Ratel: MPC-extensions for Smart Contracts

Yunqi Li
University of Illinois at
Urbana-Champaign
yunqil3@illinois.edu

Kyle Soska
University of Illinois at
Urbana-Champaign
soska@ramiel.capital

Zhen Huang
Shanghai Jiao Tong University
xmhuangzhen@gmail.com

Sylvain Bellemare
The Initiative for CryptoCurrencies
and Contracts
sbellem@gmail.com

Mikerah Quintyne-Collins
HashCloak Inc.
mikerah@hashcloak.com

Lun Wang
UC Berkeley
wanglun@berkeley.edu

Xiaoyuan Liu
UC Berkeley
xiaoyuanliu@berkeley.edu

Dawn Song
UC Berkeley
dawnsong@gmail.com

Andrew Miller
University of Illinois at
Urbana-Champaign
soc1024@illinois.edu

Abstract

Enhancing privacy on smart contract-enabled blockchains has garnered much attention in recent research. Zero-knowledge proofs (ZKPs) is one of the most popular approaches, however, they fail to provide full expressiveness and fine-grained privacy. To illustrate this, we underscore an underexplored type of Miner Extractable Value (MEV), called Residual Bids Extractable Value (RBEV). Residual bids highlight the vulnerability where unfulfilled bids inadvertently reveal traders' unmet demands and prospective trading strategies, thus exposing them to exploitation. ZKP-based approaches failed to address RBEV as they cannot provide post-execution privacy without some level of information disclosure. Other MEV mitigations like fair-ordering protocols also failed to address RBEV.

We introduce Ratel, an innovative framework bridging a multi-party computation (MPC) prototyping framework (MP-SPDZ) and a smart contract language (Solidity), harmonizing the privacy with full expressiveness of MPC with Solidity 's on-chain programmability. This synergy empowers developers to effortlessly craft privacy-preserving decentralized applications (DApps). We demonstrate Ratel's efficacy through two distinguished decentralized finance (DeFi) applications: a decentralized exchange and a collateral auction, effectively mitigating the potential RBEV issue. Furthermore, Ratel is equipped with a lightweight crash-reset mechanism, enabling the seamless recovery of transiently benign faulty nodes. To prevent the crash-reset mechanism abused by malicious entities and ward off DoS attacks, we incorporate a cost-utility analysis anchored in the Bayesian approach. Our performance evaluation of the applications developed under the Ratel framework underscores their competency in managing real-world peak-time workloads.

1 Introduction

The substantial growth and expansion of cryptocurrencies, marked by a combined market cap exceeding 1 trillion USD [33], is significantly attributed to the versatility of smart contracts. These blockchain-embedded, user-defined programs have catalyzed the evolution of decentralized finance (DeFi), flourishing applications such as decentralized exchanges (DEX) and lending platforms. Though most leading DeFi projects are plaintext-operated, there's a growing demand for privacy to protect proprietary trading strategies and ward off arbitrageurs. Yet, integrating privacy while retaining the flexibility of smart contracts remains an open problem.

Commitments and zero-knowledge proofs (ZKPs) are the most popular for enhancing blockchain privacy [7, 23, 59] but fail to provide a general-purpose solution. For example, in ZKP-based sealed-bid auctions, bids remain private until the auction ends, but winner determination requires either a trusted third party to receive plaintext bids and compute results [59] or public bid opening. These approaches fall short in terms of expressiveness; as ZKPs can only be generated for known secrets, conducting computations among distrustful parties without information leakage is unfeasible. Meanwhile, they offer limited fine-grained privacy, ensuring bid secrecy until the auction's close but not beyond. The lack of post-hoc privacy is pivotal as the **residual bids** (failed or partially fulfilled bids) unwittingly expose traders' unmet demand and potential future intentions, making them susceptible to exploitation.

We propose a novel, general-purpose solution leveraging secure multi-party computation (MPC) to provide privacy for smart contracts. In this model, confidential data are shared in an encoded manner to a consortium of MPC nodes, known as the MPC committee. The MPC committee collectively carries out computation over secret shared data without revealing them and maintains confidential storage (retaining secret-shared states) alongside an existing plaintext blockchain. We introduce a programming framework called Ratel that integrates MPC extensions to existing smart contract platforms. We implement the Ratel language as a mashup of the language for MP-SPDZ [56] (a generic MPC framework) with the Solidity [86] smart contract language. This enjoys features from both existing languages, simplifying the creation of privacy-enhanced decentralized applications (DApps).

Integrating MPC prototyping frameworks [22, 41, 56, 75] with smart contract-enabled blockchains presents unique challenges: 1) MPC Framework Selection: We choose MP-SPDZ [56] due to its extensive protocols and user-friendly interfaces. In contrast, alternatives are restricted to 2 [41] or 3 [22] party computations or lack malicious security [75]; 2) Public Blockchain and MPC Committee Coordination: It's essential to seamlessly unify public and private computation/storage and ensure secure transition data between

them; 3) Ratel Language Design: The language retains features from existing interfaces of MP-SPDZ and Solidity. Developers can mark data fields as private, ensuring any derived computations remain confidential. The compiler translates the source code for diverse components, ensuring consistent data type interpretation across implementations; 4) Performance Optimization: The bottleneck of MPC is communication latency. We employ concurrent MPC task execution, which linearly scaled throughput, exemplified in the proportional overall throughput improvement of the automatic market maker (AMM) we developed with increased trading pools. While this approach adds complexity to coordination, we ensure user simplicity by abstracting concurrency control from the frontend. We also incorporate ZKP in scenarios involving secret data from a single user to optimize performance. This cuts down the communication latency, evidenced by a 10% reduction in our AMM's single trade processing time.

In enduring systems, even "honest" nodes would crash occasionally, evidenced by the node uptime data we collected from Ripple [10] where most nodes remain online for less than a year. We consider a mixed adversary model accounting for both benign and Byzantine faults. Since MP-SPDZ will abort at any fault (even a benign stop fault) occurs, we enhance MP-SPDZ software with crash fault tolerance and restart MPC without identified Byzantine MPC nodes [1]. We equip the framework with a lightweight crash-reset mechanism that allows restarted nodes to gracefully rejoin the MPC committee without halting the system. Our practical solution enables a faulty node to restore 1000 private states in about 2.6s, using 225KB bandwidth. When compiling an application's source code, the recovering code is automatically generated. Moreover, since the crash-reset mechanism consumes resources of other MPC nodes, we offer a Bayesian-based cost-utility analysis for MPC nodes to evaluate the feasibility of recovering a node, and deter the exploitation of this mechanism by Byzantine nodes.

We validate the practicality of our approach by implementing various applications using the Ratel language and evaluating their performance under real-world workloads. Two case studies are primarily highlighted: RatelSwap, a Uniswap-style [3] AMM, and RatelAuction, a Dutch-style collateral auction as used in Maker-DAO [63]. We model a 100ms one-way latency between nodes to mimic a globally distributed network scenario. RatelSwap supports 600 swaps/hour throughput under this, which we argue is adequate to handle the workloads of the most popular AMMs (Sushiswap [83], Uniswap [3] and Trader Joe [52]). This is validated by comparing it to a 3-day peak trading period (the week following the LUNA and UST collapse [68]) where the average was 300 swaps/hour, with peak hours briefly surpassing but not exceeding 700 swaps/hour. We demonstrate RatelSwap's efficacy in managing transaction bursts by testing it against 1-hour of real-world data where throughput exceeded the system's average, with temporary congestion resolving quickly as workloads decrease. For RatelAuction, we conducted a simulation using data from a large MakerDAO auction involving the auction of $65,000,000$ DAI. The computation of auction results took 98s, a duration inconsequential compared to the auction's $2,940$s lifespan.

---

[1] Other known protocols such as HoneyBadgerMPC [62] offer even stronger availability guarantees but would require greater modification to MP-SPDZ.
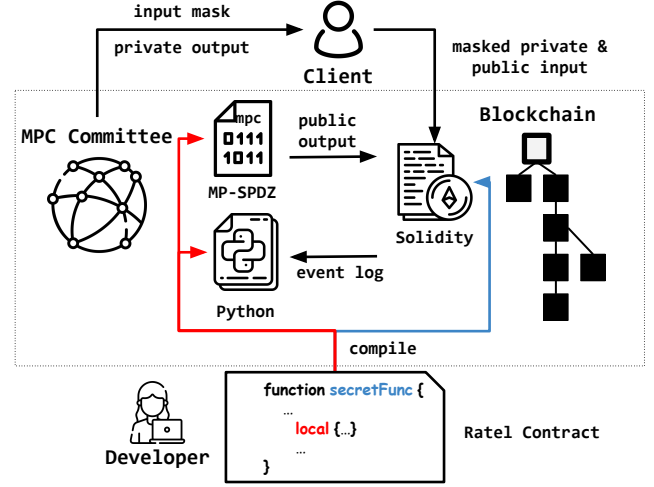

Figure 1: Ratel System Architecture

Section 2 provides essential backgrounds. The motivating issues, technical challenges, and overarching framework components are articulated in Section 3. Section 4 delves into the programming model, the crash-reset mechanism, and the applications developed. The performance of Ratel is critically evaluated in Section 5. Section 6 offers a comprehensive review of related works, and Section 7 discusses the limitations and potential areas for improvement in our study.

2 Background

Smart Contracts on Public Blockchains. Smart contracts are user-defined blockchain programs, often written in high-level languages such as Solidity [86], and compiled into bytecode for execution on the Ethereum Virtual Machine (EVM). Users modify these contracts' persistent states by sending transactions to invoke designated functions. Users can access contract states using *getter* methods or by inspecting *event logs*. Ratel is adaptable and can be seamlessly deployed on various EVM-compatible platforms, including public blockchains like Ethereum [87], layer-2 solutions like Arbitrum [53], or permissioned ecosystems like Hyperledger Fabric [8].

Shamir's Secret Sharing Based MPC. Ratel is designed to be compatible with any Shamir Secret Sharing (SSS) [77] based MPC protocols as they are currently the only practical t-of-n protocols for $n > 2$. The MPC committee consists of n MPC nodes $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ where up to t of them might be corrupted by a malicious adversary. For any secret x in the prime field $\mathbb{F}_p$, MPC nodes collaboratively share the secret by sampling a degree-t polynomial $\phi \colon \mathbb{F}_p \to \mathbb{F}_p$, where $\phi(0) = x$ and S_i knows only share $[x]^i = \phi(i)$. MPC nodes compute on shares without reconstructing any secrets. Linear combinations of shares are computed locally while multiplication requires inter-node communication. A secret is reconstructed by receiving shares from nodes and interpolating the polynomial. We follow the standard online/offline paradigm by continuously running the offline phase to maintain buffers of preprocessed elements (e.g., random bits, Beaver multiplication triples [18]) to expedite online execution.

MP-SPDZ. Multi-Protocol SPDZ [56] is a framework designed for prototyping and benchmarking MPC applications. It offers a high-level Python-like programming interface that compiles into a common bytecode, which can be executed by a variety of underlying MPC protocols under different secure models. We use its SSS protocol with honest majority and malicious corruption in a synchronous network. Adapting Ratel to other MPC protocols may require a different system design and could be promising for future research directions. MP-SPDZ extends beyond basic prime field arithmetic, enabling comparison, equality, and bit shifts. It also supports various mathematical operations, including trigonometric functions, square roots, exponentials, and logarithms, as in [6]. Both integer and fractional numbers fixed-point and floating-point) are accommodated within the framework.

3 Overview of Ratel

3.1 Motivation: The Residual Bids Problem

The transparent nature of blockchains exposes DeFi users to "front-running" by malicious actors. Miners, for instance, can manipulate transaction orders for personal gains after observing transactions in the mempool. This phenomenon, termed Miner Extractable Value (MEV) [38], is widespread today [74]. Current solutions to mitigate MEV are insufficient for broader scenarios. Highlighting this, we introduce **Residual Bids Extractable Value (RBEV)**, capturing potential losses from mismatches between intended and executed trades, which malicious actors can exploit.

MEV vs. RBEV in Uniswap: Understanding the Difference. Uniswap [3], a popular DEX, exemplifies issues with MEV and RBEV. Its trade function pseudocode is in Listing 1. The contract tracks balances for tokens A and B for each user, and the volumes of both tokens in the liquidity pool. Uniswap implements a constant-product AMM [67], where prices are determined by maintaining a constant product of the pool volumes. A trade (or swap) in Uniswap only proceeds if the received amount of token B is at least the user-defined slippage limit, amtB_min, to account for price fluctuations between the time a trade is submitted and when it is executed. The trade is canceled if this condition isn't met.

MEV arises from pre-confirmation activities, where malicious actors insert or reorder transactions to front-run existing trades. In Uniswap, they exploit this by skewing prices to the victim's slippage limit, ensuring trade execution at the least favorable price. RBEV stems from the on-chain persistence of canceled trades records (due to slippage limit breaches). This allows malicious actors to infer traders' future actions, with the problem exacerbated by the full transparency of on-chain trading strategies.

Ineffectiveness of MEV Solutions to RBEV. Mitigating MEV primarily entails restricting adversaries from utilizing the knowledge of pending transactions to manipulate their order in blocks. Fair-ordering consensus protocols [54, 55] enforce transaction ordering based on some notion of fairness, such as first-come-first-serve, thereby preventing miners from unilaterally determining the final order. However, once transactions get published, the attempted amounts and slippage limits are revealed. Other methods involve cryptographic techniques, such as ZKP or threshold encryption, to maintain trading data privacy pre-execution. However, even if bids are encoded as proofs, the verification of proofs inadvertently leaks

```
1  mapping (address => uint) public balA, balB;
2  uint public poolA, poolB;
3  function trade(uint amtA, unit amtB_min) {
4      require(balA[msg.sender] >= amtA);
5      uint amtB = poolB - poolA * poolB / (poolA + amtA);
6      require(amtB >= amtB_min);
7      poolA += amtA; balA[msg.sender] -= amtA;
8      poolB -= amtB; balB[msg.sender] += amtB;
9  }
```

Listing 1: Pseudocode for Trade on Uniswap Smart Contract

the residual bid info that arbitrageurs can leverage. This challenge underpins our advocacy for MPC as a comprehensive solution for enhancing smart contract privacy, and reducing MEV and RBEV in particular.

3.2 Technical Challenges and Design Choices in MPC-Blockchain Hybrid System

Before diving into the specifics of Ratel, we address the challenges and design choices that arise when integrating MPC and blockchain. See Appendix A.1 for a concise overview of our solutions.

Incorporate MPC within Blockchain. The first challenge is deciding the appropriate way to integrate MPC into the blockchain. There are two primary approaches: 1) every blockchain node participates in MPC (layer-1); 2) involving a smaller set of parties conducting MPC parallel to the blockchain (layer-2).

Coordination between Blockchain and MPC. MPC runs asynchronously with the blockchain, and an MPC task may be executed during a time interval that spans multiple blocks.

Programming Model for Privacy-preserving DApps Development. We aim to add MP-SPDZ as an extension to Solidity to enjoy MPC's privacy features. However, given that MP-SPDZ is designed for one-shot, stateless computation, adaptations are essential to fit our continuously running, stateful system. The integration of two individual frameworks also presents design challenges, including the transmission of instructions and data, managing both public and private data storage, and articulating confidentiality and disclosure policies.

Access Control of Client Interaction. Careful consideration must be given to processing client requests to execute MPC tasks, validating inputs received from clients as they cannot be fully trusted, and only allowing authorized clients to query private states.

Limited Performance of MPC. Although MPC allows for writing general-purpose programs, its high communication cost poses a bottleneck to the system's performance (shown in Section 5), raising concerns about its practical adoption.

Tolerating MPC Nodes Failures. Implementing a long-term practical MPC system necessitates a strategy for managing non-Byzantine node crashes and ensuring uninterrupted computation. Yet, MP-SPDZ assumes the consistent availability of all players and halts computation upon any deviation, even a benign fault. This poses a complex issue: integrating the privacy strengths of MP-SPDZ while maintaining the availability inherent in the blockchain. If a node crashes for an extended time, it can miss vital state updates, disqualifying it from future MPC tasks. Node recovery is complex because each MPC node's internal state is private and not directly replicated by others. Current solutions like Proactive Secret-sharing (PSS) schemes [19, 30, 49, 64] suggest resharing all secret states

to a new committee, leading to a quadratic communication overhead—an impractical solution given our expansive private state set. Moreover, encompassing the entire state refresh would disrupt ongoing MPC tasks. As this recovery consumes resources of other MPC nodes, it is susceptible to resource exhaustion attacks.

3.3 System Overview

Ratel is a programming framework designed to enable privacy-preserving applications, and in particular reduce MEV and RBEV. It adds MPC extensions to general purpose smart contracts. Figure 1 depicts the architecture of Ratel, its main components, and how they interact with each other. There are four types of entities in Ratel:

- The *developer* writes Ratel source code, compiles it into target programs (MP-SPDZ, Python, and Solidity), and deploys them to respective components.
- The *application contract* deployed on the EVM-compatible public *blockchain* acts as the coordinator that orchestrates the MPC committee and serves as a gateway for receiving and serializing client requests.
- The *MPC committee* consists of MPC nodes who run Python program that maintains secret-shared states and performs MPC over secret-shared data (via invoking MP-SPDZ program) following instructions from the contract.
- The *client* sends MPC requests that provide both public and private inputs and query public and authorized private states from the contract.

3.4 Workflow

We now sketch the workflow for application deployment and request execution, providing more details for Figure 1.

a) Application deployment: A developer $\mathcal{D}$ writes the Ratel program σ_R and compiles it to EVM bytecode σ_E, Python code σ_P, and MP-SPDZ bytecode σ_M, as shown in the compilation process in Figure 4b. $\mathcal{D}$ deploys σ_E to the blockchain and obtains the application contract address pk_{app}. Then, $\mathcal{D}$ queries another contract that manages the membership of MPC committee to obtain the list of MPC nodes $\mathcal{S}$, and sends σ_P, σ_M, and pk_{app} to each $S_i \in \mathcal{S}$. S_i runs the Python code σ_P to instantiate the new application. σ_P will 1) watch for any event emitted by pk_{app} and conduct MPC task. Note that each application may contain various tasks; 2) maintain preprocessing element buffers, and 3) process HTTP requests from clients for authorized secret-shared states.

b) Request execution: An MPC task execution is modeled as $(st'_{pub}, st'_{priv}) := \delta(st_{pub}, st_{priv}, req)$, where request req, consisting of $(id_{mpc}, in_{pub}, in_{priv})$, represents a composite of the MPC task identifier and the public and private inputs. To prepare req, a client C sends HTTP requests to MPC nodes to obtain input mask m to hide their private inputs, with further explanations forthcoming. C then submits req to the blockchain, triggering an event e containing both inputs to schedule the task id_{mpc}. Upon monitoring e, nodes carry out MPC with the given inputs, and apply updated private states st'_{priv} to their local confidential key-value store. The MPC committee applies st'_{pub} on blockchain to finalize the task.

3.5 Network and Adversary Model

We use an existing blockchain to coordinate MPC committee and handle the plaintext portion of applications. The blockchain serves as an authenticated broadcast channel for parties. We assume the underlying blockchain has the following properties: 1) *Availability*: Parties can interact with the blockchain by sending transactions that are committed within Δ, and once a transaction is committed, every party learns of it within Δ. 2) *Finality*: Any two parties eventually observe a consistent sequence of transactions, i.e. there are no forks. 3) *Integrity*: Clients are assumed to be able to process the global log of transactions according to EVM execution rules. Therefore, any event logs or queries observed by clients are the result of correct executions.

The adversary can fully control any number of clients. For the MPC committee, we consider a mixed adversary model similar to what is described in FHM98 [46]. Besides the blockchain, we additionally assume the adversary can actively corrupt up to t_a nodes and fail-corrupt up to t_f nodes within the group of n total MPC nodes. The consideration of crash faults captures the behavior of non-Byzantine nodes suffering from sporadic crashes in a long-running system. We assume a synchronous network where honest parties can communicate within a known maximum time Δ. Perfectly secure channels exist between parties.

3.6 Security Goals

Briefly, Ratel aims to support general-purpose programs while enforcing the following properties:

Integrity: We inherit the integrity guarantees of the underlying blockchain as well as the MPC protocol, that all computations are carried out correctly.

Confidentiality: Data stored in secret-shared namespace remains confidential unless explicitly disclosed. However, given the potential leakage of secret data during computation (for example, exposure through the reconstruction of derived values), it falls upon application developers to meticulously design application logic to safeguard confidentiality. [2]

Availability: Our availability goal is to ensure that the addition of an MPC-based sidechain does not diminish the availability of the underlying blockchain.

4 Framework

4.1 Ratel Programming Model

We explain our programming model and demonstrate most of our design features via the example Ratel code in Listing 2.

4.1.1 Building Blocks

Secure Private Input Handling. We develop an input-sharing scheme ensuring confidential and consistent sharing of private inputs with MPC nodes. Given the unreliability of clients in generating valid secret sharing, we utilize preprocessed random field elements, or *input masks*, to address this issue.

To share a secret x, C reserves an unused input mask im_{idx} from blockchain, where idx is the unique index. C then requests shares of im_{idx} from MPC nodes, reconstructs it locally, and publishes the masked value $x + im_{idx}$ to blockchain. MPC node S_i retrieves $x + im_{idx}$ from an event log and recovers their share $[x]^i = (x +$

---

[2]We generally leave anonymity and access pattern protection out of the scope of our framework.

```
1   function secretWithdraw(address token, fix amt) {
2       address user = msg.sender
3       require(amt > 0)
4       local(token, amt, user) {
5           $fix balance = load(secretBalance(token, user))
6           zkrp(amt <= balance)
7           mpc(&balance, amt) {
8               balance -= amt
9           }
10          store(secretBalance(token, user), balance)
11      }
12      publicBalance[token][user] += amt
13  }
14
15  function trade(address tokenA, address tokenB,
16  $fix amtA, $fix amtB) {
17      ...
18      disclose(key=f'price_{seq}',users=[msg.sender])
19      local(...) {...}
20      ...
21  }
```

Listing 2: Ratel Source Code Example

Receiving transaction contains $req, S_j, \{\sigma_i\}$, apply the state transition $(st'_{pub}, st'_{priv}) := \delta(st_{pub}, st_{priv}, req)$ iff

$$S_j \in \mathcal{S} \wedge |\{\sigma_i | \text{verify}(\sigma_i) = \text{True}\}| > t,$$

where MPC nodes apply st'_{priv} after k confirmations.

Figure 2: Propose(req, S_j) Subroutine: S_j **proposes request** req **with signatures** σ_i **from other MPC nodes for consensus.**

$\text{im}_{idx}) - [\text{im}_{idx}]^i$. In this way, private inputs can be posted to the blockchain alongside public inputs.

Access Control for Client Queries. While clients can access public states via getter methods or event logs, we propose an access control mechanism that enables authorized clients to query private states. By default, these states are inaccessible to external users and only open to the application contract's internal logic. However, in some cases like in a privacy-preserving AMM, traders need to know their swaps' price, requiring an authorization mechanism for specific private state queries.

We enforce the access control policy on the blockchain, while off-chain retrieval handles the shares. The comparisons with other design choices are provided in Appendix A.2. For instance, the disclose function, as shown in Listing 2 line 18, allows traders to query the final price. When the blockchain approves a query request, it notifies the MPC nodes to prepare encrypted shares for the queried value. The blockchain ensures consistency by coordinating MPC and query requests together. Clients can then obtain the encrypted shares through private HTTP requests to access the queried value.

MPC Committee Consensus through Blockchain. We primarily use on-chain consensus for simplicity, though high gas costs could warrant off-chain alternatives. The blockchain consensus aids in updating public MPC outputs and replenishing preprocessed element buffers. Our process involves a single MPC node proposing a request to the blockchain, which is executed upon majority approval. This "one-node-propose" abstraction is also useful in crash-reset scenarios. We illustrate the consensus process in Figure 2, which involves only 1 transaction to blockchain.

Management of Preprocessed Element Buffers. Nodes in the MPC committee maintain buffers for various preprocessed elements, including random bits, random field elements, Beaver triples, input masks (for client private inputs), and state masks (for private states

Initial State: $\text{buf}_{\text{input mask}} = \{\text{im}_1, \ldots, \text{im}_{i-1}, \text{im}_i, \ldots, \text{im}_j\}$ with the first $i - 1$ elements already utilized.

Procedure:

(1) Schedule: invoke Propose(req ="schedule generation of input masks with batch size B",S) IF

   $j - i < T$ (for normal refill) OR
   S is permitted to rejoin the MPC committee and its input masks are outdated (for crash reset)

   with the state updates AS

   (public) Blockchain assigns a version number v to the upcoming batch of input masks
   (private) Active MPC nodes invoke the adapted offline program to generate B input masks $\{\text{im}'_1, \text{im}'_2, \ldots, \text{im}'_B\}$.

(2) Finalize: invoke Propose(req ="finalize generation of input masks with version v",S) IF input mask generation program completes, with the state updates AS (private) $\text{buf}'_{\text{input mask}} =$

$$\begin{cases} \{\text{im}_1, \ldots, \text{im}_j, \text{im}'_1, \ldots, \text{im}'_B\}, & \text{normal refill} \\ \{\text{im}_1, \ldots, \text{im}_k, \text{im}'_1, \ldots, \text{im}'_B\}, & \text{crash reset}, i \leq k \leq j \end{cases}$$

Figure 3: Input Mask Buffer Maintenance: for both normal refill and crash reset The pre-set parameter T **ensures the buffer is replenished in time.** $k \geq i$ **as clients can still reserve input masks during off-chain generation.**

recovery). We would like to reuse MP-SPDZ's offline program however they are designed to produce elements specifically for a given MPC program, leading to an equal number of all types of elements generated although their actual usage might differ. We enhanced the MP-SPDZ code to allow the generation of individual types of elements to cater to distinct needs efficiently, optimizing resource use. We show the input mask buffer maintenance as an example in Figure 3, which may also be invoked by the crash-reset mechanism as detailed in Section 4.2.2.

4.1.2 Source Language

The Ratel language is designed to extend the functionality of Solidity by incorporating MP-SPDZ for additional data confidentiality, while still utilizing the familiar syntax and structure of Solidity. And the Python mid-layer serves as a bridge between Solidity and MP-SPDZ. As Solidity programs define smart contracts as objects with functions (methods), Ratel adheres to this convention by specifying which portion of each Solidity function should be executed off-chain by MPC nodes. Ratel inherits the expressiveness of the respective languages in each portion.

A **function** in Ratel follows the code format illustrated in 4a. The outer layer is Solidity code that runs on the blockchain. The code inside the **local** function header is executed by MPC nodes on their local machines, where the code within the **mpc** function represents MP-SPDZ code to perform complex operations over secret-shared data and the rest are mostly Python code. Ratel supports private data types, indicated by a leading **$**. Private data remain hidden until they are explicitly revealed in an mpc function. Ratel introduces the following new statements and maintains the remaining code consistent with the semantics of Solidity, Python, and MP-SPDZ:

- local(arg1,arg2,...) where arguments represent data transmitted from the blockchain to the MPC committee (line 4).

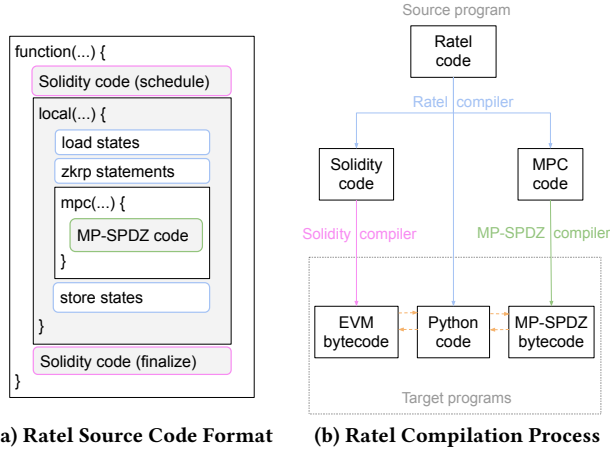**(a) Ratel Source Code Format**   **(b) Ratel Compilation Process**

Figure 4: Solidity and MP-SPDZ compilers are integrated into the Ratel compiler to generate targeted bytecode. The orange dotted arrows represent the information flow between the target programs.

- mpc(arg1,arg2,...) where arguments denote input data to the one-shot MPC. The reference-typed arguments (e.g., &balance) are outputs of the computation (line 7).
- load(key) and store(key,value) enables MPC nodes to access the secret-shared key-value store. key is common to nodes and value are respective shares of the state (line 5, 10).
- zkrp(expr1 op expr2) allows clients to provide proofs for conditions concerning their private inputs and known secret states. It validates (in)equality op between arithmetic expressions expr1, expr2.
- disclose(key,users) authorizes a list of users to query the private state associated with the given key.

4.1.3 Compilation Process

The Ratel compiler converts the high-level source program into three components, as depicted in Figure 4b. The Ratel code is first parsed into Solidity and off-chain portions, with the latter further divided into Python and MP-SPDZ code. The parsing is performed in a per-function manner.

Code Parsing. The Solidity code is broken down into distinct schedule and finalize functions. For instance, the Ratel function secretWithdraw translates into secretWithdrawSchedule and secretWithdrawFinalize. Clients employ schedule for arranging the MPC task; MPC nodes utilize finalize to write back public outputs and trigger side effects on blockchain like updating public account balances. load, zkrp, and store statements and code triggering the MP-SPDZ program are contained within a single Python function. If the finalize Solidity code isn't empty, the Python code to activate finalize is generated, enabling MPC nodes to alter public states.

Data Type Interpretation. Ratel handles variable type interpretation across components. Take amtA in the trade function for example. It is of type $fix (a private fixed-point number). In Ratel, a fixed-point number x is represented by an integer $y = x \cdot 2^\kappa$ with precision κ. The multiplication of two fixed-point numbers equates to an integer multiplication with subsequent truncation of κ bits. This approach simplifies the interpretation of private fixed-point numbers down to managing private integers. Each private input,

as detailed in Section 4.1.1, is masked by an input mask. In the Solidity code, amtA is interpreted as a pair of uint (unsigned 256-bits integer), denoting the masked value and the input mask's index. In the Python code, nodes use this index to locate the input mask share in the key-value store and then recover the input to a secret share, a field element that is treated as int type. For MP-SPDZ, it supports sfix and sint for $fix and $int respectively.

Facilitating Scheduling, Task Execution and Crash Reset. The schedule function in Solidity assigns each MPC task a sequence number seq, determining the ordering among MPC tasks and also client queries. Nodes typically use seq to mark task completion, and in crash-reset, it assists in private state recovery (see Section 4.2.2). Each schedule function emits an event to relay key information like seq, id_{mpc}, and client inputs from blockchain to MPC committee. Symmetrically, Python code is equipped with event monitoring code to allow MPC nodes to scan event logs in confirmed blocks and extract MPC task data. To execute MPC on secret-shared data, the Python and MP-SPDZ programs communicate via generated interfaces, ensuring smooth input transmission and output retrieval. Moreover, the interface facilitating crash-reset is generated, which persists input data on Solidity contract and allows the Python program to automatically determine the keys each MPC task has updated by just referencing seq.

4.1.4 Optimizations

Zero-knowledge Range Proof Module. Our initial Ratel iteration highlighted a challenge with the extensive communication rounds needed for comparisons in MPC programs, evident in scenarios like the secretWithdraw function where the amount withdrawn must not exceed the private account balance. To address this, we incorporate a Zero-Knowledge Range Proof (ZKRP) module, enabling clients to demonstrate their private inputs and known secret-shared states satisfying application-defined conditions without revealing actual values, as shown in Listing 2 line 6, which enhances efficiency.

ZKRP statements take the form zkrp(expr1 op expr2), with op as a comparison operator and expr1, expr2 as arithmetic expressions over both public and private values. We simplify this to expr>0, leading to a ZKRP scheme expressed as $ZoK\{(expr, r) : expr \in [0, \lfloor \frac{p}{2} \rfloor), C = g^{expr}h^r\}$, following the Camenisch-Stadler notation [25], with p as the field prime and r as an input mask. Assuming only linear operations in expr, client C gets input mask r, computes commitment C, and creates a valid range proof prf. Each S_i calculates share $[expr]^i$ locally and collaboratively opens C to verify prf. Complex operations in expr, like multiplication over secret values, require extra input masks and commitments(as in Appendix A.3).

The compiler provides an interface for client proof provision, in the Solidity function header and event declaration, and server-side verification, in Python that calls our modified Bulletproofs [24] (the most efficient protocol for range proofs) Rust library. When multiple ZKRP statements are present, the openings of commitments are performed together, thus requiring only 1 communication round. ZKRP statements are verified in the Python program, halting execution upon encountering false conditions, before any mpc and store statements are executed. As a result, invalid ZKRP statements result in no state update at all.

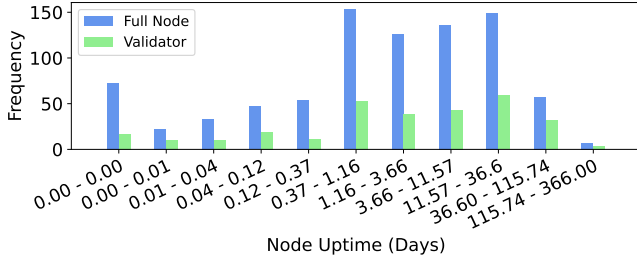**Figure 5: The empirical uptime distribution for Ripple (XRP) full nodes (amateurs) and validators (professionals) as of 10/31/2021.**

Concurrency Control. Ratel optimizes throughput by allowing parallel execution of MPC tasks, supported by a concurrency control mechanism. However, tasks accessing the same field in the key-value store must be executed sequentially. Each task's sequence number, reflecting its blockchain position and transaction fee, determines its execution order. This ensures tasks with higher fees are prioritized.

Concurrency is managed through a locking mechanism. As the monitor coroutine in Python receives a new MPC task, it acquires separate locks for each accessed field in the task. Locks are distributed on a first-come, first-served basis, with priority given to tasks with smaller sequence numbers. This ensures that competing tasks are executed in the order determined by the blockchain, and non-competing tasks, whose execution order has no impact on the final states, can be executed out-of-order. Unneeded locks are promptly released to maintain efficiency. Ratel achieves concurrent execution by assigning different ports to tasks via a port assignment scheme orchestrated on blockchain.

In summary of Section 4.1, assuming an uncompromised MPC committee, if the Ratel program is posted on-chain, all honest MPC nodes would faithfully run the local program. Node actions, such as MPC task execution and buffer refills, are instructed by the blockchain, ensuring seamless coordination and conflict-free operation among MPC nodes. Tasks adhere to an all-or-nothing principle, where they fully complete and update both public and private states or result in no changes. Robust input-sharing and access control mechanisms limit malicious client behavior; inputs placed on the blockchain ensure uniform outcomes across nodes. Should invalid inputs be posted, the corresponding MPC task will fail, leaving the system state intact, yet the malicious client incurs transaction fees.

4.2 Optimizing System Uptime: Enhancing the Availability of the Framework

A straightforward integration of MP-SPDZ with blockchain encounters availability issues. Firstly, protocols in MP-SPDZ with malicious security adhere to the malicious-with-abort model, halting upon deviation detection [56]. The prerequisite of all pairs of MPC nodes to establish sockets before proceeding (none can be absent) further constrains its applicability. Therefore we have to refine the MP-SPDZ implementation. Second, during an extended operating period, occasional failures are anticipated even under benign conditions, as illustrated by the uptime data of nodes in the Ripple [10] network (Figure 5). The network's reliance on a small group of MPC nodes for stability, reliability, and integrity mirrors Ratel's structure. With even professional validating nodes rarely surpassing a year of uninterrupted uptime, we're motivated to facilitate the seamless re-entry of crashed nodes upon restart. This model also captures the case when rare network partitions occur, allowing lagging nodes to sync with the rest of the network.

4.2.1 Enhancement to MP-SPDZ

Upon detection of any deviation, Ratel restart the halted MPC without the identified faulty nodes. Given that each restart proceeds with a reduced set of nodes, completion is assured over time. We enhance the crash fault tolerance of MP-SPDZ by masking socket failures via message buffering, which allows MPC nodes to write and read messages asynchronously, thereby enabling MPC programs to proceed without having to wait for slower or absent MPC nodes.

4.2.2 Crash-Reset Mechanism

Given that message buffers have limited size, MPC nodes would cease to send messages to nodes with overfilled buffers—particularly those absent for an extended period. In these scenarios, the crash-reset mechanism is used to recover missing (private) state updates, enabling the crashed node to re-engage in upcoming MPC tasks.

Problem Statement. A crashed node S_j, whose local secret-shared key-value store is outdated and thus inhibits its involvement in future MPC tasks due to invalid initial states st_{priv}, sends a crash-reset request to currently active nodes $\mathcal{S}_A$ ($S_j \notin \mathcal{S}_A$). Our mixed adversary model requires the presence of more than t nodes to proceed (elaborated in Section A.6, with $t = t_a$ in the basic case and $t = 2t_a$ otherwise, where t_a is the number of Byzantine faults). $\mathcal{S}_A$ evaluate the request based on a cost-benefit analysis. If approved, S_j updates its state to mirror the interim changes and gear up for upcoming tasks. We elucidate the core elements in the mechanism, with Figure 6 illustrating the workflow involved.

Private State Recovery. For sporadic recoveries involving a few nodes (usually just one), utilizing proactive secret sharing to distribute all secret-shared states to a new committee can be overly intricate. Thus we introduce a lightweight private state recovery mechanism outlined below.

Recovering a single secret-shared state $[v]^j$ with key u is similar to the client input process. Utilizing a *state mask* sm_{idx} (a specific random field element for private state recovery), S_j sends a request that includes key u and state mask index idx to $\mathcal{S}_A$ then interpolates the responses $\{(i, [v]^i + [sm_{idx}]^i)\}_{i \neq j}$ to recover $[v]^j$. The single-state recovery can be extended for batch-state recoveries by including multiple keys and corresponding state mask indexes in a single request.

State masks are produced either on-demand when a node requires state recovery or preemptively for anticipated failures. By invoking Propose(req ="reserve B state masks",S_j), $\mathcal{S}_A$ collectively generates shares of B new random field elements specifically for S_j to recover its missing states.

Missing MPC Tasks Determination. S_j identifies missed tasks by comparing its local records with the data on pub. Use T_X to denote the set of MPC tasks corresponding to the sequence numbers in set X, and $[x]$ to represent the set of positive integers up to x. Given seq_I, the total count of initialized MPC tasks available on-chain,
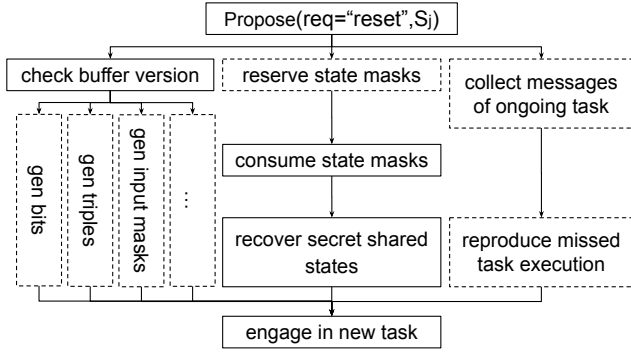
Figure 6: Crash-Reset (dotted boxes indicate optional steps).

and I_j, the set of sequence numbers of tasks S_j has locally updated before crashing, S_j identifies the tasks it missed as $T_{[\text{seq}_I]/I_j}$.

Next, S_j identifies the states it needs to recover. It uses the input data available on-chain to compute the updated states for each task in $T_{[\text{seq}_I]/I_j}$, focusing on identifying the keys associated with these states. The set of missing states for S_j is then given by $\cup_{x \in [\text{seq}_I]/I_j} st_x^1, st_x^2, \ldots$, where st_x^y represents the y-th state updated by the MPC task T_x.

Preprocessed Elements Buffer Sync. For S_j, a node offline for an extended period, lacking the latest preprocessed elements could inhibit participation in upcoming MPC tasks. Two options are available: recover missed elements via private state recovery, consuming a state mask for each element; or request to generate a new batch of elements. Considering most applications have an offline phase with fewer rounds than the online phase, eliminating the need for an extensive buffer of preprocessed elements, we favor regeneration, as illustrated in Figure 3. The generation of various preprocessed elements is optional, independent, and in parallel, contingent on the consistency of S_j's local buffer version with $\mathcal{S}_A$.

Uninterrupted Operation. A primary design objective is ensuring the crash-reset process seamlessly integrates, without interrupting ongoing tasks or inhibiting the initiation of new MPC tasks. This goal poses several challenges.

Upon approval of the crash-reset request, certain in-progress MPC tasks, yet to yield available state updates, pose a challenge. To address this, we include missed tasks $T_{[\text{seq}_I]/I_j}$ (along with state mask indexes) in the private state recovery HTTP request. Every $S_i \in \mathcal{S}_A$ maps out the states aligned with $T_{[\text{seq}_I]/I_j}$ and assigns each state a counter indicating the number of tasks in $T_{[\text{seq}_I]/I_j}$ influencing it. The counters decrease at each task's completion. The initial HTTP response to S_j only includes states with a 0 counter. As tasks progress, S_i updates these counters and offers updates each time a state's counter hits ground 0.

Once Propose(req ="reset",S_j) is confirmed, buffer sync and state recovery are triggered. In parallel, $\mathcal{S}_A$ continue to process ongoing/new MPC tasks using elements from old buffers. However, S_j may not be ready to engage in a new MPC task T_w ($w > \text{seq}_I$) due to outdated initial states or preprocessed elements. To avoid missing more tasks, $\mathcal{S}_A$ send T_w's messages to S_j, which are captured and stored by S_j for local simulation of T_w once it's ready. An edge case arises when S_j lacks the necessary shares of preprocessed elements to simulate T_w. Here, S_j issues a second HTTP request to

$\mathcal{S}_A$, offering additional state mask indexes to obtain either missing preprocessed elements or T_w's results.

4.2.3 Cost-Utility Analysis
We introduce a cost-utility analysis to weigh the resources expended by MPC nodes in aiding a crashed node's reset against the enhanced system robustness, ensuring the crash-reset mechanism isn't abused by Byzantine nodes.

Cost. Our cost analysis for recovering a crashed node is inspired by Ethereum's Gas mechanism [87]. We convert various resource considerations - time, storage, computation, networking, and economic costs - into a unified scalar metric (T, S, CPU, COM, ECO). See more details in Appendix A.4.

Utility. The MPC committee may *enter a stale state* if $t_h \leq t$, with t_h as the count of active non-Byzantine nodes. We denote reliability - the probability of avoiding stale state - as R. The utility from recovering a crashed node is quantified as enhancement in reliability (ΔR). Through Bayesian analysis, we estimate individual node reliability r_i (characterized by the distribution of failure rate λ_i), and then determine R.

We model node S_i's repeated crashes as a Poisson process. This characterization of each node's failure rate is known as the *failure rate problem* [80]. For S_i with failure rate λ_i, the likelihood of experiencing F_i failures over a time interval T is $l(F_i \mid \lambda_i, T) =$ Poisson$(x = F_i; \lambda = \lambda_i T) = \frac{\lambda^x e^{-\lambda}}{x!}$. In Bayesian analysis, λ_i is not a constant but a random variable adhering to a prior distribution. Gamma distribution is typically selected as the prior, given the Poisson-distributed likelihood function. This choice simplifies the derivation of posterior distribution, which remains Gamma-distributed, though other priors could also be applicable. The prior's probability density function (PDF) is $g(\lambda_i \mid \alpha, \beta) = $ Gamma$(x = \lambda_i; \alpha, \beta) = \frac{x^{\alpha-1} e^{-\beta x} \beta^{\alpha}}{(\alpha-1)!}$. Historical data from Ripple (Figure 5) aids in establishing this prior, specifically, in determining α and β. Supposing μ and σ^2 as the mean and variance of Ripple's validating nodes uptime, α and β are obtained by solving $\mu = E(g(\lambda_i \mid \alpha, \beta)) = \alpha\beta$ and $\sigma^2 = \text{Var}(g(\lambda_i \mid \alpha, \beta)) = \alpha\beta^2$. When F_i failures are observed over the past time frame T, the posterior PDF's hyperparameters adjust to $\alpha_i = \alpha + F_i$ and $\beta_i = \beta + T$. We estimate node S_i's failure rate with the posterior distribution mean, denoted as $\overline{\lambda_i} = \frac{\alpha_i}{\beta_i}$. The node's reliability, or the probability it remains online through time T, is calculated as $r_i = $ Poisson$(x = 0; \lambda = \overline{\lambda_i} T) = e^{-\overline{\lambda_i} T}$.

System reliability R, considering n' ($t < n' \leq n$) active nodes with reliability $r_1, r_2, \ldots, r_{n'}$, is derived by estimating the probability that over t nodes will remain online post time T. Utilizing the inclusion-exclusion principle, we get $R = \sum_{i=t+1}^{n'} (-1)^{i-t} \sum_{X \subseteq \mathcal{S}, |X|=i} \prod_{S_j \in X} r_j$. For a more efficient estimation, approximation algorithms [50] can be employed.

Mitigating Adversarial Exploitation. Unrestricted crash-reset requests can expose the MPC committee to denial-of-service attacks by malicious nodes. To differentiate between varying recovery costs, we update F_i to reflect weighted impacts instead of merely tallying crash occurrences. It also adjusts during MPC executions upon deviation detected.

We start with a common prior distribution Gamma(α, β), allowing nodes to form distinct posterior distributions Gamma(α_i, β_i) based on their specific crash histories. The variance in posterior

aids in distinguishing potential malicious nodes seeking to exploit the crash-reset mechanism.

The combined utility and cost equation $U = U_1(\Delta R) - (U_2(T) + U_3(S) + U_4(CPU) + U_5(COM) + U_6(ECO))$ evaluates the net benefit of executing the crash-reset. Each U_i describes the utility of the corresponding value. $\mathcal{S}_A$ approve if $U > 0$. The exact parameterization is highly specific to factors like MPC committee composition, blockchain setup, and the system's anticipated lifespan.

4.3 Applications

We present two DeFi applications implemented in Ratel, highlighting our solutions to privacy issues, particularly addressing RBEV. See more applications in Appendix A.5 [3].

MPC stands out for its exceptional expressiveness, and Ratel amplifies this by granting developers the liberty to mark specific data fields for privacy. However, the realized privacy level is intrinsically tied to the application's distinct logic. It's unfeasible to assess privacy guarantees in isolation from the application context. Hence, we evaluate the leakage function in applications to articulate the afforded privacy.

4.3.1 Constant Product AMM - RatelSwap

As explained in Section 3.1, Uniswap suffers from both MEV and RBEV issues. In response, we introduce RatelSwap, a privacy-enhancing variant [4] of Uniswap V2 [3]. Amounts in add/remove liquidity and trade (swap) are marked as private. This ensures associated values, including pool volumes, token balances, and bid closing prices, remain private unless explicitly revealed. For brevity, we'll focus on trade, omitting the discussion on liquidity operations.

In RatelSwap, clients transfer funds from the blockchain to the application contract, enabling MPC committee to maintain private in-application balances. Trade amounts are concealed, with signs indicating swap size (negative) and slippage limit (positive), ensuring swap directions remain undisclosed. We mitigate RBEV through batched public price discovery and delayed private price disclosure. At the end of a trade, only the trader is allowed to query whether it was fulfilled and at what price. Liquidity pool volumes and in-application account balances are concealed to prevent adversaries from inferring swap statuses based on changes in these values. Clients can query their own balances while pool volumes remain undisclosed. We strike a balance between privacy and price discovery by disclosing the average prices of recent swap batches to guide the setting of slippage limits [5]. A mandatory delay before traders can access swap prices and updated balances thwarts adversaries' attempts to deduce liquidity pool states via consecutive trades.

Consider a batch of B bids $\{bid_i = (a_i, b_i, m_i, s_i)\}_{i \in [B]}$, with bid_i submitted by $trader_i$, expressing the intention to exchange m_i units of token a_i for at least s_i units of token b_i. The outcome of executing bid_i yields c_i, with 1 for success and 0 for failure. The price of bid_i is:

$$p_i = \begin{cases} 0 & \text{if } c_i = 0 \\ s_i'/m_i & \text{if } c_i = 1 \text{ and address}(a_i) < \text{address}(b_i) \\ m_i/s_i' & \text{otherwise.} \end{cases}$$

The leakage function for trade is represented as:

$$\mathcal{L}_{\text{trade}} = \begin{cases} \{a_i, b_i\}_{i \in [B]}, \text{Mean}(\{p_i\}_{i \in [B], c_i=1}) & \text{to the public} \\ bid_i, c_i, p_i & \text{to } trader_i \end{cases}$$

Here, $\{a_i, b_i\}$ denotes an unordered pair, meaning the public is aware of the pool a trader engages with but remains uninformed about the exact swap direction. If the MPC committee remains uncompromised, the MPC nodes gain no more information than the public.

RatelSwap mitigates MEV by concealing swap size and slippage limits, preventing adversaries from identifying vulnerable transactions. Swaps details and results stay confidential post-execution, guarding against RBEV.

4.3.2 Collateral Auction - RatelAuction

In DeFi lending protocols, collateral assets are liquidated via an auction process to pay off loan debt when their prices fall. Maker-DAO [63], a popular lending protocol, employs a Dutch auction to entice bidders, where the collateral's price methodically reduces starting from an oracle-referenced initial price. An auction ends when either all collateral is bought, the loan is entirely repaid, or the auction reaches its expiry.

In such public auctions, bidders, by observing the bidding frequency, bids' size, and remaining collateral, can strategically place their bids towards the auction's end to secure the lowest price, resulting in collaterals being undervalued. This price discrimination is especially severe for high-volume auctions, which tend to have longer durations and give liquidators ample time to respond to market shifts. As per [73], by April 2021, the profit from 28,138 liquidations in MakerDAO totaled 63.59M USD. This profit stems from the gap between the auctioned collateral's price and its market value. The discrimination is an implicit form of RBEV, if we view the auction as a single massive order, with bidders trading against it. The auction "order" is at a disadvantage since the large unfulfilled amounts are public knowledge. Moreover, bidders face potential MEV issues during the auction's concluding moments; those who successfully settle their bids secure favorable positions, while others miss out.

The goal of designing a "fair" collateral auction scheme is to prompt liquidators to repay the loan at a price consistent with their true willingness to pay (usually the market price), regardless of the auction's current status. We improve the liquidation auction by marking the price and size of all bids as private and consequently make the remaining collateral volume private [6]. RatelAuction accepts bids priced above the auction's floor price. Given the descending nature of the auction's price, RatelAuction periodically checks if the cumulation of accepted bids with a price above the current price suffices to cover the debt - concluding the auction upon affirmation. Should the auction's closure yield an excess of qualified bids, preference is accorded to earlier submissions. This mechanism incentivizes fast bid placements, accelerates the auction progression, and culminates in a more favorable price for the auctioneer.

Consider a collateral auction with an expiry T and a total collateral amount V. The function $p(\cdot)$ captures the price change in the auction, with $p(t) = p(T)$ when $t \geq T$. Given that, at time t, the application has collected bids $\{bid_i = (v_i, p_i)\}$. In the original

---

[3]Notably, Ratel's RockPaperScissors implementation offers availability unattainable with standard ZKP-based approaches.

[4]Source code of RatelSwap.

[5]A similar decision has been adopted by Rialto [48] where they reveal top-k settlement prices in their double orderbook DEX.

[6]Source code of RatelAuction.

MakerDAO auction, anyone can calculate the remaining collateral volume $r(t) = \max(0, V - \sum_{p_i \geq p(t)} v_i)$, enabling them to estimate the auction's end time and discern the reason for its conclusion. However, the leakage function for RatelAuction is

$$\mathcal{L}_{\text{RatelAuction}}(t) = \begin{cases} (|\{bid_i\}|, \text{term}, V, p(t)) & \text{to the public} \\ bid_i, v'_i & \text{to trader}_i \text{ if term} = 1 \end{cases}$$

indicating that only the total count of bids collected and whether the auction was terminated or not (without the underlying reason) are disclosed. trader$_i$ learns the fulfilled volume $v'_i \leq v_i$ at the auction's end. Hiding $r(t)$ effectively prevents bidders from forming an accurate expectation regarding the auction's remaining duration.

RatelAuction minimizes RBEV and aims to bridge the 63M USD profit gap liquidators currently exploit [7]. We expect the auction price to be slightly above market price due to the auctioneer's less favorable market position. Although bid concealment doesn't completely eliminate MEV, it reduces its impact. Blind bidding encourages prioritization, expediting auction closure and yielding a fairer final price.

5 Evaluation

We implement the Ratel framework and benchmark it, finding that MPC communication latency is the primary bottleneck. Our evaluation demonstrates Ratel's capability to efficiently manage real-world workloads for RatelSwap and RatelAuction. Evaluation on crash-reset is in Appendix B.

Implementations and Optimizations. We adapted the MP-SPDZ codebase with several modifications to meet our specific needs: 1) It only supported revealing cleartext outputs to parties. We extended it to output secret shares to MPC nodes for storage in the confidential key-value store; 2) We refined its offline phase to enable the generation of specific types of preprocessed elements in designated batch sizes; 3) We enhanced it with crash fault tolerance, allowing MPC to proceed even when some parties are absent; 4) We reengineered its sockets to support asynchronous read/write, enabling nodes to send messages to absent peers who can later retrieve and simulate the MPC independently; 5) We optimized its startup phase, reducing it from over 80 to just 7 communication rounds by removing redundant socket setup processes and adding multithreading for socket setup.

Since the crash-reset involves massive Geth queries, we optimized it by employing the batch query function of the aio_eth library [69]. we also introduced multiprocessing to accelerate the private state interpolation process.

We enhanced the performance of RatelSwap through application-level optimizations detailed in Appendix B.1.

Experiment Setup. We established a private blockchain using Geth [47] and utilized its Proof-of-Authority consensus protocol, enabling us to configure block time and simulate various public blockchains. We run $n = 4$ MPC nodes with $t = 1$ on a single machine with an Intel Xeon E5-2620 v4 CPU and 128 GiB of RAM running Ubuntu 20.04. We model the MPC committee as being managed by servers from several reputable organizations, stationed in varied geographical locations. To mimic realistic network (globally

distributed) conditions [8], we incorporate a simulated latency, causing packet delays of 100 ± 5ms. Given the high simulated inter-node latency, the primary bottleneck is the per-round communication delay, not the number of nodes. This approach, excluding random network fluctuations and packet losses, assures stable benchmark results. We use 128-bit field elements, with a precision of $\kappa = 16$ for fixed-point numbers.

5.1 Bottleneck Analysis

As a framework composed of multiple components, we are curious about which part or factor is the performance bottleneck of Ratel. Evaluating a general-purpose framework like Ratel holds no significance without being contextualized within a specific application. Therefore, we benchmark Ratel using the trade operation in RatelSwap, as it is the most frequently used operation in our most representative application.

Latency. A trade request is first sent to blockchain and executed by MPC committee once the corresponding transaction is confirmed. The confirmation time of the blockchain portion varies with the configuration of blockchain being used, which dramatically ranges from a few seconds to several minutes. The Proof-of-Stake-based Ethereum network produces blocks every 12s on average, with most exchanges and merchants waiting for 10-50 confirmations thus resulting in a latency of 2-13min. Other popular EVM-compatible chains such as Binance Smart Chain [27] and Avalanche [84] have faster confirmation times of approximately 3s, and 2s respectively.

The overall off-chain execution time (to run Python program) of trade without using ZKRP is measured to be 5.9s (0.08s without the simulated network latency). Since database access takes less than 0.001s, the majority of latency comes from the MP-SPDZ program. The trade MP-SPDZ program has 48 communication rounds and the online phase takes 5.12s. The remaining 0.779s comes from socket setup which could be amortized by reusing the same sockets across multiple MP-SPDZ programs.

The offline phases are run in advance to ensure that preprocessed elements are readily available. The trade MPC program consumes 2 input masks, 876 beaver triples, and 2738 random bits. Generating different preprocessed elements can run in parallel and the time cost to generate required triples and bits takes 2.4s and 3.3s respectively, which means offline phase runs faster than online phase. Similar to the online phase, at least 0.7s are used for socket setup.

The use of ZKRP reduced the off-chain running time by 0.4s, where it takes 5.18s for the MP-SPDZ program to process all private data, plus the 0.23s for the 1 round of communication to open the commitment to secret-shared states. Our use of ZKRP requires MPC nodes to verify 3 ranged proofs and reconstruct 2 commitments taking 0.006s per ranged proof and 0.014s per commitment.

Throughput. The throughput of RatelSwap is dependent on whether trades are parallelizable off-chain and the block time and block capacity of the underlying blockchain network. In particular, trades sent to the same pool must be executed in sequence, as they access the same fields such as pool volume sizes. The throughput in a single pool is 10 trades per minute.

---

[7] In an ideal, risk-neutral competitive market, excess profit diminishes, leaving negligible profit margins for liquidators.

[8] Contrary to our methodology, related works such as P2DEX [16] and Kicking-the-Bucket [36], exhibit testing conditions that aren't reflective of a realistic DApps environment. They record server latencies of less than 25ms and ping times of 1.003ms respectively, figures that are notably optimistic when compared to the typical latencies observed in AWS instances [66].
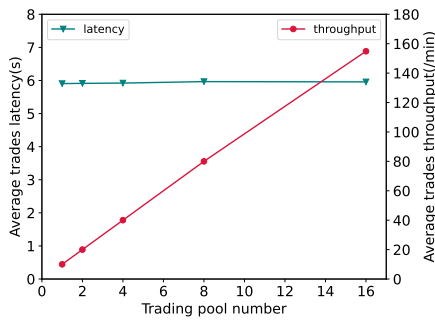
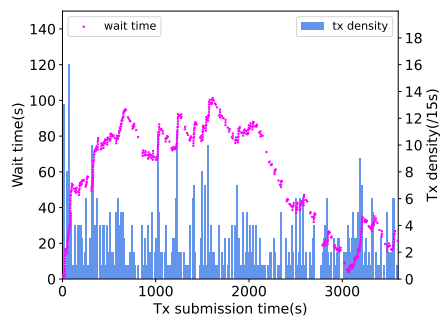**Figure 7: Performance of RatelSwap over Multiple Trading Pools (with Concurrent Execution)**

Figure 8: MPC Task Wait Time vs. Transaction Submission Frequency: simulate RatelSwap over a one-hour period of the real-world workload from the USCD.e-WAVAX pool on Trader Joe.
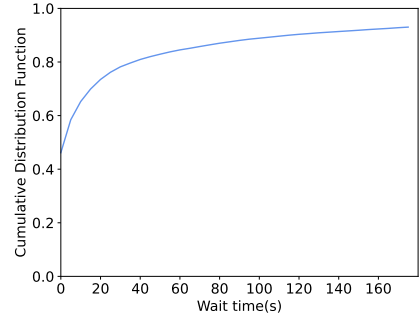
Figure 9: Wait time probability distribution of simulating RatelSwap over 3 days of historical data from the USCD.e-WAVAX pool on Trader Joe.

Figure 7 illustrates that maximum concurrency is achieved when trades, originating from distinct users, are directed to different trading pools. These trades, being data-independent, are processed in parallel. The total throughput of RatelSwap exhibits linear scalability with the increase in the number of trading pools, capped by the parallel computing capacity of the underlying hardware. Interestingly, the serial latency is minimally impacted and remains relatively constant.

A trade transaction consumes under 40k units of gas. Even when all input data is stored on-chain (for crash-reset), the cost remains a modest 204k units - comparable to the 101k and 141k units expended by FairMM and Uniswap respectively. On the Ethereum network, with a 30m unit gas limit per block, over 150 trades can be accommodated in each block, equating to a maximum throughput of 600 trades per minute. On the Binance Smart Chain, with 140m block gas limit, the peak throughput reaches about 13k trades per minute.

Economic Cost. We evaluate the financial overhead to run MPC nodes. The pure computation time for trade on the MPC committee is approximately 0.1s, with an extra 0.37s for the offline phase, and the cost of generating input masks is negligible. Considering the on-demand hourly rate for Amazon EC2 m5.xlarge instances is 0.192 USD [12], and the data exchange per trade is just a few megabytes, the overall amortized cost is negligible, compared to the average transaction fee of approximately 1 USD paid on Avalanche or more than 30 USD paid on Ethereum during our observations.

Bandwidth. The bandwidth used by MP-SPDZ program is modest and does not saturate a broadband connection, with each server transmitting 0.17 MB of data per trade.

5.2 Performance over Real-World Data

We demonstrate the effectiveness of Ratel, by collecting and analyzing the historical data of AMM and collateral auction on the public blockchains. Our results indicate that Ratel is capable of handling real-world traffic for both applications.

RatelSwap Under High-Traffic Conditions in the Most Popular Trading Pool. We monitor all transactions submitted to Sushiswap [83] and Uniswap [3] on the Ethereum network, and Trader Joe [52] on the Avalanche network, over a period of 5 days

in May 2022. This period follows the collapse of LUNA and UST, resulting in especially elevated on-chain trading activity.

Figure 11 shows the average number of swap rates over the top 40 trading pairs in these markets. The most prolific pool is the Trader Joe USCD.e-WAVAX pair on Avalanche, with 324 swaps/hour on average. This is less than the maximum throughput of RatelSwap, which is around 600 swaps/hour.

Although RatelSwap is able to handle the average sustained workload of the most prolific measured pool, it may still experience temporary backlogging during periods of acute activity. The per-hour trading history of the USCD.e-WAVAX pool during our observation period is illustrated in Figure 12. We simulated RatelSwap over the 1-st hour of our observation period, during which there are 622 swaps in total by submitting swap transactions to RatelSwap at the exact times measured in the pool. We then observed the queuing time of each swap, which is the delay between a transaction being confirmed by the blockchain and the MPC nodes starting to execute the MPC task. The results, as shown in Figure 8, indicate that during sharp bursts of transactions, the MPC committee becomes congested, and confirmed transactions may be queued for up to 101 seconds before being executed. However, when the workload decreases, the queue clears quickly, and the delay drops back to a typical wait time of less than a minute.

Figure 9 shows the cumulative distribution of the transaction delays obtained by performing repeated simulations of the entire 3-day history that we collected from the USDC.e-WAVAX pool. The simulation found that 83.77% of trades have a wait time of less than 1min with 58.44% of trades being queued for less than 5s.

RatelAuction in a Rarely Occurring Large-Scale Auction. We evaluate RatelAuction using the data of an ETH-A auction with $65,000,000$ Dai (1 Dai ≈ 1 USD) that happened on MakerDAO. There is only one multiplication in the MP-SPDZ program for bid submission, which is merely at no cost. At the end of the auction, all 49 received bids are traversed to determine the exact collateral payouts for each bid. The traverse takes 98s, which is acceptable compared with its 2940s lifetime. Moreover, the final settlement operation is run once per auction and such large auctions are infrequent in general.

6 Related Work

Most popular DEXes [1, 3, 4, 14, 45, 51, 52, 61, 83] lack protection against MEV and RBEV. Several solutions exist to mitigate MEV by enforcing specific order, either in the consensus layer [55, 60, 88], or in the application layer [32]. However, these solutions do not address RBEV. Another approach is hiding the content of bids to mix potential victim bids with all other bids. We categorize these works according to their generality and underlying cryptographic primitives used to provide privacy in Table 3. Some works are designed specifically for DEX and can be compared with RatelSwap, while others are more general-purpose frameworks like Ratel providing privacy for blockchain.

Some works [20, 28, 42, 71, 76] use Trusted Execution Environments (TEEs) such as Intel Software Guard Extensions (SGX) [35] to emulate a trusted third-party operating the exchanges, but these have to trust that the chip manufacturers do not leave backdoors and also known side-channel have been patched. However, new attacks are still possible. ZKP-based approaches [23, 31, 43, 44, 57–59, 65, 70, 82] is efficient to provide privacy and prevent MEV without additional trust assumptions, but it is limited to specific types of DEXes as no one has the ability to provide proof of secret data from other parties. ZKP approaches also have availability issues as they rely on the data provider to be online to provide the original data whenever required.

Cartlidge et al. [26] first proposed using MPC to mitigate price impact and front-running in dark pools, followed by Kicking-the-Bucket [36] that further optimizes the matching algorithms, and Asharov et al. [11, 13] that combines MPC and homomorphic encryption to address privacy concerns. P2DEX [16] uses publicly verifiable MPC and threshold signatures to resist both front-running and secret key theft in cross-chain exchanges. It shares similarities with Rialto [48] as they both implement a double order-book DEX with fixed-size orders, considering only order prices. Rialto and Ratel are on the same path that combines MPC and ZKP, striking a balance between efficiency and functionality. However, double order-book DEXes may not be able to tolerate the high latency of MPC as they are sensitive to market price fluctuation and the sorting algorithm involves too many comparison operations, which is very expensive in MPC.

Several general-purpose frameworks have been proposed to provide privacy for blockchains using either MPC or Homomorphic Encryption (HE) schemes. Gage MPC [5] uses non-interactive MPC for short-term strong security but may leak residual bids for long-term use in a DEX. GABLE [34] uses garbled circuit schemes directly on smart contracts, but it is not practical for complex dApps on popular blockchains like Ethereum due to high costs. None of these MPC-based frameworks [17, 34, 40, 79] considers programmability or has a user-friendly front-end. White-City [79] and Eagle [17] model a private state machine replication similar to Ratel, but Eagle does not address crash reset. ZeeStar [81] and Pesca [37] provide a programming framework for non-experts to instantiate private smart contracts, similar to Ratel, they both use HE along with ZKP for privacy guarantees. However, ZeeStar uses additive HE, which limits its expressivity to linear operations over foreign data. Pesca is the most relevant work to Ratel, which uses fully HE (FHE) instead

of MPC so that private data could be stored on the mainchain for better reliability, but the tradeoff is the high storage gas expenses.

Crash Recovery in the MPC Literature. MATRIX [15] and FlexSMC [85] are MPC management and orchestration frameworks that assign MPC nodes to execute user-registered MPC programs. Both frameworks follow the traditional approach of considering stateless, one-shot computation and not storing private states for future use. FlexMPC addresses node failures and communication interruptions by using a centralized gateway to monitor servers in an MPC session and restart the session if necessary. Partisia [72] is another MPC-as-a-Service framework, which stores secret variables off-chain but does not address how secret variables are maintained over time. Proactive Secret-sharing (PSS) schemes [19, 30, 49, 64] allow secret states shared in an old MPC committee to be reshared with a new committee, but this is inefficient in our setting and can halt the system from taking on new MPC tasks. White-City [79] is a concurrent work to Ratel that supports crash-reset recovery by recording all MPC messages in the encrypted form on a public bulletin board, but this approach is not practical for mainstream blockchains like Ethereum due to high on-chain storage costs. In contrast, Ratel proposes a more lightweight recovery mechanism that only sends shares of missing states to the reset server and does not halt the system from taking on new MPC tasks.

7 Discussions and Conclusion

Ratel demonstrates how the synergy of MPC and blockchain can effectively craft privacy-preserving DApps, notably addressing the RBEV challenge. We identify key areas for future improvement of the framework.

The definition of end-to-end latency in RatelSwap is ambiguous due to batching and delayed price disclosure. While sequential ordering remains vital, latency is only problematic if it constitutes a significant part of the disclosure window. The current performance might be insufficient as trading pool activity rises. A potential remedy is segmenting the pool into sub-pools to distribute workload and letting market dynamics dictate each sub-pool's priority.

Rethinking the fee mechanism is essential due to the introduction of concurrency control through the locking mechanism. The evaluation should not only consider traditional metrics like gate numbers and circuit depth but also the count of locks and total blocking time associated with an MPC task. These dynamic factors, often discerned at runtime, influence overall performance. A model akin to Ethereum's gas mechanism, where fees are estimated upfront and states reverted if prepaid fees fall short, could be an effective approach.

We've primarily discussed managing private states within individual applications. However, our framework can be extended to support interoperability across various applications, allowing them to share private data fields. For an in-depth explanation of one possible approach, refer to Appendix C.

The current Ratel compiler offers limited support for the ZKRP module, requiring developers to discern when to substitute MPC operations with ZKRP. An enhanced Ratel compiler could auto-determine the computations suited for ZKP, optimizing MPC workload. Ultimately, a holistic integration of smart contracts, MPC, and

ZKP is a promising avenue for refining privacy and efficiency in future developments.

References

[1] 0x. 2022. 0x: Powering the decentralized exchange of tokens on Ethereum. https://www.0x.org.

[2] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. 2021. Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 740–755. https://doi.org/10.1145/3453483.3454074

[3] Hayden Adams, Noah Zinsmeister, and Dan Robinson. 2020. Uniswap v2 Core.

[4] AirSwap. 2022. AirSwap: Peer-to-peer Token Trading DEX and Open Source Developer DAO. https://www.airswap.io.

[5] Ghada Almashaqbeh, Fabrice Benhamouda, Seungwook Han, Daniel Jaroslawicz, Tal Malkin, Alex Nicita, Tal Rabin, Abhishek Shah, and Eran Tromer. 2021. Gage MPC: Bypassing Residual Function Leakage for Non-Interactive MPC. *Proceedings on Privacy Enhancing Technologies* 2021 (2021), 528 – 548.

[6] Abdelrahaman Aly and Nigel P Smart. 2019. Benchmarking privacy preserving scientific operations. In *International Conference on Applied Cryptography and Network Security*. Springer, 509–529.

[7] O Andreev, B Glickstein, V Niu, T Rinearson, D Sur, and C Yun. 2019. *ZkVM: fast, private, flexible blockchain contracts*. Technical Report. Technical report, Online.

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 30, 15 pages. https://doi.org/10.1145/3190508.3190538

[9] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. 2016. Secure multiparty computations on bitcoin. *Commun. ACM* 59, 4 (2016), 76–84.

[10] Frederik Armknecht, Ghassan O Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. 2015. Ripple: Overview and outlook. In *Trust and Trustworthy Computing: 8th International Conference, TRUST 2015, Heraklion, Greece, August 24-26, 2015, Proceedings 8*. Springer, 163–180.

[11] Gilad Asharov, Tucker Hybinette Balch, Antigoni Polychroniadou, and Manuela Veloso. 2020. Privacy-Preserving Dark Pools. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems* (Auckland, New Zealand) *(AAMAS '20)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1747–1749.

[12] AWS. 2022. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[13] Tucker Balch, Benjamin E. Diamond, and Antigoni Polychroniadou. 2021. SecretMatch: Inventory Matching from Fully Homomorphic Encryption. In *Proceedings of the First ACM International Conference on AI in Finance* (New York, New York) *(ICAIF '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 7 pages. https://doi.org/10.1145/3383455.3422569

[14] Bancor. 2022. Bancor V3 - Bancor V3 Technical Docs. https://docs.bancor.network/about-bancor-network/bancor-v3.

[15] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. 2018. An End-to-End System for Large Scale P2P MPC-as-a-Service and Low-Bandwidth MPC for Weak Participants. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 695–712. https://doi.org/10.1145/3243734.3243801

[16] Carsten Baum, Bernardo David, and Tore Kasper Frederiksen. 2021. P2DEX: Privacy-Preserving Decentralized Cryptocurrency Exchange. In *Applied Cryptography and Network Security*, Kazue Sako and Nils Ole Tippenhauer (Eds.). Springer International Publishing, Cham, 163–194.

[17] Carsten Baum, James Hsin yu Chiang, Bernardo David, and Tore Kasper Frederiksen. 2022. Eagle: Efficient Privacy Preserving Smart Contracts. Cryptology ePrint Archive, Paper 2022/1435. https://eprint.iacr.org/2022/1435 https://eprint.iacr.org/2022/1435.

[18] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology — CRYPTO '91*, Joan Feigenbaum (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 420–432.

[19] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. 2020. Can a Public Blockchain Keep a Secret?. In *Theory of Cryptography*, Rafael Pass and Krzysztof Pietrzak (Eds.). Springer International Publishing, Cham, 260–290.

[20] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. 2019. Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1521–1538. https://doi.org/10.1145/3319535.3363221

[21] Iddo Bentov and Ranjit Kumaresan. 2014. How to Use Bitcoin to Design Fair Protocols. In *Advances in Cryptology – CRYPTO 2014*, Juan A. Garay and Rosario Gennaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 421–439.

[22] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Computer Security - ESORICS 2008*, Sushil Jajodia and Javier Lopez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 192–206.

[23] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. ZEXE: Enabling Decentralized Private Computation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 947–964. https://doi.org/10.1109/SP40000.2020.00050

[24] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 315–334. https://doi.org/10.1109/SP.2018.00020

[25] Jan Camenisch and Markus Stadler. 1997. Efficient group signature schemes for large groups. In *Advances in Cryptology — CRYPTO '97*, Burton S. Kaliski (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 410–424.

[26] John Cartlidge, Nigel P. Smart, and Younes Talibi Alaoui. 2019. MPC Joins The Dark Side. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (Auckland, New Zealand) *(Asia CCS '19)*. Association for Computing Machinery, New York, NY, USA, 148–159. https://doi.org/10.1145/3321705.3329809

[27] BNB Smart Chain. 2018. BNB Smart Chain: A Parallel BNB Chain to Enable Smart Contracts. https://www.binance.org/en/smartChain.

[28] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Stockholm, Sweden, 185–200. https://doi.org/10.1109/EuroSP.2019.00023

[29] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *Advances in Cryptology – CRYPTO 2018*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham, 34–64.

[30] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. 2020. Fluid MPC: Secure Multiparty Computation with Dynamic Participants. *IACR Cryptol. ePrint Arch.* 2020 (2020), 754.

[31] Shumo Chu, Qiudong Xia, and Zhenfei Zhang. 2020. Manta: Privacy Preserving Decentralized Exchange. Cryptology ePrint Archive, Paper 2020/1607. https://eprint.iacr.org/2020/1607 https://eprint.iacr.org/2020/1607.

[32] Michele Ciampi, Muhammad Ishaq, Malik Magdon-Ismail, Rafail Ostrovsky, and Vassilis Zikas. 2022. FairMM: A Fast and Frontrunning-Resistant Crypto Market-Maker. In *Cyber Security, Cryptology, and Machine Learning: 6th International Symposium, CSCML 2022, Be'er Sheva, Israel, June 30 – July 1, 2022, Proceedings* (Be'er Sheva, Israel). Springer-Verlag, Berlin, Heidelberg, 428–446. https://doi.org/10.1007/978-3-031-07689-3_31

[33] CoinMarketCap. 2022. CoinMarketCap Total Cryptocurrency Market Cap. https://coinmarketcap.com/charts/.

[34] Christopher Cordi, Michael P. Frank, Kasimir Gabert, Carollan Helinski, Ryan C. Kao, Vladimir Kolesnikov, Abrahim Ladha, and Nicholas Pattengale. 2022. Auditable, Available and Resilient Private Computation on the Blockchain via MPC. In *Cyber Security, Cryptology, and Machine Learning*, Shlomi Dolev, Jonathan Katz, and Amnon Meisels (Eds.). Springer International Publishing, Cham, 281–299.

[35] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016 (2016), 86.

[36] Mariana Botelho da Gama, John Cartlidge, Antigoni Polychroniadou, Nigel P. Smart, and Younes Talibi Alaoui. 2022. Kicking-the-Bucket: Fast Privacy-Preserving Trading Using Buckets. In *Financial Cryptography and Data Security*, Ittay Eyal and Juan Garay (Eds.). Springer International Publishing, Cham, 20–37.

[37] Wei Dai. 2022. PESCA: A Privacy-Enhancing Smart-Contract Architecture. Cryptology ePrint Archive, Paper 2022/1119. https://eprint.iacr.org/2022/1119 https://eprint.iacr.org/2022/1119.

[38] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 910–927. https://doi.org/10.1109/SP40000.2020.00040

[39] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and Unconditionally Secure Multiparty Computation. In *Advances in Cryptology - CRYPTO 2007*, Alfred Menezes (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 572–590.

[40] Didem Demirag and Jeremy Clark. 2021. Absentia: Secure Multiparty Computation on Ethereum. In *Financial Cryptography and Data Security. FC 2021*

International Workshops, Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 381–396.

[41] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*. USENIX Association, San Diego, California, USA.

[42] DerivaDEX. 2021. Introduction to DerivaDEX Architecture. https://medium.com/derivadex/introduction-to-derivadex-architecture-1dac2910cd81.

[43] Felix Engelmann, Thomas Kerber, Markulf Kohlweiss, and Mikhail Volkhov. 2022. Zswap: zk-SNARK Based Non-Interactive Multi-Asset Swaps. https://eprint.iacr.org/2022/1002 https://eprint.iacr.org/2022/1002.

[44] Felix Engelmann, Lukas Müller, Andreas Peter, Frank Kargl, and Christoph Bösch. 2021. SwapCT: Swap confidential transactions for privacy-preserving multi-token exchanges.

[45] EtherDelta. 2022. EtherDelta. https://etherdelta.com.

[46] Matthias Fitzi, Martin Hirt, and Ueli Maurer. 1998. Trading correctness for privacy in unconditional multi-party computation. In *Advances in Cryptology — CRYPTO '98*, Hugo Krawczyk (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–136.

[47] Ethereum Foundation. 2021. Go Ethereum. https://github.com/ethereum/go-ethereum.

[48] Kavya Govindarajan, Dhinakaran Vinayagamurthy, Praveen Jayachandran, and Chester Rebeiro. 2022. Privacy-Preserving Decentralized Exchange Marketplaces. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, Shanghai, China, 1–9. https://doi.org/10.1109/ICBC54727.2022.9805505

[49] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. 2022. Storing and Retrieving Secrets on a Blockchain. In *Public-Key Cryptography – PKC 2022*, Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe (Eds.). Springer International Publishing, Cham, 252–282.

[50] Klaus D Heidtmann. 1982. Improved method of inclusion-exclusion applied to k-out-of-n systems. *IEEE Transactions on Reliability* 31, 1 (1982), 36–40.

[51] IDEX. 2022. IDEX High-Performance Decentralized Exchange. https://docs.idex.io.

[52] Trader Joe. 2023. Trader Joe Decentralized Exchange. https://traderjoexyz.com

[53] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, BALTIMORE, MD, USA, 1353–1370.

[54] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. 2021. Themis: Fast, Strong Order-Fairness in Byzantine Consensus. Cryptology ePrint Archive, Paper 2021/1465. https://eprint.iacr.org/2021/1465 https://eprint.iacr.org/2021/1465.

[55] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. 2020. Order-Fairness for Byzantine Consensus. In *Advances in Cryptology – CRYPTO 2020*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 451–480.

[56] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1575–1590. https://doi.org/10.1145/3372297.3417872

[57] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. 2021. KACHINA – Foundations of Private Smart Contracts. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, Dubrovnik, Croatia, 1–16. https://doi.org/10.1109/CSF51468.2021.00002

[58] Rami Khalil, Arthur Gervais, and Guillaume Felley. 2019. TEX - A Securely Scalable Trustless Exchange. Cryptology ePrint Archive, Paper 2019/265. https://eprint.iacr.org/2019/265 https://eprint.iacr.org/2019/265.

[59] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, IEEE, San Jose, CA, USA, 839–858.

[60] Klaus Kursawe. 2020. Wendy, the Good Little Fairness Widget: Achieving Order Fairness for Blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies* (New York, NY, USA) *(AFT '20)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3419614.3423263

[61] kyberswap. 2022. KyberSwap Docs. https://docs.kyberswap.com.

[62] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and Its Application to Anonymous Communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 887–903. https://doi.org/10.1145/3319535.3354238

[63] MakerDAO. 2022. The Maker Protocol's Collateral Auction House (Liquidation System 2.0). https://docs.makerdao.com/smart-contract-modules/dog-and-clipper-detailed-documentation.

[64] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. 2019. CHURP: Dynamic-Committee Proactive Secret Sharing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2369–2386. https://doi.org/10.1145/3319535.3363203

[65] Fabio Massacci, Chan Nam Ngo, Jing Nie, Daniele Venturi, and Julian Williams. 2018. FuturesMEX: Secure, Distributed Futures Market Exchange. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 335–353. https://doi.org/10.1109/SP.2018.00028

[66] matt adorjan. 2023. CloudPing - AWS Latency Monitoring. https://www.cloudping.co/grid#

[67] Vijay Mohan. 2022. Automated market makers and decentralized exchanges: a DeFi primer. *Financial Innovation* 8, 1 (2022), 1–48.

[68] Q. ai-Powering a Personal Wealth Movement. 2022. What Really Happened To LUNA Crypto? https://www.forbes.com/sites/qai/2022/09/20/what-really-happened-to-luna-crypto/

[69] Narasimha1997. 2023. aio-eth - Asynchronous JSON-RPC client for Ethereum. https://github.com/Narasimha1997/aio-eth

[70] Chan Nam Ngo, Fabio Massacci, Florian Kerschbaum, and Julian Williams. 2021. Practical Witness-Key-Agreement for Blockchain-Based Dark Pools Financial Trading. In *Financial Cryptography and Data Security*, Nikita Borisov and Claudia Diaz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 579–598.

[71] obscuro. 2021. Obscuro: Confidential Smart Contracts for Ethereum. https://whitepaper.obscu.ro/assets/images/obscuro-whitepaper-0-9.pdf.

[72] Partisia. 2021. MPC Techniques Series, Part 10: MPC-as-a-Service — the Partisia Blockchain Infrastructure. https://medium.com/partisiablockchain/mpc-techniques-series-part-10-mpc-as-a-service-the-partisia-blockchain-infrastructure-9b4833e77965.

[73] Kaihua Qin, Liyi Zhou, Pablo Gamito, Philipp Jovanovic, and Arthur Gervais. 2021. An Empirical Study of DeFi Liquidations: Incentives, Risks, and Instabilities. In *Proceedings of the 21st ACM Internet Measurement Conference* (Virtual Event) *(IMC '21)*. Association for Computing Machinery, New York, NY, USA, 336–350. https://doi.org/10.1145/3487552.3487811

[74] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–214.

[75] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *2014 IEEE Symposium on Security and Privacy*. IEEE, Berkeley, CA, USA, 655–670. https://doi.org/10.1109/SP.2014.48

[76] SecretSwap. 2021. SecretSwap Update: AMM Rewards + Governance Token. https://scrt.network/blog/secretswap-update-amm-rewards-governance-token.

[77] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[78] Shantanu Sharma and Wee Keong Ng. 2020. Scalable, On-Demand Secure Multiparty Computation for Privacy-Aware Blockchains. In *Blockchain and Trustworthy Systems*, Zibin Zheng, Hong-Ning Dai, Mingdong Tang, and Xiangping Chen (Eds.). Springer Singapore, Singapore, 196–211.

[79] Omer Shlomovits. 2020. White-City: A Framework For Massive MPC with Partial Synchrony and Partially Authenticated Channels. https://github.com/ZenGo-X/white-city/blob/master/White-City-Report/whitecity_new.pdf.

[80] JK Shultis, DE Johnson, GA Milliken, and ND Eckhoff. 1981. *GAMMA: a code for the analysis of component failure rates with a compound Poisson-gamma model. Final technical report*. Technical Report. Kansas State Univ., Manhattan (USA). Dept. of Nuclear Engineering.

[81] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. 2022. ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zeroknowledge Proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, IEEE, SAN FRANCISCO, CA, & ONLINE, 1543–1543.

[82] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. 2019. Zkay: Specifying and Enforcing Data Privacy in Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1759–1776. https://doi.org/10.1145/3319535.3363222

[83] SushiSwap. 2022. SushiSwap. https://dev.sushi.com/docs/intro.

[84] Dmitry Tanana. 2019. Avalanche blockchain protocol for distributed computing security. In *2019 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*. IEEE, 1–3.

[85] Marcel von Maltitz, Stefan Smarzly, Holger Kinkelin, and Georg Carle. 2018. A management framework for secure multiparty computation in dynamic environments. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Taipei, Taiwan, 1–7. https://doi.org/10.1109/NOMS.2018.8406322

[86] Maximilian Wohrer and Uwe Zdun. 2018. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, Campobasso, Italy, 2–8.

[87] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[88] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine Ordered Consensus without Byzantine Oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, virtual, 633–649. https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao

A Design Details

A.1 Challenges and Solutions

Incorporate MPC within Blockchain. The first challenge is deciding the appropriate way to integrate MPC into the blockchain. There are two primary approaches: 1) every blockchain node participates in MPC (layer-1); 2) involving a smaller set of parties conducting MPC parallel to the blockchain (layer-2).

For Ratel, we opt for the layer-2 approach, utilizing a separate and smaller set of nodes, the MPC committee, operate a "sidechain". The MPC committee is assumed to consist of a consortium of reputable parties. Private contract states are moved into a confidential zone backed by the MPC committee, and MPC nodes store these values encoded as secret shares. Computations involving private data are carried out using MPC, specifically using the MP-SPDZ framework as the backend. We choose the layer-2 approach as the stringent requirements (e.g., intensive computation, high bandwidth, and consistent online presence necessitated by the complex recovery processes on the MPC "sidechain") on MPC nodes are unsuitable for regular nodes in a permissionless blockchain.

Coordination between Blockchain and MPC. MPC runs asynchronously with the blockchain, and an MPC task may be executed during a time interval that spans multiple blocks.

In Ratel, off-chain MPC tasks are scheduled by the blockchain [9], which ensures that the order of MPC tasks is well-agreed upon. The blockchain also helps with deploying the MPC committee, such as authenticating the list of MPC nodes and ensuring preprocessed elements are available before a task is scheduled to run.

Programming Model for Privacy-preserving DApps Development. We aim to add MP-SPDZ as an extension to Solidity to enjoy MPC's privacy features. However, given that MP-SPDZ is designed for one-shot, stateless computation, adaptations are essential to fit our continuously running, stateful system. The integration of two individual frameworks also presents design challenges, including the transmission of instructions and data, managing both public and private data storage, and articulating confidentiality and disclosure policies.

When designing the Ratel language, we start with a straightforward mashup. Programmers explicitly annotate sections of their code designated for MP-SPDZ, while the rest belongs to Solidity programs, so the compiler can simply split them. [10] We introduce a mid-layer in Python that runs off-chain by every MPC node. This intermediary layer maintains a key-value store for confidential secret-shared data, persisting private states between rounds. It maintains preprocessed element buffers to ensure adequate supply for the online phase. It actively monitors event logs to extract information from the blockchain and wrap up transactions invoking callback methods to finalize MPC tasks on-chain. Ratel provides developers with the flexibility to define disclosure policies. By default, access to the confidential storage is limited to functions defined by the same application. However, allowances can be made to disclose data to authorized users identified by their addresses.

Access Control of Client Interaction. Careful consideration must be given to processing client requests to execute MPC tasks, validating inputs received from clients as they cannot be fully trusted, and only allowing authorized clients to query private states.

The blockchain acts as a gateway for clients to submit MPC requests. Requests are formulated as transactions and serialized using the consensus core of blockchain. We adopt a robust input-sharing scheme to ensure that clients can post masked private inputs on-chain and guarantee that MPC nodes receive consistent shares of inputs. Access control for client queries is enforced through the mechanism implemented on the blockchain.

Limited Performance of MPC. Although MPC allows for writing general-purpose programs, its high communication cost poses a bottleneck to the system's performance (shown in Section 5), raising concerns about its practical adoption.

To address this challenge, we focus on two aspects: 1) optimizing Ratel's performance and 2) identifying suitable applications that require high expressiveness despite high latency. We propose a concurrency control mechanism for executing MPC tasks in parallel to improve throughput, and a Zero Knowledge Range Proof (ZKRP) module to reduce expensive comparison operations in MPC. We present two applications: 1) an AMM where the price of swaps is determined by their position in the waiting queue, which is ordered by the blockchain and unaffected by MPC, and 2) an auction where the computation-intensive settling phase is insignificant compared to the long bid collecting phase.

Tolerating MPC Nodes Failures. Implementing a long-term practical MPC system necessitates a strategy for managing non-Byzantine node crashes and ensure uninterrupted computation. Yet, MP-SPDZ assumes the consistent availability of all players and halts computation upon any deviation, even a benign fault. This poses a complex issue: integrating the privacy strengths of MP-SPDZ while maintaining the availability inherent in the blockchain. If a node crashes for an extended time, it can miss vital state updates, disqualifying it from future MPC tasks. Node recovery is complex because each MPC node's internal state is private and not directly replicated by others. Current solutions like Proactive Secret-sharing (PSS) schemes [19, 30, 49, 64] suggest resharing all secret states to a new committee, leading to a quadratic communication overhead—an impractical solution given our expansive private state set. Moreover, encompassing the entire state refresh would disrupt ongoing MPC tasks. As this recovery consumes resources of other MPC nodes, it is susceptible to resource exhaustion attacks.

We enhanced the MP-SPDZ networking code to mask socket-level errors and disconnections. If a node is unavailable to receive messages, the sender buffers the message until the socket becomes available or the buffer is full. We modified MP-SPDZ to resume the MPC execution, excluding nodes identified as faulty, ensuring task completion. A lightweight crash-reset mechanism is introduced, enabling specific nodes to restore their lost private state updates and re-engage in upcoming MPC tasks without disrupting ongoing operations. Integrated within the programming framework, this mechanism automatically identifies the set of missing states for efficient recovery. Furthermore, we provide a Bayesian-based cost-utility analysis that identifies and mitigates the influence of nodes exhibiting fault patterns indicative of abuse, thus limiting their impact on the system.

---

[9]We primarily use on-chain primitives for simplicity, though off-chain alternatives, executed by consensus among MPC nodes, could be considered if gas costs become a concern.

[10]Approaches like Viaduct [2] that automate this partitioning according to high-level instructions would be complimentary.

A.2 Reasoning About Access Control for Client Queries

A fully on-chain scheme for access control is expensive as it requires a transaction to be sent to the blockchain, and MPC nodes need to encrypt their respective shares of the queried value under the client's public key and post the encrypted shares back to the chain. On the other hand, a fully off-chain scheme suffers from synchronization issues, as the private state transition keeps happening in the MPC committee, and due to network latency, the time when MPC nodes receive off-chain queries might differ. Clients have to specify the version of the private outputs to obtain consistent shares. Thus we chose a design in the middle.

A.3 Further Details about Zero-Knowledge Range Proofs

MPC nodes Open Commitment The server S_i has shares $[x]_i$, $[r]_i$ and they need to compute $C = g^x h^r$. They locally calculate $g^{[x]_i} h^{[r]_i}$ and broadcast it. After the broadcast, S_i has a list of points $\{(x = j, y = g^{[x]_j} h^{[r]_j})\}_{j \in \mathbb{Z}_n^*}$. For simplicity, we assume the first $t + 1$ points come from honest servers. Then they get the value

$$g^x h^r = \prod_{j=1}^{t+1} y_j^{l_j(0)}$$

$$= \prod_{j=1}^{t+1} g^{[x]_j * l_j(0)} h^{[r]_j * l_j(0)}$$

$$= g^{\sum_{j=1}^{t+1} [x]_j l_j(0)} h^{\sum_{j=1}^{t+1} [r]_j l_j(0)},$$

where $\{l_j(x)\}_{j \in \mathbb{Z}_{t+1}^*}$ is Lagrange basis.

Multiplication Proof in ZKRP For multiplication in ZKRP, we need to provide proof for $z = xy \in [0, R)$, where both x and y are secret-shared states and c is a public range. We omit the range proof for $z \in [0, R)$ and only show the proof for relation $z = xy$.

Client request input masks r_x and r_y from MPC nodes. Clients do the following things:

- samples random field elements $r_z, k_x, k_y, k'_x, k'_y, k'_z \xleftarrow{\$} \mathbb{F}_p$.
- $C_x = g^x h^{r_x}, C_y = g^y h^{r_y}, C_z = C_x^y h^{r_z} = g^{xy} h^{r_x y + r_z}$.
- $K_x = g^{k_x} h^{k'_x}, K_y = g^{k_y} h^{k'_y}, K_z = C_x^{k_y} h^{k'_z}$.
- compute challenge using Fiat-Shamir heuristic $c \xleftarrow{\$} H(K_x | K_y | K_z)$ where H is a hash function.
- $s_x = cx + k_x, s'_x = cr_x + k'_x, s_y = cy + k_y, s'_y = cr_y + k'_y, s'_z = cr_z + k'_z$.
- post (C_z, prf) to the blockchain, where

$$prf = (K_x, K_y, K_z, s_x, s_y, s'_x, s'_y, s'_z).$$

On the server side, they do:

- collaborate to open C_x and C_y.
- check $g^{s_x} h^{s'_x} \overset{?}{=} C_x^c K_x$.
- check $g^{s_y} h^{s'_y} \overset{?}{=} C_y^c K_y$.
- check $C_x^{s_y} h^{s'_z} \overset{?}{=} C_z^c K_z$.

A.4 Crash-reset Cost Analysis

The time required to assist a crashed server to rejoin the protocol is roughly $max(t_{buffer}, t_{state})$ where t_{buffer} is the time to sync preprocesed element buffer and t_{state} is the time required to generate up to $|S|$ state masks and recover $|S|$ states. The input

mask generation costs are the same for every out-of-date node and require B parallel rounds of communication and an on-chain transaction of fixed size whose cost depends on the market for gas prices of the chain. The amount of CPU and network overhead for state recovery scales linearly with the amount of out-of-date states $|S|$ which will depend on the specific workload and number of missed tasks. Fortunately, this operation is highly parallelizable, and each state field can be recovered simultaneously. For many realistic workloads the same state fields will be updated multiple times so $|S|$ will be sublinear in the number of tasks that have been missed.

A.5 Other Applications Implemented in Ratel

Rock Paper Scissors This is a two-player lottery game that requires financial fairness [9, 21] and additionally privacy [59]. Beyond that, the MPC committee provides data availability so that the first player does not need to wait for another player to show up and reveal their inputs in person. Each player in the game submits their action (rock, paper, or scissors) in the secret sharing form. At the end of the game, an MPC function is invoked to compare their actions without revealing what the actions are. The only information revealed is the winner of a game, whereas the behavior patterns of the players remain hidden.

Double Orderbook Exchange Volume matching is a practical algorithm to realize a double orderbook exchange. Buy and sell orders are matched in terms of volume, while the price is provided by an external price oracle. Volume matching is widely used in dark pools, in which the volumes of orders are hidden to reduce the price impact. We follow the same design as in [26] and also integrate it with a blockchain payment system. The matching process involves the computation of private data from multiple sellers and buyers, which is not suitable for the use of ZK-based approaches. We implement volume matching to demonstrate the feasibility of achieving a double-orderbook exchange, while other forms of matching algorithms (continuous double auction, periodic auction, etc.) are also achievable by the MPC committee framework.

Supply Chain We implemented a database application that collects statistics on private user data. Users input their personal information into the database, but only authorized or paid parties can access the information of other users. The MPC committee framework enables fine-grained disclosure policies that maximize flexibility while ensuring privacy, which also provides a practical means of enabling government audits and compliance policies.

A.6 Security Analysis

The Basic Case. We start with the security analysis considering a synchronous network and an adversary capable of statically compromising up to t_a (Byzantine) nodes and dynamically crashing t_f non-Byzantine nodes, assuming $t_f + 2t_a < n$ (due to their limited capability and our effective crash-reset mechanism) [11] and t_a, t_f are

---

[11]The adversary can simultaneously sustain the crash state of t_f nodes. To crash an additional node, they must relinquish control over one of the already compromised t_f nodes. Moreover, the adversary's ability to induce a crash in another node is outpaced by the crash-reset process.

known parameters, as we described in Section 3.5. Our implementation is based on MP-SPDZ framework and uses its Shamir-based protocol for malicious honest majority [12]. The corresponding MPC protocol is CGH+18 [29], ensuring security against a static malicious adversary with honest majority. The protocol is secure with abort and lacks fairness. We argue the security goals defined in Section 3.6 are achieved during normal MPC task execution and crash-reset paths by setting the threshold of Shamir secret sharing to t_a.

Ratel inherits integrity from the underlying blockchain and CGH+18 protocol during the input, MPC execution, and finalize phases. As for crash-reset, by conducting the private state recovery scheme using verifiable secret sharing (VSS), Byzantine nodes are incapacitated from preventing MPC nodes from receiving correct shares. For confidentiality, in the input phase, masked data posted on-chain remain private since malicious nodes cannot reconstruct the associated input masks. Throughout the MPC execution, an honest majority ensures the protocol's security. During crash-reset, Byzantine nodes do not learn any extra information. Availability is guaranteed in a synchronous network where a minimum of $t_a + 1$ honest nodes consistently participate. Our enhancement to MP-SPDZ facilitates recovery from failed MPC executions by initiating a rerun with a refined set of MPC nodes, excluding identified faulty ones. For crash-reset, the continuous online presence of at least t_a+1 non-faulty nodes aids the swift progress catch-up of recovering nodes.

Further Analysis We also explore the possibility of relaxing the strong assumptions made in the basic case.

Under synchrony but with an unrestricted t_f (the adversary may crash any number of non-Byzantine nodes), safety (confidentiality and integrity) can be guaranteed while availability may not when there are insufficient non-Byzantine nodes online.

Safety firstly hinges on the active participation of at least $2t_a + 1$ MPC nodes to ensure an honest majority. On the other hand, given that Byzantine nodes can feign crashes, at least $2t_a + 1$ non-Byzantine nodes are essential to advance in the presence of dormant Byzantine entities. Therefore, $3t_a < n$ is necessary to guarantee safety.

However, the protocol can only proceed when $3t_a < n - t_f$. Otherwise, the system enters a stale state (no availability), halting MPC executions and barring benign node recovery. In this case, a system "reboot" is initiated to restore normal operations from *last checkpoint* (the last on-chain finalized MPC task), as the finalization implies the successful storage of output states by a minimum of $2t_a + 1$ non-Byzantine nodes. This reboot is contingent on the online status of a specific set of non-Byzantine nodes, or alternatively, computations can resume from a preceding checkpoint, albeit necessitating repeated computation.

Given the indefinite nature and unpredictable recovery timing of benign crashes, this scenario parallels an asynchronous network subjected to t_a Byzantine faults, characterized by indeterminate honest node message delays. In these non-basic cases, the

integration of an asynchronous MPC protocol, such as HoneybadgerMPC [62], is necessary, with protocol proceeds with the fastest $2t_a + 1$ nodes.

B   More Evaluation Results

B.1   Application Level Optimization for RatelSwap

For RatelSwap, we make two optimizations specific to the trade function, as it is the most frequently used function. To ensure the validity of the amounts provided by users, we employ the ZKRP module to replace a series of comparisons and multiplications for these checks. Additionally, to avoid the complexity of a division operation, we opt to not calculate the closing price of a trade in the MPC program. Instead, we provide shares of changes in both tokens, allowing traders to independently calculate the reconstructed values.

B.2   Crash Recovery Latency Analysis

We measure the time required for a single MPC node S_4 to recover with the assistance of three other active MPC nodes $S_1 - S_3$. The overall latency is affected by various factors, including the number of states to recover and certain steps outlined in Figure 6 that may be skipped.

In the best case scenario, where S_4 still possesses consistent batches of preprocessed elements with other MPC nodes and ample state masks, S_4 only needs to receive approval from the MPC committee and recover any missing states. The time required for approval takes either one or two blockchain confirmation times, depending on the implementation of the approval process, and we call this request approval time. For state recovery, it takes an additional request approval time for S_4 to mark state masks to use on-chain, as well as the off-chain latency, which includes the sum of communication, interpolation, and database storage latency, as shown in Table 1. There is only one round trip communication between S_4 and $S_1 - S_3$. When an active MPC node receives a request from S_4, it must calculate the set of states to recover, which invokes the *eth_call* API of Geth to query input data of MPC tasks. The bandwidth listed is for a single response and the total bandwidth usage should be roughly multiplied by the number of MPC nodes n, whereas in a proactive secret-sharing scheme, the overhead needs to be multiplied by n^2.

In a worse case, extra work might be required including updating outdated preprocessed elements and reserve state masks. We assume that there is no ongoing MPC task that the MPC nodes need to wait for in order to get the outputs, as it is difficult to accurately estimate their finish time. The generation of preprocessed elements and state masks both involve on-chain triggering (1 blockchain confirmation time required) and random field element generation, and preprocessed elements generation requires 1 more confirmation time to finalize on-chain. The state mask generation must be completed before the state recovery, which will inevitably introduce extra latency. Not only offline phases for different purposes can be run concurrently, they can also run in parallel with the state recovery phase. Whichever finishes last determines the total latency of the crash recovery process. We list the time to generate random

#States	Comm w/(w/o) Latency(s)	Interpolate shares(s)	DB Store(s)	Bandwidth of Resp(Bytes)
100	0.7013 (0.2966)	0.1402	0.0012	7696
1000	2.3545 (1.7239)	0.2363	0.0067	76890
2000	4.3432 (3.3093)	0.3388	0.0113	153728
4000	7.5974 (6.3534	0.5394	0.0224	307417

Table 1: Off-chain Overheads to Recover States: The total latency is the sum of one round-trip communication, interpolating shares and storing states shares in the database. The communication time, without latency, is used to query the Geth node and calculate the set of states to recover.

Batch Size	4000	8000	16000	32000	64000
Time(s)	2.39202	2.69845	3.10843	3.38505	4.21237

Table 2: Time to Generate Input/State Masks.

```
1  contract{
2    view isReadAllowed(externalContractAddr,key)
3    view isWriteAllowed(externalContractAddr,key)
4    function authorize(addr,key,approv,allowWrite){
5        verify(approv is from admins)
6        isReadAllowed[addr][key] = true
7        if (allowWrite)
8          isWriteAllowed[addr][key] = true
9    }
10  }
```

Listing 3: Solidity Program: Access Authorization for External Contracts

Suppose an application B would like to write to field x of application A.

Application Launch:
- Deploy contract to the blockchain to get the contract address $addrB$.
- Acquire authorization from contract A. Once the authorization is ready, send request to the MPC committee to launch the new application.
- Upon receiving request, for the writeDB(key = x, owner=$addrA$, value=y) statement in B's Ratel code, since the owner is not $addrB$, MPC nodes check if $isWriteAllowed(addrB, key)$ in contract A is $True$. If not, reject to launch B.
- The writeDB(key=x, owner=$addrA$, value=y) statement is compiled to writeDB(key=$addrA$+x, value=y).

Figure 10: Interoperability Support

field elements under simulated latency with different batch size in Table 2.

In a worse-case scenario, extra work may be required, including updating outdated preprocessed elements and reserving state masks. We assume that there is no ongoing MPC task that the nodes need to wait for in order to obtain the outputs, as it is difficult to accurately estimate their finish time. The generation of preprocessed elements and state masks both invokes REQUEST (1 request approval time) and off-chain generation. The generation of preprocessed elements requires 1 more block confirmation time to finalize on-chain. The state mask generation must be completed before the state recovery, which will inevitably introduce extra latency. Whichever finishes last determines the total latency of the crash reset process.

C Application Interoperability

The unit of data management is per the application contract. There are three levels of visibility for data: 1) private: only accessible by the owner contract(application) itself. 2) external-read-only: readable by authorized contracts. 3) external-read-and-write: modifiable by authorized contracts. Every application contract specifies a set of administrators at the creation time. The administrators have the ability to authorize any external contract to access some data field. When an application intends to read or write data from another contract, MPC nodes need to check whether they have permission to access the corresponding field. More details are in Listing 3 and Figure 10.

D   Extra Code Listings

```solidity
1  event SecretWithdraw(
2    address token, uint amt,
3    address user, uint seq
4  );
5
6  event Trade(
7    uint seqTrade, address user,
8    address tokenA, address tokenB,
9    uint idxAmtA, uint maskedAmtA,
10   uint idxAmtB, uint maskedAmtB
11 );
12
13 function secretWithdraw(
14   address token, uint amt
15 ) public {
16   address user = msg.sender;
17   require(amt > 0);
18   uint seq = getUniqueSeq();
19
20   emit SecretWithdraw(token, amt, user, seq);
21 }
22
23 function publicBalanceAdd(uint amt,
24   address token, address user, uint seq
25 ) public onlyServer {
26   address server = msg.sender;
27   require(
28     publicBalanceValue[token][user][seq][server]==0);
29   require(
30     publicBalanceFinish[token][user][seq]==false);
31
32   publicBalanceValue[token][user][seq][server]=amt;
33   publicBalanceCount[token][user][seq][amt]++;
34   if (publicBalanceCount[token][user][seq][amt]>T){
35     publicBalanceFinish[token][user][seq] = true;
36     publicBalance[token][user] += amt;
37   }
38 }
39
40 function trade(
41   address tokenA, address tokenB,
42   uint256 idxAmtA, uint256 maskedAmtA,
43   uint256 idxAmtB, uint256 maskedAmtB
44 ) public {
45   require(inputMaskOwner[idxAmtA]==msg.sender);
46   require(inputMaskOwner[idxAmtB]==msg.sender);
47   ...
48   emit Trade(
49     user, tokenA, tokenB,
50     idxAmtA,maskedAmtA,
51     idxAmtB,maskedAmtB
52   );
53 }
```

Listing 4: Solidity Program

```python
1  // read data from input file
2  secretBalance = read_sfix(0)
3  amt = read_cfix(1)
4
5  enough = (secretBalance >= amt).reveal()
6
7  // write to output file
8  cint.write_to_file(enough)
```

Listing 5: MPC Program: secretWithdraw

```python
1  async def runSecretDeposit(server, log):
2    // parse log
3    token = log['args']['token']
4    amt = log['args']['amt']
5    user = log['args']['user']
6    seq = log['args']['seq']
7
8    ... // acquire all read, write, and port locks
9
10   // readDB
11   secretBalance=bytes_to_int(
12     server.db.Get(f'balance_{token}_{user}')
13   )
14
15   // prepare mpc inputs
16   with open(location_sharefile, "wb") as f:
17     f.write(
18       int_to_hex(secretBalance)
19       + int_to_hex(amt)
20     )
21
22   // execute mpc program
23   await run_online(server.serverID,
24     server.players, server.threshold, 'secretWithdraw')
25
26   // get mpc outputs
27   input_arg_num = 2
28   with open(location_sharefile, "rb") as f:
29     f.seek(input_arg_num * sz)
30     enough = hex_to_int(f.read(sz))
31
32
33   if (enough == 1):
34     // update private output to local db
35     secretBalance -= amt
36     server.db.Put(
37       f'balance_{token}_{user}',
38       int_to_bytes(secretBalance)
39     )
40
41     // upload public output to blockchain
42     tx = server.contract.functions.publicBalanceAdd(
43       amt, token, user, seq).buildTransaction()
44     sign_and_send(tx, server.web3, server.account)
45
46   // release locks
47   // mark task-seqSecretDeposit as finished
48
49  async def runTrade(server, log):
50    // parse log
51    idxAmtA=log['args']['idxAmtA']
52    maskedAmtA=log['args']['maskedAmtA']
53    idxAmtB=log['args']['idxAmtB']
54    maskedAmtB=log['args']['maskedAmtB']
55
56    // recover masked inputs
57    amtA=recover_input(db,maskedAmtA,idxAmtA)
58    amtB=recover_input(db,maskedAmtB,idxAmtB)
59    ...
```

Listing 6: Python Program

```
1  mpcext(...) {
2    ...
3    set(varName, index, value)
4    add(varName, index, seq, value)
5  }
```

Listing 7: Ratel Program: Upload Data to blockchain

```
1  function varNameSet(index, value) {
2      server = msg.sender
3      require(isServer[server])
4      prevValue = varNameValue[index][server]
5      if (prevValue) {
6          varNameCount[index][prevValue]--
7      }
8      varNameValue[index][server] = value
9      varNameCount[index][value]++
10     if (varNameCount[index][value] > T) {
11         varName[index] = value
12     }
13  }
14
15  function varNameAdd(index, seq, value) {
16     server = msg.sender
17     require(isServer[server])
18     require(varNameValue[index][seq][server] == 0)
19     require(varNameFinish[index][seq] == false)
20     varNameValue[index][seq][server] = value
21     varNameCount[index][seq][value]++
22     if (varNameCount[index][seq][value] > T) {
23         varNameFinish[index][seq] = true
24         varName[index] += value
25     }
26  }
```

Listing 8: Solidity Program: Upload Data to blockchain

E   Notations

|	
$[t]n$	the number of MPC nodes
t	general fault tolerance
t_a	Byzantine faults
t_f	benign faults
$t_h = n - t_a - t_f$	number of active honest nodes
$\mathcal{S}$	the set of MPC nodes
$S_{i \in \mathbb{N}}$	MPC node i
S_A	the set of active MPC nodes
x	secret
$\mathbb{F}_p$	prime field
ϕ	polynomial
$[x]^i$	share of x to S_i
$\mathcal{D}$	developer
σ_R	Ratel program
σ_E	Solidity program
σ_P	Python program
σ_M	MP-SPDZ program
pk_{app}	application contract address
δ	state transitin
st_{pub}, st_{priv}	global public/private states
req	MPC request
id_{mpc}	MPC task ID
id_{pub}, id_{priv}	public/private inputs
C	client
im_{idx}	input mask
sm_{idx}	state mask
σ_i	signature of S_i
κ	fixed-point number precision
seq	MPC task sequence number
$(u, [v]^i)$	key and secret-shared value of a state
B	batch size
T_X	MPC tasks with sequence numbers X
T_x	MPC task with sequence number x
$[x]$	positive integers$\{1, 2, \ldots, x\}$
seq_I	count of initialized MPC tasks
I_j	locally updated MPC tasks for S_j
st_x^y	y-th state updated by MPC task T_x
R	system reliability
r_i	reliability of node S_i
λ_i	failure rate of node S_i
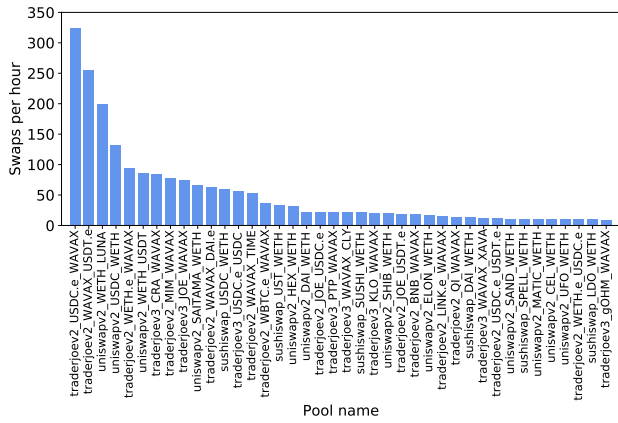
Figure 11: Pool Swap Rates for the Top 40 Trading Pairs in Sushiswap and Uniswap on the Ethereum Network, and Trader Joe on the Avalanche Network, over a period of 5 days post the collapse of LUNA and UST.
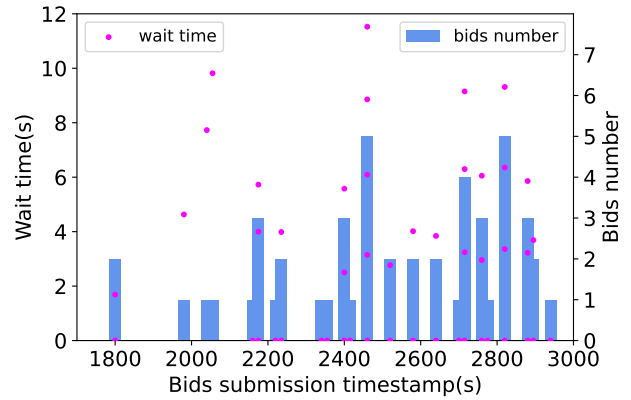

Figure 12: Trader Joe USDC.e-WAVAX Pool Trading History

Figure 13: Queuing time of submitBid in RatelAuction

F Figures

G Table of Related Works

Table 3: Related work that provides privacy to decentralized exchanges (DEX) or underlying blockchain, which can be useful in preventing residual bids and miner extractable value (MEV).

	TEE	ZKP	MPC	HE w/ ZKP
Application Only	Tesseract [20] DerivaDEX [42] SecretSwap [76]	SwapCT [44] ZSwap [43] Manta [31] WKA for Dark Pools [70] FuturesMEX [65] Tex [58]	P2DEX [16] Rialto [48] (+ZKP)	
Framework	Ekiden [28] Obscuro [71]	Hawk [59] ZEXE [23] KACHINA [57] zkay [82]	Absentia [40] White-City [79] Scalable,On-demand MPC [78] GABLE [34] Gage MPC (+ZKP) [5] Eagle (+ZKP) [17]	ZeeStar [81] Pesca [37]