates an unlocked UTXO that is simply owned by the user. In order to lock that UTXO to an application denoted by a unique id app-id, the user submits a transaction directly on-chain that consumes an unlocked UTXO and produces a locked UTXO with the aforementioned field within pd set to app-id – all other aspects of the UTXO, such as the owner, the asset id, value, etc. must be preserved (except for any fees). Then, the on-chain settlement for any application will only accept UTXOs that are locked to that application.

Note that UC Security of Jigsaw is required for concurrent MPC instances, and we discuss UC Security in A.

5.4.5 Security

The aforementioned construction, *Jigsaw*, is a secure doubly private smart contract (DPSC) scheme, according to the definition in Section 4. We defer the security proof to Appendix A.

5.5 Instantiation and optimizations

While there are alternatives, our choice for concrete cryptographic instantiations are governed by some principles: 1) avoiding use of non-native arithmetic within the zkSNARK circuit; 2) avoiding (relatively) expensive operations such as hashing or commitments within the collaborative relation; 3) allowing for efficient composition of all components, namely the local zkSNARK proof, the collaborative zkSNARK proof, and any commit-and-prove linking proofs.

5.5.1 SNARK Gadgets:

Pedersen hashing is employed for both the Merkle tree hash function and the commitment function Com_1. To that end, we use a twisted Edwards curve, whose base field is the scalar field of the zkSNARK's pairing friendly elliptic curve (e.g. BabyJubJub on BN-254 [WBB20]). For the PRF, we use the Poseidon[GKR+21] hash, modeled as a random oracle.

5.5.2 (Collaborative) zkSNARKs: zkSNARK, colzkSNARK and Com_2.

For zkSNARK and colzkSNARK we rely on PLONK [GWC19]. Ozdemir and Boneh presented in [OB22a] an efficient Collaborative zkSNARK based on Plonk. We use this for colzkSNARK.

The caveat here is that we need a commit-and-prove variant of PLONK: We need Com_2 to be opened in both the local zkSNARK and the MPC (Collaborative) zkSNARK. A straightforward solution would be

[12] This guarantee is modulo whatever metadata or auxiliary information is already known to the adversary before the transaction is executed.

to resort again to Pedersen Commitment for Com_2 that, in combination with the JubJub curve, can be efficiently opened inside the zkSNARK. This would be, indeed, efficient when PLONK is locally computed; however, this is not the case when it goes to the collaborative zkSNARK. This is because the Pedersen circuit has a (relatively) large multiplication depth, which translates to many rounds of interaction even to compute the extended witness in the collaborative case. Therefore, this would be a source of inefficiency for the collaborative zkSNARK.

We design an algebraic protocol for the opening of a commitment inside PLONK. Let $\mathsf{cm}_{\mathsf{cp}} = \mathsf{Com}_2(x_1, \ldots, x_n; r)$ be a commitment and a PLONK proof that (among other things) proves something with $x_1, \ldots, x_n$ as witness. Our main observation is that the PLONK proof already includes a KZG commitment [KZG10] to the extended witness, which includes the actual witnesses $x_1, \ldots, x_n$:

$$A = \mathsf{KZG.Com}(\ldots + x_1 \cdot L_{i_1}(X) + \ldots + x_n \cdot L_{i_n}(X) + \ldots),$$

for some indices $i_1, \ldots, i_n$, where L_k denotes the Lagrange polynomial. Therefore we set Com_2 to be a KZG commitment:

$$\mathsf{Com}_2(x_1, \ldots, x_n; r) = \mathsf{KZG.Com}(x_1 \cdot L_1(X) + \ldots + x_n \cdot L_n; r).$$

After that linking the PLONK proof with the commitment boils down to proving the element mapping i_1 to $1, \ldots, i_n$ to n between the two polynomials in the commitments, which resembles a permutation argument.

We present our full technique for Commit-and-Prove PLONK, using the above linking idea in Appendix B.

5.5.3 Correctness of Secret Sharing: Com_3 and NIZK

For correctness of the secret sharing we, first, set the secret sharing scheme to be m-out-of-m Shamir's secret sharing. Secret Sharing x, essentially, gives an $m-1$-degree polynomial $P_{\mathsf{SS}}(X)$ such that $P_{\mathsf{SS}}(\omega^0) = x$, $P_{\mathsf{SS}}(\omega^1) = [x]_1, \ldots, P_{\mathsf{SS}}(\omega^m) = [x]_m$. Then we set the secret sharing commitment Com_3 to be a KZG commitment of the above polynomial.

Now, for the correctness of secret sharing in our scheme we want to link

$$\mathsf{cm}_{\mathsf{cp}} = \mathsf{KZG.Com}(x_1 \cdot L_1(X) + \ldots + x_n \cdot L_n(X); r)$$

with the above secret sharing type of commitments. Therefore we let $\mathsf{cm}_{\mathsf{SS}} = \mathsf{Com}_3(x_1, \ldots, x_n; \boldsymbol{r})$ consist of n KZG commitments to the secret sharing polynomials of $x_1, \ldots, x_n$ respectively:

$$\begin{array}{c} \mathsf{KZG.Com}(P_{\mathsf{SS},1}(X); r_1) \\ \vdots \\ \mathsf{KZG.Com}(P_{\mathsf{SS},n}(X); r_n) \end{array}$$

After that the NIZK proof is, similarly to above, n mapping proofs between indices in the $\mathsf{cm}_{\mathsf{cp}}$ polynomial and the 0-th root of the n $\mathsf{cm}_{\mathsf{SS}}$ polynomials respectively.

5.5.4 Multi-party Computation for Discovery: MPC

For the plain MPC protocol that computes the δ_i's (step (1) of the protocol of the MPC Service in Figure 2) we suggest the use of the SPDZ protocol [DPSZ12], with optimizations from [DEF+19b].

5.5.5 Signature of Knowledge: SoK

As shown in [GM17], any simulation extractable zkSNARK can be a signature of knowledge. Furthermore, in [GKK+22] it was shown that PLONK is already simulation extractable and can also be used as a Signature of Knowledge, and even further, recently it was shown that real-world implementations of PLONK are Simulation Extractable [FFR24]. In light of these results, we get the SoK σ, using the local PLONK proof, with zero overhead.

For more implementation details, we refer to Section 7.

6 Applications

We implement several financial applications using our framework, such as trading, auctions and voting. In each of these applications, we observe the familiar "fire-and-forget" feature: users submit their orders to the MPC service and go offline, and can return online at any later point to view the results of the computation. This is a significant win in user experience. Not surprisingly, we demonstrate that the mechanism of discovering counterparties, such as in trading, can also be executed privately (i.e., the orders or user identities are never revealed in the clear to any party) and autonomously (i.e., without user intervention). This is yet another win in user experience.

While our system can support arbitrary record types, our financial applications must transact with assets such as coins. Recall that records are denoted by their commitment $\mathsf{cm} = \mathsf{Com}(\mathsf{apk}, \mathsf{pd}, \rho; \underline{r})$, where the payload pd contains application-specific data. For coins, we set this payload to have two elements, amount and assetID, where the amount is a 64-bit number (encoded as a field element) and the assetID is a unique identifier such as the ERC-20 token address (also encoded as a field element). A coin can be used in any smart contract, by submitting the order to the MPC service, via the use of the aux field as shown in Section 5.3. For instance, an atomic swap is performed by inputting a coin that the user owns and specifying the desired coin value and asset id as aux fields.

For exposition ease, we will elide details on transaction fees from each of our applications. They can be accounted for by considering an additional output UTXO that pays the MPC service – for efficiency, the MPC service can be paid as a single payee, making the number of UTXOs independent of the number of MPC servers.

In addition to these applications, we borrow the **split-join** transactions from zCash [HBHW22] to allow users to split a coin into multiple coins (of potentially lesser value) or combine multiple coins into one (of aggregate value); these transactions can also be used to send funds to another user, thus enabling anonymous payments a la zCash. Split-join transactions can be invoked at any time by a user, and their proof can be computed locally, bypassing the MPC network altogether. Similar to the local proofs in each application, a split-join transaction must prove that the input coins are valid – i.e., their commitments exist in the merkle tree, the prover knows the corresponding secret keys, etc. – and the output coins are well-formed and aggregate value is conserved.

6.1 Atomic Swap

Let us consider an atomic swap functionality that allows any two users to swap their assets (with left-over change) via a single transaction – this can be naturally extended to multiple parties, but we focus on two-party swaps as they are widely used in DeFi today. While prior works such as Zexe also enable atomic swaps, they require the users to discover the counterparties, possibly via a trusted third party, and, even

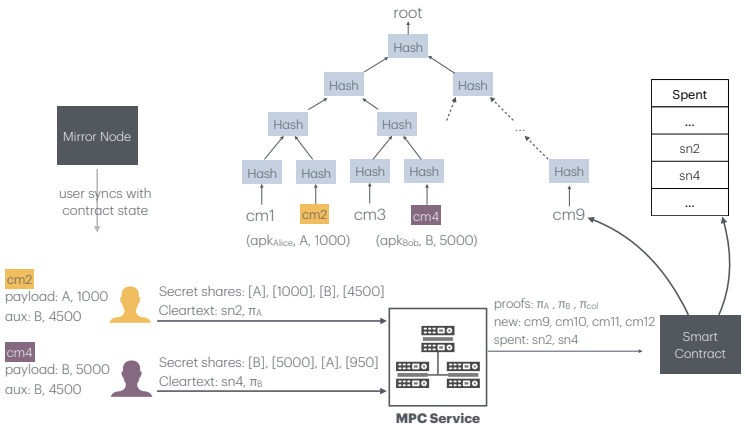

Figure 3: System Architecture for an Atomic Swap contract

after the matching, require the counterparties to interact with each other to generate a single zkSNARK proof. In contrast, Jigsaw lets users simply "fire-and-forget" their orders, and let the MPC service perform the matching and (the remaining steps of the) proof generation. This is a big win in user experience.

Functionality. Consider users Alice and Bob, where Alice has asset A and wishes to acquire B, and vice versa for Bob – assets are identified by some unique descriptor, such as their ERC-20 token address. Alice specifies her swap order via the tuple $(\mathsf{A}, \mathsf{B}, \mathsf{amount_A}, \mathsf{min_B_for_A})$, and Bob specifies his swap order via the tuple $(\mathsf{B}, \mathsf{A}, \mathsf{amount_B}, \mathsf{min_A_for_B})$; as an example, Alice submits (A, B, 1000, 4500) indicating her desire for 4500 B tokens in exchange for 1000 A tokens, while Bob submits (B, A, 5000, 950) indicating his desire for 950 A tokens in exchange for 5000 B tokens. While the two swap orders do not match perfectly, they are nevertheless compatible, and can be executed by issuing 4500 B tokens to Alice (with 500 tokens changed back to Bob) and similarly issuing 950 A tokens to Bob (with 50 tokens changed back to Alice). More generally, any pair of swap orders are compatible if $\mathsf{amount_A} \geq \mathsf{min_A_for_B} \wedge \mathsf{amount_B} \geq \mathsf{min_B_for_A}$.

Local Computation. Each party submits, in addition to the locally-generated zkSNARK proof, one serial number denoting the input coin, two commitments denoting the output and change coins (for now, they commit to random payloads), and an auxiliary data aux encoding the desired order. For the swap functionality, we need aux to include two values: desired asset type assetID and min_desired_amount (either min_A_for_B or min_B_for_A); coins have the expected payloads containing the values assetID and amount.

MPC Matching. We implement a simple strategy that maintains an evolving database of pending, unmatched orders[13]. Whenever a user U submits an order, the MPC service evaluates the compatibility check $\mathsf{U.input.amount} \geq \mathsf{P.aux.min_desired_amount} \wedge \mathsf{P.input.amount} \geq \mathsf{U.aux.min_desired_amount}$ for each pending order P in its database, and either outputs a matched swap or stores the order in the order book. Note that the above compatibility check is evaluated as a MPC computation over secret-shared data. If the assetID must be kept private, then the MPC can additionally enforce equality over those payload bits.

Collaborative zkSNARK. The output of the previous matching computation is a pair of matched swap

[13] MPC-based dark pools have been recently developed [BK], and one could implement those strategies here. The matching algorithm could provide additional properties of interest, such as fair ordering or optimal matching, but we consider those out of scope as we focus this paper on the design of the collaborative zkSNARK.

orders, and the MPC service now computes the collaborative zkSNARK proof. Besides the construction-specific constraints, the following constraints must also be satisfied for a valid swap transaction: 1) asset ID and amount compatibility, 2) conservation of value, 3) correct output asset IDs.

- **asset compatibility**: Alice.aux.assetID = Bob.input.assetID $\wedge$ Bob.aux.assetID = Alice.input.assetID
- **amount compatibility**: Alice.input.amount $\geq$ Bob.aux.min_desired_amount $\wedge$ Bob.input.amount $\geq$ Alice.aux.min_desired_amount
- **conservation**: Alice.input.amount = Bob.output.amount + Alice.change.amount $\wedge$ Bob.input.amount = Alice.output.amount + Bob.change.amount
- **correct assets**: Alice.output.assetID = Alice.aux.assetID $\wedge$ Alice.change.assetID = Alice.input.assetID $\wedge$ Bob.output.assetID = Bob.aux.assetID $\wedge$ Bob.change.assetID = Bob.input.assetID

6.2 Sealed-bid Auctions

We implement a private auction that allows participants to participate anonymously and also keep their bids private. Moreover, upon completion, each participant only learns a single additional bit: whether their bid succeeded. We implement a first price auction for simplicity. While we do not limit the number of participants, we must set bounds in order to perform the downstream MPC, as the collaborative zkSNARK circuit requires a relation with a fixed size.

Local Computation. Consider an asset A being auctioned off (e.g. a non-fungible token). The asset holder submits to the MPC network the record denoting A, along with auxiliary data encoding the desired asset type (e.g. USDC), and optionally, the minimum acceptable amount. Each bidder $\mathsf{B_i}$ submits an order derived from their input coin cm_i and the auxiliary order information aux_i, which specifies the auctioned asset A. The bid also includes an output record $\widetilde{\mathsf{cm}_i}$, to be used in case that bidder is selected to be the winner. The output record has an identical payload as the record for asset A, but has the bidder's public key apk instead of the current asset owner.

MPC Auctioneering. The auction computation takes as input an array of secret-shared bids, and outputs (in the clear) the index j of the highest bid. To that end, since we experiment with only hundreds of bids, we use a sequence of secure comparators, although more efficient algorithms for maximum computation exist in literature.

Collaborative zkSNARK. An auction amongst n participants is settled via a single on-chain transaction representing n input coins and 1 output record commitment, which is the winning bidder j's submission. In addition to the construction-specific constraints, the collaborative zkSNARK proof checks for the satisfaction of the following constraint:

$$\forall_{i \in n} \ \text{bidder_i.aux.assetID} = \text{auctioneer.input.assetID}$$
$$\forall_{i \in n} \ \text{bidder_j.input.amount} \geq \text{bidder_i.input.amount}$$

6.3 Lotteries

We implement a lottery contract that allows a participating user i to enter with x_i USDCs. The winner is chosen using some coin tossing protocol amongst the MPC servers, though we do not prove its outcome for simplicity. Each user has a winning probability that is weighted with x_i. A lottery instance with

n participants is settled via a single on-chain transaction representing n input coins and 1 output coin containing the prize amount of $\sum_i^n x_i$ USDCs. As with auctions, we

Local Computation. In addition to proving ownership of a valid input coin, the user uses an aux field to specify the lottery instance.

MPC Lottery. The MPC computes the payout $\sum_i^n x_i$ for the winning user, and the appropriate adjustment δ_i for user i's output coin.

Collaborative zkSNARK. The relation asserts that the output coin value equals $\sum_i^n x_i$, and that all inputs' aux values are equal.

6.4 Voting

We consider a form of voting that is popular in decentralized autonomous organizations today. Each DAO member owns a stake, which corresponds to some weight in the voting process. There are various mechanisms for translating stake to voting weight. We implement a popular quadratic voting scheme, wherein a user with x^2 stake gets x votes.

Consider a proposal P with two outcomes: YES or NO[14]. The voting phase commences when the DAO produces a (temporary) output record with two payload fields, one for tallying up the YES votes and one for the NO votes; the output record is generated under a public key chosen by the DAO, and the corresponding secret key could be published to the DAO members for disemminating the results of the election. When voting on a proposal, each user submits a vote order to the MPC service where the input record specifies the user's stake (e.g. within a NFT token issued by the DAO). The MPC service adds up all the votes, and produces the modifications for the YES and NO fields of the output record. The voting process with n voters terminates via a single transaction containing n input records and 1 output records.

Local Computation. As stated above, each user i submits to the MPC service an input record denoting their stake x_i^2, an auxiliary data (x_i, v_i) denoting their voting weight x_i and binary vote v_i, and, as expected, a locally generated zkSNARK proof required by our protocol.

MPC Tallying. The MPC computation simply adds up the secret-shared weights to compute $\Sigma_i x_i \cdot v_i$ for the YES votes and $\Sigma_i x_i \cdot (1 - v_i)$ for the NO votes. These sums remain in secret-shared forms, and the MPC computes the adjustments to the two aforementioned output fields.

Collaborative zkSNARK. The collaborative zkSNARK proof is responsible for proving the correctness of the voting weights, and the correct computation of the summation. Specifically, we have the constraints:

$$\forall_{i \in n} \ \text{voter_i.input.stake} = \text{voter_i.aux.x} * \text{voter_i.aux.x}$$

$$\text{output.pd.yes} = \Sigma_i \ \text{voter_i.aux.x} * \text{voter_i.aux.v}$$

$$\text{output.pd.no} = \Sigma_i \ \text{voter_i.aux.x} * (1 - \text{voter_i.aux.v})$$

Similar to the auction, the relation must have fixed size, so we choose an upper bound n (e.g. 100) for the number of input votes – this can be the number of DAO members.

[14] Our mechanism naturally supports concurrent voting on several proposals, where the user must allocate their x votes across multiple proposals, by generating x records each of unit weight.

7 Implementation and Evaluation

System Architecture. Our system contains the following key components: 1) a local SNARK prover that runs on client devices; 2) a MPC service that performs the application-specific computation (e.g. matching or tallying); 3) a collaborative SNARK prover; and 4) smart contract for verifying proofs and settling transactions. The local SNARK prover is implemented using a combination of TurboPLONK [GW20], as implemented in the Noir toolchain [nc24], and arkworks libraries [ac22] for commit-and-prove linking. We use the OB [OB22b] construction for the collaborative SNARK protocol, as implemented in the Taceo toolchain [tc24]. We implement any application-specific MPC using MP-SPDZ [Kel20], specifically its implementation of [DEF+19a]. Finally, we use write the smart contract in Solidity for use on any EVM-compatible (Ethereum Virtual Machine) blockchain.

Crytographic Primitives. Our choice of concrete cryptographic instantiations follows key principles: 1) avoiding non-native arithmetic inside the zkSNARK circuit; 2) minimizing expensive operations like hashing or commitments within the collaborative relation; 3) enabling efficient composition of all components — local zkSNARK proofs, collaborative zkSNARK proofs, and commit-and-prove linking proofs. Our SNARKs use pairing-friendly elliptic curves, specifically the BN-254 [DSD07] curve. We also use the Baby JubJub twisted Edwards curve [WBB20], whose base field matches the scalar field of BN-254. Commitments and Merkle Trees rely on Pedersen hashing [HBHW22], employing a 4-bit window Pedersen hash that maps bit sequences to compressed elliptic curve points, allowing PRF outputs to be field elements in the circuit's scalar field. Pedersen hash generators come from a "nothing-up-my-sleeve" setup by hashing a public string to curve elements. Our PRFs use the Poseidon Hash [GKR+21].

Experimental Setup. We use a Macbook Pro with 8-core M2 CPU and 16 GB RAM for the local computation. The MPC protocols for matching and collaborative SNARK proving are run on a cluster of 3 AWS c4.xlarge machines, with 4 vCPUs and 8 GB RAM each. We do an end-to-end evaluation in both the Local Area Network (LAN) setting and Wide Area Network (WAN) setting. For the LAN configuration, we allocate the machines in the same AWS region, by instantiating a virtual private cloud (VPC) to place them in the same LAN subnet – the latency between any two machines in the LAN setting was found to be under 5 ms. For WAN experiments, we allocate the machines across the US-East and US-West zones – we measure the maximum latency to be roughly 75 ms.

Evaluation. We implement four applications, which are described in detail in Section 6. We find that our system delivers real-time performance, comparable to the likes of related systems such as Zexe, despite offering the added security properties. At a high-level, in all of our applications, we observe Jigsaw's latency – measured to be the time between the user action and submitting the transaction to the smart contract for settlement – to be roughly 2 seconds in addition to any application-specific MPC computation. Since we do not study application-specific MPC protocols, we do not report their measurements – there are decades of research in practical MPC protocols for matching[15] and auctions[16], and we wish to adopt them to run on Jigsaw.

Recall the various steps in creating a Jigsaw transaction:

- Local Computation:

[15] Mazloom et al. [MDPB23] develop a production-level dark pool system that can process between 25-300 orders per second at 0.01-1.5 second latency, depending on various parameter and protocol-level choices.

[16] Novakovic et al. [NKGJ24]'s protocol for sealed-bid auctions takes $\leq$ 0.5 sec for a single run with 128 bidders and a price range of 1024 values.

1. Use of TurboPLONK to construct a proof π_i of the local relation R_i
2. Proving equality of shared witnesses aux_i and pd_i between the extended witness committed in π_i and the linking commitment $\mathsf{cm}_{\mathrm{cp},i}$
3. Computing secret shares for the MPC servers, and proof of correct secret sharing.

- Collaborative Computation:

1. Use of collaborative PLONK to construct a proof π_{MPC} of the collaborative relation R_{MPC}.
2. Proving equality of shared witnesses aux_i and pd_i between the extended witness committed in π_{MPC} and the linking commitment $\mathsf{cm}_{\mathrm{cp},i}$

We now study the individual performance characteristics of each of these steps in more detail.

7.1 Local Proving

The local computation has three distinct steps listed above, all of which can be predominantly computed in parallel (once the extended witness is computed within the TurboPLONK program). Interestingly, the first two steps take roughly 0.9 secs each (across all our applications), while the last step takes < 0.1 secs, thus making the total local computation roughly 1 sec when run with sufficient CPU cores, such as on a Macbook Pro. Let us dive deeper.

7.1.1 Proving Local Relation

As one would expect, the running time of the TurboPLONK prover depends on the circuit size, which in turn is affected by the choice of cryptographic primitives (e.g. hash functions, PRFs, etc.), and the choice of parameters such as:

- Merkle tree depth: a large fraction of the circuit in R_i handles the Merkle path verification, which is comprised of a number of hash function evaluations (Pedersen hashing in our case) equivalent to the depth of the Merkle tree of coin commitments.
- payload size: since they are hashed as part of the coin commitment, the circuit size also depends on the number of field elements comprising the coin's payload. We find this to be small in practice; for instance, financial apps only require coins to encode two field elements: the token type and amount.

Figure 4 reports the number of PLONK constraints and the prover running time. We let the prover avail all available CPU cores; across our experiments, we find roughly 400-500% CPU utilization, indicating a 4-5x speedup on multicore compared to a single CPU. If one is restricted to running on a single core, the running time across all parameter choices is measured to be between 3.8 - 4.5 secs.

7.1.2 Proving Consistency of Linking Commitment

The local computation must also prove consistency of shared witnesses – namely the payload pd_i and auxiliary input aux_i – between the extended witness for the relation R_i and the opening to $\mathsf{cm}_{\mathrm{cp},i}$. Since both the extended witness and the shared witness are encoded as polynomials and committed to using KZG commitments, we use a straightforward PLONK-style permutation argument described in Appendix B. The running time for this step depends on the degree of the two polynomials, and we provide some measurements in Figure 5. Across our applications, this proof needs under 0.9 secs to construct.

Merkle Depth	Gate Count		Running Time (sec)	
	payload: 4 $\mathbb{F}$	payload: 8 $\mathbb{F}$	payload: 4 $\mathbb{F}$	payload: 8 $\mathbb{F}$
16	7942	9186	0.773	0.802
24	9170	10414	0.795	0.809
32	10398	11642	0.802	0.845
48	11626	12870	0.824	0.881

Figure 4: Prover running time and circuit size, varying: (1) the Merkle tree depth, and (2) number of fields in the record payload

R_i size	$\mathsf{cm}_{\mathbf{cp},i}$ size	Prover Time		Memory	Proof Size
		1 CPU	4 CPUs		
4096	128	1.51 s	0.45 s	27.4 MB	400 B
	256	0.91 s	0.26 s	26.1 MB	400 B
8192	128	4.83 s	1.42 s	26.6 MB	400 B
	256	2.88 s	0.82 s	27.1 MB	400 B
16384	128	16.8 s	4.61 s	27.11 MB	400 B
	256	9.35 s	2.67 s	27.31 MB	400 B

Figure 5: Proving consistency of linking commitment: running time measured for both single-core and multi-core execution

7.1.3 Proving Correctness of Secret Sharing

Recall that we prove correctness of secret sharing by proving consistency between the KZG commitments to the secret-sharing polynomial and the polynomial underlying $\mathsf{cm}_{\mathrm{cp},i}$. That is, we prove equality of the polynomial evaluations at select locations, using a prescribed permutation check similar to above. The secret-sharing polynomial has degree n, which is the number of MPC servers; the polynomial underlying $\mathsf{cm}_{\mathrm{cp},i}$ is typically of small degree (less than 16 in our applications). The use of smaller degree polynomials makes this proof relatively cheap to compute; see Figure 6 for some sample measurements. While the proof size is not of paramount concern here – note that this proof is not present in the final transaction but only verified by the MPC servers – it is nevertheless small: 400 + 64 n bytes.

n	$\mathsf{cm}_{\mathbf{cp},i}$ size	Prover Time	Proof Size
4	64	30.3 ms	656 B
	128	85.8 ms	
	256	263 ms	
16	64	16.3 ms	1424 B
	128	33.4 ms	
	256	86.4 ms	

Figure 6: Proving correct secret sharing to n MPC servers: this is a relatively inexpensive component of the local proof generation.

7.2 Collaborative Proving

Figure 7 demonstrates the efficiency of our collaborative SNARK generation phase. Despite being an MPC computation, the running time is modest, and sufficient providing users with a real-time experience. The key driver for this efficiency is our protocol design, which only places the absolutely-necessary application-specific logic, namely the function f from section 5.3, within the collaborative SNARK relation. That is, the relation does not contain any protocol-specific cryptographic operations like hashing, PRFs or commitments, which keeps the circuit both small and low-depth. As a result, not only is the extended witness generation faster (fewer network round-trips), but the entire protocol has lower memory and bandwidth requirements (due to lower degree polynomials).

App	Parameter	Ext Witness Gen		Plonk Proof Gen	
		LAN	WAN	LAN	WAN
Swap	2 parties	0.26 s	1.08 s	1.14 s	1.84 s
Auction	50 bids	1.02 s	2.22 s	2.10 s	2.85 s
	100 bids	1.81 s	3.07 s	2.10 s	2.85 s
Lottery	100 entries	0.09 s	0.16 s	0.13 s	0.88 s
	1000 entries	0.10 s	0.17 s	0.25 s	1.03 s
Voting	10 voters	0.09 s	0.16 s	0.15 s	0.91 s
	100 voters	0.11 s	0.18 s	0.25 s	1.03 s
	1000 voters	0.17 s	0.26 s	1.38 s	2.21 s

Figure 7: Performance measurements of collaborative SNARKs: we report the execution time of the two main sub-protocols, extended witness generation and collaborative PLONK proving.

7.3 Verification: Time, Gas Fee, and Proof Size

Since verification happens on-chain, it is imperative to study its cost, both in terms of CPU running time and EVM gas cost.[17] The verification complexity varies across applications, mainly owing to the difference in the number of local proofs to be verified in each transaction. For instance, the auction application can see hundreds of inputs, each with a proof to be verified, whereas the atomic swap only has two inputs. Before we report the verification costs for each application, we first study the breakdown by components:

- Each local proof is a TurboPLONK proof of size 2200 bytes, and its verification requires roughly 50 millisecs on a single CPU or 432K gas cost to verify in EVM.
- The collaborative proof is a PLONK proof of size 654 bytes, and its verification requires roughly 15 millisecs on a single CPU or 283K gas cost to verify in EVM.
- Each linking proof (one per input record) is 400 bytes, and its verification requires roughly 4 millisecs on a single CPU or 189K gas cost to verify in EVM.

While we do not implement them, we find several avenues for further optimization in gas cost using techniques from [BDFG20].

[17] We will assume precompiles such as EIP-1108 for lower gas cost of pairing curve operations in BN-254

7.4 Discussion and Comparison

While we do not implement it, one could perform back-of-the-envelope calculation (using the results reported by [OB22b]) of the expected running time and bandwidth requirements for a baseline solution that computes the entire relation within a collaborative SNARK protocol. Observe that the baseline approach would require non-black-box use of cryptographic primitives – including the Merkle tree, PRFs, and various commitments – within the SNARK relation, which would lead to relatively large polynomials (upwards of 10K constraints) and high multiplicative depth circuits (upwards of 250). With low bandwidth links, as expected in WAN deployments, the overheads are prohibitive.

Consider the atomic swap, an application with real-time performance needs. Our local proof computation takes about 1 second, while the collaborative prover takes 1.4 seconds in a LAN setting or 2.2 seconds in a WAN setting. Compare that to the baseline approach which (for a circuit of size 2^{14} constraints would take upwards of 60 seconds, even with a 3 gigabit LAN connection; on lower bandwidth WAN connections, we can expect the proof generation to take upwards of 170 seconds. We also expect an order of magnitude higher communication in the baseline approach, which further adds to the dollar costs in cloud deployments.

Acknowledgements

This work is supported in part by the AFOSR Award FA9550-24-1-0156 and research grants from the Bakar Fund, J. P. Morgan Faculty Research Award, Supra Inc., Sui Foundation, and the Stellar Development Foundation. Dimitris Kolonelos was also supported in part by a Berkeley Center for Responsible, Decentralized Intelligence (RDI) Fellowship. Part of this work was done while the second author was at NTT Research and Purdue University.

References

[ac22] arkworks contributors. `arkworks` zksnark ecosystem. `https://arkworks.rs`, 2022. 33

[BAZB20] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020*, volume 12059 of *LNCS*, pages 423–443, February 2020. 6

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014. 6

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964. IEEE Computer Society Press, May 2020. 3, 4, 5, 6, 7, 15, 20, 25

[BCT21] Aritra Banerjee, Michael Clear, and Hitesh Tewari. zkhawk: Practical private smart contracts from mpc-based hawk. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 245–248. IEEE, 2021. 3, 5

[BDF21] Carsten Baum, Bernardo David, and Tore Kasper Frederiksen. P2DEX: Privacy-preserving decentralized cryptocurrency exchange. In Kazue Sako and Nils Ole Tippenhauer, editors, *ACNS 21, Part I*, volume 12726 of *LNCS*, pages 163–194, June 2021. 6

[BDFG20] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Efficient polynomial commitment schemes for multiple points and polynomials. *Cryptology ePrint Archive*, 2020. 36

[BG12] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 263–280, April 2012. 43

[BK] Christopher Bender and Joseph Kraut. Renegade whitepaper. 4, 30

[BSCG+14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014. 4, 6, 20

[BT22] Aritra Banerjee and Hitesh Tewari. Multiverse of hawkness: A universally-composable mpc-based hawk variant. *Cryptography*, 6(3):39, 2022. 3, 5

[ByCDF23] Carsten Baum, James Hsin yu Chiang, Bernardo David, and Tore Kasper Frederiksen. Eagle: Efficient privacy preserving smart contracts. LNCS, pages 270–288, April 2023. 3, 5

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2075–2092. ACM Press, November 2019. 9, 13, 14

[CL06] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 78–96, August 2006. 11, 21

[CLMZ23] Alessandro Chiesa, Ryan Lehmkuhl, Pratyush Mishra, and Yinuo Zhang. Eos: Efficient private delegation of zkSNARK provers. In *USENIX Security 2023*, pages 6453–6469. USENIX Association, August 2023. 6

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002. 9

[CZK+19] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. pages 185–200, 2019. 3, 5, 6

[DEF+19a] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1102–1120. IEEE, 2019. 33

[DEF+19b] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to

private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120. IEEE Computer Society Press, May 2019. 28

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662, August 2012. 28

[DSD07] Augusto Jun Devegili, Michael Scott, and Ricardo Dahab. Implementing cryptographic pairings over barreto-naehrig curves. In *International Conference on Pairing-Based Cryptography*, pages 197–207. Springer, 2007. 33

[FFR24] Antonio Faonio, Dario Fiore, and Luigi Russo. Real-world universal zksnarks are non-malleable. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3138–3151, 2024. 29

[GGJ+23] Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. zkSaaS: Zero-knowledge SNARKs as a service. In *USENIX Security 2023*, pages 4427–4444. USENIX Association, August 2023. 6

[GKK+22] Chaya Ganesh, Hamidreza Khoshakhlagh, Markulf Kohlweiss, Anca Nitulescu, and Michal Zajac. What makes fiat-shamir zkSNARKs (updatable SRS) simulation extractable? LNCS, pages 735–760, 2022. 29

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, August 2021. 27, 33

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 581–612, August 2017. 21, 29

[GW20] Ariel Gabizon and Zachary J Williamson. Proposal: The turbo-plonk program syntax for specifying snark programs, 2020. 5, 33

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. 12, 27, 43, 44

[HBHW22] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, 2022. 25, 29, 33

[Kel20] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1575–1590, New York, NY, USA, 2020. Association for Computing Machinery. 5, 33

[KGC+18] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1353–1370. USENIX Association, August 2018. 3, 5, 6

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society Press, May 2016. 3, 5, 6

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194, December 2010. 12, 28, 43

[LSH+24] Yunqi Li, Kyle Soska, Zhen Huang, Sylvain Bellemare, Mikerah Quintyne-Collins, Lun Wang, Xiaoyuan Liu, Dawn Song, and Andrew Miller. Ratel: MPC-extensions for smart contracts. In *ASIACCS 24*. ACM Press, June 2024. 6

[LZW+24] Xuanming Liu, Zhelei Zhou, Yinghao Wang, Jinye He, Bingsheng Zhang, Xiaohu Yang, and Jiaheng Zhang. Scalable collaborative zk-SNARK and its application to efficient proof outsourcing. Cryptology ePrint Archive, Report 2024/940, 2024. 6

[MDPB23] Sahar Mazloom, Benjamin Diamond, Antigoni Polychroniadou, and Tucker Balch. An efficient data-independent priority queue and its application to dark pools. *Proceedings on Privacy Enhancing Technologies*, 2023. 33

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society Press, May 2013. 6

[nc24] noir contributors. domain specific language for zero knowledge proofs. https://github.com/noir-lang/noir, 2024. 33

[NKGJ24] Andrija Novakovic, Alireza Kavousi, Kobi Gurkan, and Philipp Jovanovic. Cryptobazaar: Private sealed-bid auctions at scale. *Cryptology ePrint Archive*, 2024. 33

[OB22a] Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-SNARKs: Zero-knowledge proofs for distributed secrets. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 4291–4308. USENIX Association, August 2022. 4, 8, 15, 27

[OB22b] Alex Ozdemir and Dan Boneh. Experimenting with collaborative zk-SNARKs: Zero-Knowledge proofs for distributed secrets. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4291–4308, Boston, MA, August 2022. USENIX Association. 5, 14, 33, 37

[SBBV22] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin T. Vechev. ZeeStar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy*, pages 179–197. IEEE Computer Society Press, May 2022. 3, 5, 6

[SBG+19] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1759–1776. ACM Press, November 2019. 3, 5, 6

[SBV22] Samuel Steffen, Benjamin Bichsel, and Martin T. Vechev. Zapper: Smart contracts with data and identity privacy. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2735–2749. ACM Press, November 2022. 3, 5, 6

[SWA23] Ravital Solomon, Rick Weber, and Ghada Almashaqbeh. smartFHE: Privacy-preserving smart contracts from fully homomorphic encryption. pages 309–331, 2023. 3, 5, 6

[tc24] taceo contributors. Tooling for creating collaborative snarks for circom and noir circuits. https://github.com/TaceoLabs/co-snarks, 2024. 4, 5, 8, 33

[WBB20] Barry WhiteHat, Jordi Baylina, and Marta Bellés. Baby jubjub elliptic curve. *Ethereum Improvement Proposal, EIP-2494*, 29, 2020. 27, 33

[Woo14] Daniel Davis Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014. 3

[XCZ+23] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. VeriZexe: Decentralized private computation with universal setup. In *USENIX Security 2023*, pages 4445–4462. USENIX Association, August 2023. 3, 5, 6

[YCW+24] Yunbo Yang, Yuejia Cheng, Kailun Wang, Xiaoguo Li, Jianfei Sun, Jiachen Shen, Xiaolei Dong, Zhenfu Cao, Guomin Yang, and Robert H Deng. Siniel: Distributed privacy-preserving zk-snark. *Cryptology ePrint Archive*, 2024. 6

A Security Proof

We show that *Jigsaw* is a secure doubly private smart contract scheme according to the definition in Section 4.

We assume that the simulator simulates a setup to generate all the desired public parameters along with the trapdoors for the zkSNARK, NIZK, SoK before the protocol begins.

We start by describing a simulator $\mathcal{S}$ who will interact with the $\mathcal{F}_{\mathsf{DPSC}}$ to simulate an indistinguishable view for the adversary in the ideal world.

Simulator $\mathcal{S}$. Let tx be a transaction that a corrupted party is asked to post on chain. Given this transaction, the simulator proceeds as follows:[18]

- **Case 1: The transaction is invalid.** If the $\mathsf{Verify}^{f,\mathbf{L}}(\mathsf{pp},\mathsf{tx}) = 0$, output abort are terminate.
- **Case 2: The transaction is valid.** If the $\mathsf{Verify}^{f,\mathbf{L}}(\mathsf{pp},\mathsf{tx}) = 1$:
  1. Parse the transaction as $\mathsf{tx} = \{\mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*\}_{i=1}^n, \mathsf{memo} = \bot, \star = \big(\{\mathsf{root}_i, \mathsf{cm}_{\mathsf{cp},i}, \pi_i, \delta_i, \sigma_i\}_{i=1}^n, \pi_{\mathsf{MPC}}, f\big)$.
  2. **Simulating messages sent from honest clients to malicious servers:** For each honest client C_i:
     (a) Create a tentative record honestly as in real world.
     (b) Samples a random $\underline{\mathsf{cp}_i}$ and set the linking commitment $\mathsf{cm}_{\mathsf{cp},i}$ to be the same as in tx.

[18] Note that the simulator needs to store and keep track of all the commitments, serial numbers public keys that it encounters to avoid duplicate or unauthorized use. For simplicity we assume that this can be handled as usual and omit these consistency checks from the description of the simulator.

(c) Honestly compute secret shares of $\underline{\mathsf{cp}_i}$ and $\widetilde{\mathsf{pd}_i}^*$. For shares of pd_i, compute secret shares of 0.
(d) Use the simulator of the underlying signature of knowledge scheme to simulate an SoK σ_i.
(e) Honestly compute $\mathsf{cm}_{\mathsf{SS},i}$ using the secret shares computed above. For each corrupt server S_j, simulate NIZK $\pi_{i\to j}$.
(f) Use simulator of the underlying zkSNARK to simulate π_i

For each corrupt server S_j, the simulator simulates sending the following to the adversary:

$$\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathsf{cp},i}, \pi_i, [\mathsf{pd}_i]_j, [\widetilde{\mathsf{pd}}_i^*]_j, [\mathsf{aux}_i]_j, [\underline{\mathsf{cp}_i}]_j, \sigma_i, \mathsf{cm}_{\mathsf{SS},i}, \pi_{i\to j}$$

3. **Extracting corrupt clients' inputs:** For each corrupted client C_i:
   (a) The simulator receives the following from C_i for each honest server S_j. $\mathsf{root}_i, \mathsf{sn}_i, \widetilde{\mathsf{cm}}_i^*, \mathsf{cm}_{\mathsf{cp},i}, \pi_i, [\mathsf{pd}_i]_j, [\widetilde{\mathsf{pd}}_i^*]_j, [\mathsf{aux}_i]_j, [\underline{\mathsf{cp}_i}]_j, \sigma_i, \mathsf{cm}_{\mathsf{SS},i}, \pi_{i\to j}$. It uses σ_i to extract $\mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*, \mathsf{aux}_i, \rho_i, \widetilde{\rho}_i, \mathsf{apk}_i, \widetilde{\mathsf{apk}}_i, \underline{\boldsymbol{r}_i}, \underline{\widetilde{\boldsymbol{r}}_i}$. Check if the extracted values satisfy relation R_i, if not output abort and terminate.
   (b) It then computes $\widetilde{\mathsf{pd}}_i = \delta_i + \widetilde{\mathsf{pd}}_i^*$.
4. **Invoking ideal functionality:** For each corrupt client C_i, the simulator first checks whether $\sigma_i, \mathsf{root}_i, \pi_i$ and $\pi_{i\to j}$ for each honest server j verifies. If so, it simulates sending $\perp$ to $\mathcal{F}_{\mathsf{DPSC}}$. Otherwise, the simulator invokes $\mathcal{F}_{\mathsf{DPSC}}.\mathsf{Execute}$ on inputs $f, (\boldsymbol{r}_i, \mathsf{sn}_i)_{i\in\mathcal{I}}$, $\left(\widetilde{\mathsf{cm}}_i^*, \widetilde{\mathsf{apk}}_i, \widetilde{\mathsf{pd}}_i\right)_{i\in\mathcal{I}}, \mathsf{aux}_i\}_{i\in\mathcal{I}}, \mathsf{memo} = \perp$, where $\boldsymbol{r}_i = (\mathsf{cm}_i, \mathsf{apk}_i, \mathsf{pd}_i, \rho_i, \underline{\boldsymbol{r}_i})$. and $\mathcal{I}$ is the set of corrupt clients. The ideal functionality returns $\{\widetilde{\boldsymbol{r}}_i\}_{i\in\mathcal{I}}$ and $\{[\widetilde{\mathsf{pd}}_i]_j\}_{i\in[n],j\in\mathcal{J}}$, where $\mathcal{J}$ denotes the set of corrupt servers.
5. **Simulating Adversary's view in the MPC execution:** The simulator proceeds as follows:
   (a) It uses $\{[\widetilde{\mathsf{pd}}_i]_j\}_{i\in[n],j\in\mathcal{J}}, \{\delta_i, \mathsf{aux}_i, \widetilde{\mathsf{pd}}_i^*\}_{i\in\mathcal{I}}$ and $\{[\widetilde{\mathsf{pd}}_i^*]_j, [\mathsf{aux}_i]_j\}_{i\in[n]}, j\in\mathcal{J}$ to run the simulator of the underlying MPC protocol to simulate an indistinguishable view for the adversary and output for the honest parties in this protocol.
   (b) It uses π_{MPC}, $\left(\{\delta_i\}_{i=1}^n, f, \{\mathsf{cm}_{\mathsf{cp},i}\}_{i=1}^n\right)$, $\left(\{\mathsf{pd}_i, \widetilde{\mathsf{pd}}_i^*, \widetilde{\mathsf{pd}}_i, \mathsf{aux}_i\}_{i\in\mathcal{I}}\right)$, $\{\underline{\mathsf{cp}_i}\}_{i=1}^n$ and $\left(\{[\mathsf{pd}_i]_j, [\widetilde{\mathsf{pd}}_i^*]_j, [\widetilde{\mathsf{pd}}_i]_j, [\mathsf{aux}_i]_j, \underline{\mathsf{cp}_i}\}_{i\in\mathcal{I},j\in\mathcal{J}}\right)$ to run the simulator of the underlying colzkSNARK scheme to simulate an indistinguishable view for the adversary and output for the honest parties in this protocol.

Indistinguishability Argument. We now prove that the view generated by the above simulator is indistinguishable from an adversary's view in the real world.

Recall that in case the simulator encounters an invalid transaction, it outputs abort and terminates. From soundness of the zkSNARK, NIZK and SoK, it follows that such invalid transactions make it to the chain with negligibly small probability. Observe that this holds even if all servers are malicious. Therefore, for the rest of this proof we are only going to focus on the case where tx is valid. We prove indistinguishability via a sequence of hybrid arguments:

- Hybrid 0: This is identical to the real world.
- Hybrid 1: This is almost identical to Hybrid 0, except that we simulate the setups for zkSNARK, NIZK, SoK. From zero-knowledge property of these schemes, it follows that this hybrid is indistinguishable from Hybrid 0.

- Hybrid 2: This hybrid is almost similar to the previous one, except that instead of honestly computing the zkSNARK, NIZK, SoK, we simulate them. From zero-knowledge property of these schemes, this hybrid remains indistinguishable from Hybrid 1.
- Hybrid 3: In this hybrid, we switch to using the simulator for the underlying colzkSNARK scheme. Indistinguishability between this and the previous hybrid follows from t-zero-knowledge property of colzkSNARK.
- Hybrid 4: In this hybrid we replace honestly computed commitments with commitments to garbage values. Indistinguishability between hybrids 3 and 4 follows from the hiding property of the underlying commitment scheme.
- Hybrid 5: This hybrid is identical to the simulator described above. Indistinguishability between this and the previous hybrid follows from the MPC scheme's security.

Hence, we have shown that the view of the adversary in the real and ideal worlds is indistinguishable.

UC Security.

B Commit-and-Prove PLONK

Here, we informally describe our Commit-and-Prove PLONK instantiation. We, first, discuss a necessary building block, a succinct argument that 'links' two KZG-based commitments. Then we show how this argument can be directly used to provide an efficient commit-and-prove variant of PLONK.

B.1 Argument for Linking KZG Committed Values

Let two KZG [KZG10] commitments:

$$\mathsf{cm}_1 = \mathsf{KZG.Com}(x_1 L_1(X) + \ldots + x_m L_m(X); r) = \mathsf{KZG.Com}(f(X))$$

$$\mathsf{cm}_2 = \mathsf{KZG.Com}(y_1 L_1(X) + \ldots + y_n L_n(X); s) = \mathsf{KZG.Com}(g(X))$$

where $m = c \cdot n$ and f and g are interpolated respectively over sets of roots of unity:

$$\Omega = \{\omega^0, \ldots, \omega^{m-1}\}$$

$$\Phi = \{\phi^0, \ldots, \phi^{n-1}\}$$

We want to provide an argument of knowledge that $f(\omega^{i_0}) = g(\phi^0)$, ..., $f(\omega^{i_{n-1}}) = g(\phi^{n-1})$, for some known a-priori indices $i_0, \ldots, i_{n-1}$. In plain words, that all the interpolated values of the small-degree polynomial are the same as values of the large-degree polynomial in prescribed positions (i_0, i_1, etc.).

For this, we first compute a committed permuted polynomial $f'(X)$ where

$$f'(\omega^0) = f(\omega^{i_0}),\ f'(\omega^c) = f(\omega^{i_1}), \ldots,\ f'(\omega^{c(n-1)}) = f(\omega^{i_{n-1}}).$$

The rest of the values remain unchanged. The fact that f' is indeed a (prescribed) permutation of f can be efficiently proven with a permutation argument [BG12, GWC19], π_{perm} in time $O(n)$.

Now, observe that since Φ is a subfield of Ω we have that $\phi^0 = \omega^0$, $\phi^1 = \omega^c, \ldots, \phi^{n-1} = \omega^{c(n-1)}$. Therefore, proving the linking of the initial KZG commitments boils down to proving:

$$f'(\phi^0) = g(\phi^0),\ f'(\phi^1) = g(\phi^1), \ldots,\ f'(\phi^{n-1}) = g(\phi^{n-1}).$$

This can succinctly be proven with well-known techniques, in concrete it is a zero-check over Φ [GWC19], $\pi_{\mathsf{zero-check}}$.

To summarize the linking argument between cm_1 and cm_2 consists of $\mathsf{cm}_2', \pi_{\mathsf{perm}}, \pi_{\mathsf{zero-check}}$. The overall computational complexity of the argument is $O(n+m)$ and the final proof has constant size $O(1)$.

B.2 CP-PLONK

For the commit-and-prove variant of PLONK we observe that the first PLONK proof element (denote by A) is a KZG commitment to the extended witness. But the latter already contains the witness elements, therefore the elements that we want to prove opening for $\mathsf{cm}_{\mathsf{cp}}$. This means that we can 'open' $\mathsf{cm}_{\mathsf{cp}}$ by simply providing a 'linking' between A and $\mathsf{cm}_{\mathsf{cp}}$ in the proper prescribed positions. For this we can rely on the linking protocol we described above.

In conclusion our CP-PLONK is a PLONK proof together with a linking proof between A and $\mathsf{cm}_{\mathsf{cp}}$.