

# MERCURY: Practical Cross-Chain Exchange via Trusted Hardware

Xiaoqing Wen, Quanbi Feng, Jianyu Niu, *Member, IEEE*, Yinqian Zhang, *Member, IEEE*, Chen Feng, *Member, IEEE*

**Abstract**—The proliferation of blockchain-backed cryptocurrencies has sparked the need for cross-chain exchanges of diverse digital assets. Unfortunately, current exchanges suffer from high on-chain verification costs, weak threat models of central trusted parties, or synchronous requirements, making them impractical for currency trading applications. In this paper, we present MERCURY, a practical cryptocurrency exchange that is trust-minimized and efficient without online-client requirements. MERCURY leverages Trusted Execution Environments (TEEs) to shield participants from malicious behaviors, eliminating the reliance on trusted participants and making on-chain verification efficient. Despite the simple idea, building a practical TEE-assisted cross-chain exchange is challenging due to the security and unavailability issues of TEEs. MERCURY tackles the unavailability problem of TEEs by implementing an efficient challenge-response mechanism executed on smart contracts. Furthermore, MERCURY utilizes a lightweight transaction verification mechanism and adopts multiple optimizations to reduce on-chain costs. Comparative evaluations with XClaim, ZK-bridge, and Tesseract demonstrate that MERCURY significantly reduces on-chain costs by approximately 67.87%, 45.01%, and 47.70%, respectively.

**Index Terms**—Blockchain, Cross-chain, Cryptocurrency exchange, Trusted Execution Environment

## I. INTRODUCTION

IN 2008, Satoshi Nakamoto invented Bitcoin [1], the first widely deployed, decentralized cryptocurrency. Subsequently, a myriad of cryptocurrencies backed by blockchain technology (including Ethereum [2], Ripple [3], Algorand [4], etc.) have emerged for diverse applications, reshaping the digital finance landscape [5, 6]. Remarkably, as of May 2024, nearly ten thousand cryptocurrencies exist in the market, collectively valued at approximately 2360 billion USD [7]. This unprecedented diversity has significantly fuelled the interoperability demand between different blockchains, especially secure and efficient cross-chain cryptocurrency trading [8].

Cross-chain cryptocurrency exchange [9] that supports cryptocurrency trading among different blockchains typically operates as follows. Suppose Alice would like to trade 1 Ethereum coin (ETH) [2] for EOS [10] at an exchange rate of 1 ETH = 2500 EOS, as depicted in Fig. 1. This trade involves two on-chain transactions: an ETH deposit from Alice to the exchange and an EOS transfer from the exchange back to Alice. These

Xiaoqing Wen and Chen Feng are with Blockchain@UBC and the School of Engineering, The University of British Columbia (Okanagan Campus), Kelowna, BC, Canada. Email: xqwen@student.ubc.ca and chen.feng@ubc.ca.

Quanbi Feng, Jianyu Niu, and Yinqian Zhang are with the Engineering and Research Institute of Trustworthy Autonomous Systems and the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. Email: 12011501@mail.sustech.edu.cn, niuyj@sustech.edu.cn and yinqianz@acm.org.

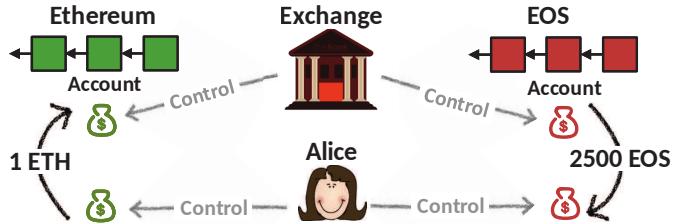


Figure 1: An example of the cross-chain exchange for Alice to exchange 1 ETH for 2500 EOS.

two transactions are inherently coupled in that they either both take place or not at all—an all-or-nothing requirement known as atomicity [11, 12]. This atomicity constitutes a central challenge for the design of cross-chain currency exchange because this property is proved to be impossible without a trusted third party in the asynchronous setting (*i.e.*, no online requirements for the involved parties) [13].

However, current cross-chain solutions suffer from high on-chain verification costs, weak threat models of central trusted parties, or synchronous requirements, making them impractical for currency trading applications. Specifically, notary-based approaches [14, 15] use a single or a small group of notaries (*i.e.*, centralized parties) to handle cross-chain transactions. However, they place high trust in notaries, which makes them suffer from centralization risks [16]. Hashed Time-Lock Contract (HTLC)-based approaches [11, 17] leverage hash and time lock to ensure the atomicity of clients who keep continuous synchronization with blockchains (*i.e.*, synchronization requirement). In addition, the time lock will be designed to be relatively long (*e.g.*, several hours [18]), which gives the initiator the priority to start or cancel the transaction, resulting in griefing attacks [19, 20]. Relay-based approaches [21, 22] rely on smart contracts to do costly on-chain verification of valid or fraud proofs. For instance, the gas consumption of verifying the zero-knowledge proof of Cosmos [23] status on Ethereum is 227K gas [22], which is about 10 times that of a simple ETH transfer on Ethereum. Therefore, we ponder the following question: *how to design a practical exchange that efficiently proceeds cross-chain transactions without centralized parties and synchronization requirements?*

In this paper, we present MERCURY<sup>1</sup>, a practical cryptocurrency exchange that is trust-minimized (*i.e.*, making no trust assumption on exchange operators), and efficient (*i.e.*, low

<sup>1</sup> MERCURY is a major god religion of financial gain, commerce, communication in Roman religion and mythology [24].

gas fees and short confirmation latency), without online-client requirement (*i.e.*, no network-synchronization requirements for clients). To this end, MERCURY leverages a group of operators running consensus inside Trusted Execution Environments (TEEs) to process the various operations of the system, eliminating the reliance on trusted operators and removing synchronization requirements for clients. Specifically, we use Raft protocol inspired by the previous work [25], which proved that running Raft [26] inside TEEs can achieve Byzantine tolerance. Note that operators spend a client’s deposits on one blockchain (*i.e.*, the source blockchain) only after transferring corresponding currencies to the client on another blockchain (*i.e.*, the target blockchain). The integrity of TEE ensures that a client’s deposit will not be spent until the transfer is made, thereby achieving atomicity.

Unfortunately, directly porting cross-chain exchange processes inside the TEE cannot lead to a practical exchange due to the TEEs’ unavailability and resource limitations. First, since the number of operators in the exchange is usually small (*e.g.*, tens of entities), there may not be enough honest operators to ensure the availability of the system. This is because TEEs’ I/O is manipulated by their hosts (*i.e.*, hypervisors or software systems) such that malicious operators can easily block messages from/to TEEs. Without availability, the system cannot ensure the atomicity of cross-chain exchange. For instance, in the above example, after Alice deposits 1 ETH to the exchange, the host can refuse to execute the EOS transfer to Alice, *i.e.*, being unavailable. As a result, Alice neither gets 2500 EOS nor withdraws 1 ETH on the source blockchain. Second, TEEs also face resource limitations, particularly in storage space, presenting a challenge in efficiently verifying the finalization of transactions. Besides, computation and storage on the blockchain are expensive. For instance, storing a 256-bit integer on the Ethereum smart contract costs 8K gas (0.46 USD) [27]. Therefore, minimizing the calculation and storage operations within smart contracts is necessary.

To tackle the first issue, we propose a *cost-efficient challenge-response* mechanism implemented by on-chain smart contracts, which provides the on-chain method for clients to withdraw the deposit. In particular, if the transfer of 2500 EOS fails, Alice can withdraw the 1ETH by initiating a challenge. This mechanism enforces transaction atomicity even when the exchange experiences periods of unavailability. Second, MERCURY proposes a *lightweight verification* mechanism by outsourcing some workload to on-chain smart contracts. Specifically, when registering on the smart contract, operators with TEE synchronize with the blockchain and submit the proof of the latest block to smart contact. Thus, operators only need to synchronize the latest block rather than all historical blocks.

To minimize the on-chain storage cost, we adopt the *periodical checkpoint* for transactions. Specifically, operators submit checkpoints regularly, updating transactions that can be challenged. We also upload transactions and state updates to the blockchain in batch, requiring only a single multi-signature verification on-chain for multiple transactions or state updates. Moreover, we introduce an *pledge incentive mechanism* to prevent malicious clients from orchestrating

Denial-of-Service (DoS) attacks by inundating the system with extensive challenge requests. Note that although MERCURY is built atop blockchains supporting smart contracts, it can be further extended to blockchains without smart contracts by using Hash time-lock contract (HTLC).

To validate our approach, we build a prototype of MERCURY using Golang [28]. We develop the on-chain smart contracts using Solidity 0.8.0 [27] and deploy them on the Ethereum test network, Sepolia [29]. Ethereum’s Virtual Machine (VM) provides Turing completeness [2], fulfilling the requirements for managing the accounts managed by TEE-equipped operators (Sec. IV). We deploy MERCURY on AWS and conduct extensive experiments to evaluate its performance. We also compare MERCURY with three state-of-the-art counterparts, XClaim [21], ZK-Bridge [22] and Tesseract [17].

**Contributions.** The contributions of this work are as follows.

- We propose MERCURY, a high-performance exchange built on TEEs to enable seamless cryptocurrency trading across diverse blockchains. MERCURY is trust-minimized and operates without online requirements for clients with the consideration of security and availability issues of TEEs.
- We introduce an efficient challenge-response mechanism, ensuring clients can withdraw their currencies when the system experiences unavailability. We further optimize the on-chain cost by batching transactions and using checkpoints.
- We discuss the extension of MERCURY to blockchains without smart contracts by replacing the challenge mechanism with Hash time-lock contracts (HTLCs).
- We evaluate MERCURY and compare its performance against three counterparts. Our results show that MERCURY significantly outperforms ZK-Bridge in terms of the transaction proof generation time (3 seconds vs. 18 seconds) and boasts a significantly lower on-chain verification and execution cost (125K gas compared to XClaim’s 389K gas and Tesseract’s 239K gas).

## II. RELATED WORK

We review existing cross-chain methods for realizing cross-chain exchanges that support trading cryptocurrencies against one another. These methods can be divided into three categories: notary-based [14, 15], HTLC-based [11, 17], and relay-based approaches [21, 22]. As mentioned in Sec. I, these methods fall short in weak threat models, synchronization requirements, and high costs. Besides, the reliance on the central party also introduces a single-point-of-failure (SPF), and some methods are tailored for specific blockchains and incompatible with other platforms. A comparison between MERCURY and prior works is provided in Table I.

**Notary-based method.** Notary-based method introduces a single or a small group of notaries to proceed with cryptocurrency transfers and exchanges between different blockchains. When a cross-chain transaction is initiated, it is firstly verified by notaries and then recorded on the involved blockchains. Centralized notary-based exchanges [30, 31] offer advantages in terms of trading fees and efficiency. But, they suffer from significant security risks, including hacking and fraud

**Table I: Comparison of existing cryptocurrency exchange approaches.** The Comp, OCR, FR, and NSPF are short for compatibility, online-client requirement, fairness, and no single-point failure, respectively. The compatibility denotes the ability to support heterogeneous blockchains. Online-client requirement signifies that there is no synchronization requirements for clients, while fairness means that no party can benefit from the fluctuation of the exchange rate.

Protocols	Trust-minimized	CMP	OCR	FR	NSPF
HTLC [11]	✓	✓	✓	✗	✓
Tesseract [17]	✓	✓	✗	✗	✗
XClaim [21]	✓	✗	✗	✓	✓
ZK-Bridge [22]	✓	✗	✗	✓	✓
Bool Network [14]	✗	✓	✓	✗	✓
<b>MERCURY</b>	✓	✓	✗	✓	✓

attacks [32, 33]. Other solutions adopt multiple nodes as a committee for processing cross-chain transactions. Yin *et al.* [14] propose Bool Network, a distributed platform that utilizes Multi-Party Secure Computing to realize distributed key management for hidden committees. Xiong *et al.* [15] implement BFT consensus among committee members. However, they place high trust in notaries, which makes them suffer from centralization risks [16]. Since the number of nodes in these systems is usually small (*i.e.*, tens of nodes [34]), an adversary can easily corrupt all of them.

**HTLC-based method.** HTLC-based method utilizes time locks and hash locks to achieve cross-chain atomic swaps. HTLC [11] requires synchronous clients (*i.e.*, it is interactive, necessitating all parties to be online and actively monitor all relevant blockchains during execution). Tesseract [17] leverages TEE to generate transaction pairs of HTLC, alleviating the requirements for clients to continuously synchronize with the blockchain. However, in Tesseract, a single server carries out transaction processing, which is vulnerable to a single point of failure. Besides, they suffer from griefing attacks [19, 20] caused by the fluctuation of the exchange rate. Specifically, the trader initiating the transaction can watch the exchange rate. If the exchange rate is favorable, the trader will finish the transaction; otherwise, the trader will opt not to disclose the secret, resulting in the abortion of the transaction.

**Relay-based method.** Relay-based method adopts the relay model [35, 36] to deal with cross-chain exchanges, which use on-chain verification to synchronize two chains. XClaim [21] and BTCrely [37] leverages a light client of Bitcoin on Ethereum to verify the transactions on Bitcoin with Simplified Payment Verification (SPV) [38]. ZK-Bridge [22] and Topos [39] utilize zero-knowledge proof technology to enable the on-chain proof for the headers of Ethereum. Zero-Cross [40] further provides privacy protection. CrossChannel [41] establishes a cross-chain channel that employs the chain relay to synchronize channel-related information. However, on-chain verification for zero-knowledge proof or SPV greatly increases the cost and time consumption for cross-chain transactions. Due to the requirements for the verification scheme for a specific blockchain, these methods are usually

**Table II: Summary of notations.**

Term	Description	Term	Description
$\mathcal{S}$	Source blockchain	$p_i$	Operator in MERCURY
$\mathcal{T}$	Target blockchain	$n$	Number of operators
$\text{vault}_{\mathcal{S}}$	Vault on source chain	$\eta_i$	Enclave for operator $p_i$
$\text{vault}_{\mathcal{T}}$	Vault on target chain	$\tau_c$	Timer for a challenge
$tx_{\mathcal{S}}$	Trans on source chain	$\tau_w$	Timer for response
$tx_{\mathcal{T}}$	Trans on target chain	$(pk_i, sk_i)$	Key pair for enclave $i$

tailored for a specific blockchain, making them incompatible with another blockchain. Cosmos [23], Polkadot [42] and Agentchain [43] introduce relay chain, a central hub to transfer and communicate cryptocurrencies between different blockchains.

### III. PROBLEM STATEMENT, MODEL AND CHALLENGES

In this section, we first present the problem statement and the system model. Then, we introduce the associated challenges and necessary preliminaries. Table II lists the frequently used notations in this paper.

#### A. Problem Statement

MERCURY enables a client with cryptocurrency  $s$  on a source blockchain, denoted as  $\mathcal{S}$ , to trade for cryptocurrency  $t$  on a target blockchain, denoted as  $\mathcal{T}$ . Specifically, a group of operators, forming an exchange  $\mathcal{E}$ , receive cryptocurrency  $s$  from the client on the blockchain  $\mathcal{S}$  and transfer cryptocurrency  $t$  to the client on the blockchain  $\mathcal{T}$ . The account of the exchange  $\mathcal{E}$  on the blockchain  $\mathcal{S}$  (resp.,  $\mathcal{T}$ ) is denoted by  $\text{vault}_{\mathcal{S}}$  (resp.,  $\text{vault}_{\mathcal{T}}$ ). To achieve cross-chain exchange, we need two transactions: 1) the client deposits currency  $s$  from its account to  $\text{vault}_{\mathcal{S}}$  on the blockchain  $\mathcal{S}$ ; and 2) the exchange  $\mathcal{E}$  transfers currency  $t$  from its account  $\text{vault}_{\mathcal{T}}$  to the client's account on the blockchain  $\mathcal{T}$ . We use  $tx_{\mathcal{S}}$  and  $tx_{\mathcal{T}}$  to denote the above two transactions, respectively. Here, clients and operators use their public/private key pairs as their identifiers on both the blockchain  $\mathcal{S}$  and  $\mathcal{T}$ , and their transactions can be verified by their signatures. We assume the client can synchronize with the blockchain  $\mathcal{S}$  and  $\mathcal{T}$  to initiate on-chain transactions.

#### B. System Model

We consider an exchange of  $n = 2f + 1$  operators, denoted by the set  $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$ . Each operator  $p_i$  is equipped with a TEE machine. We assume all operators can arbitrarily deviate from the protocol, while the software inside TEEs is off-limit (introduced shortly). The underlying reason for making this harsh assumption is that in a real-world exchange, the number of operators is usually small (*i.e.*, tens of operators [34]), and all of them may be simultaneously corrupted. What is more, we consider the worst case, in which all Byzantine operators can be controlled by a single adversary  $\mathcal{A}$ . There is a public/private key pair of the operator  $p_i$ , denoted by  $(pk_i, sk_i)$ , in which the private key is only stored and used inside TEE.

**Threat model of TEE.** We assume the TEE to protect the program inside TEE in line with other TEE-assisted

designs [17, 25, 44, 45]. We assume that TEE provides integrity and confidentiality guarantees, and secure remote attestation, which can provide unforgeable cryptographic proof that a specific program is running inside TEE [46, 47]. Also, an operator cannot know the private key  $sk_i$  inside TEE. We assume operators have full control over those machines, including root access and control over the network. The operators can arbitrarily launch, suspend, resume, terminate, and crash TEEs at any time. Besides, the TEE operators can delay, replay, drop, and inspect the messages sent to and from TEEs, *i.e.*, manipulating input/output messages of TEEs. In other words, the TEE cannot guarantee availability due to these manipulations, which is the main issue addressed in this work. The rollback attacks are orthogonal to this work and can be addressed by work [48–51].

**Blockchain model.** We establish a cross-chain exchange between two blockchains with different consensus mechanisms and models. As the execution of transactions relies on both chains, if either chain is compromised, the system cannot operate normally. Thus, we follow the assumption in [21] that the blockchain  $\mathcal{S}$  and blockchain  $\mathcal{T}$  are both safe and live. Here, the safety property means that all the honest nodes have the same view of the blockchain, while the liveness property refers to the fact that some event must eventually happen in a distributed system [52].

We assume that both blockchains provide finality, *i.e.*, once transactions or blocks are finalized, they will remain finalized forever [53]. We assume that both blockchains support smart contracts [54]. Smart contracts can access the current time using the method  $block.timestamp$  and the hash of the recent 265 blocks [27] using the method  $block.bh(i)$ , where  $i$  is the number of the block. Smart contracts can also support cryptographic schemes, including hash computing and ECDSA encryption algorithm.

### C. System Goals

we derive the following desirable properties for MERCURY under the above threat and blockchain models.

- **Atomicity.** An atomic cross-chain settlement can guarantee that both  $tx_S$  and  $tx_T$  will be confirmed on their chains, or neither  $tx_S$  nor  $tx_T$  will be confirmed on their chains.
- **Trust-Minimized Participants.** No trusted third parties are involved to ensure the atomicity of cross-chain exchange.
- **No on-lineclient requirement.** If the clients do not synchronize with the blockchain after initializing the cross-chain requests, they will not suffer any loss (*i.e.*, the atomicity can be guaranteed).

### D. Challenges

There are several challenges in designing MERCURY, which are discussed below.

**Atomicity protection against malicious operators.** The unavailability of TEEs poses a significant threat to the overall atomicity of cross-chain exchanges. For example, when all TEEs are unavailable, a client, after depositing its assets on the blockchain  $\mathcal{S}$ , will never receive the corresponding tokens

from the blockchain  $\mathcal{T}$ . To address this critical issue, we propose a practical challenge-response mechanism implemented through on-chain smart contracts. If the client's deposit is not processed for a long time, the client can raise a challenge on the smart contract to force the operator to execute the transaction and reply to the challenge. Otherwise, the deposit will be returned to the client.

**Transaction verification under limited resources in TEEs.** TEEs face resource limitations, particularly in storage space, presenting a challenge in efficiently verifying the finalization of a given transaction. Inspired by the light-client design, which stores only block headers to verify transactions, MERCURY proposes a similar solution that relies only on block headers. Unlike traditional light-client design, MERCURY doesn't require all the historical block headers and it outsources some workload to on-chain smart contracts. Specifically, in the initiation phase, TEEs engage in a process of getting the latest finalized block headers and committees. Notably, successful registration on the chain is exclusively granted to TEEs that proficiently synchronize historical block headers. Therefore, adopting simple on-chain verification can reduce the synchronization cost for historical blocks in TEE.

**Minimizing expensive on-chain costs.** In MERCURY, an on-chain vault is instrumental for interactions with operators and users, which introduces computational and storage costs that directly impact the expenses of cross-chain exchanges. Recognizing the necessity to minimize these costs, our design focuses on the efficient use of on-chain resources. To reduce the computational cost of on-chain verification for TEEs, the TEE uploads a transaction bundle to the chain, allowing for the verification of multiple transactions with a single TEE signature. To minimize on-chain storage costs, we establish checkpoints for challengeable transactions, enabling the removal of transactions preceding the checkpoint.

### E. Building Blocks

**Trusted Execution Environment.** An operator can instruct his TEE to create new enclaves, *i.e.*, new execution environments running a specified program. To develop TEE-agnostic protocols, we adopt the methodology outlined in [55], which models the functionality of attestable TEEs. Each TEE is initialized with a key pair  $(pk, sk) \leftarrow \Sigma.keyGen(1^\lambda)$  generated by its manufacturer, where  $\Sigma$  is the signature algorithm. Here,  $sk$  represents the TEE's internal master secret key, while  $pk$  is the corresponding public key. The ideal functionality of a TEE offers the following Application Interfaces (APIs) for a program code  $prog$ :

- $eid \leftarrow install(prog)$ : Install the program  $prog$  as a new enclave within the TEE, assigning it a unique ID,  $eid$ .
- $(outp, \sigma) \leftarrow resume(eid, inp)$ : Resume the enclave with ID  $eid$  to execute program  $prog$  using the input  $inp$ .  $\sigma$  is the TEE's endorsement, confirming that  $outp$  is the valid output.

Besides, we include an attestation API for TEE to generate an attestation quote  $\rho \leftarrow attest(eid, prog)$  proving that the program  $prog$  has been installed in the enclave  $eid$ . And  $\rho$  can be verified through method  $verifyquote(\rho)$  [56].

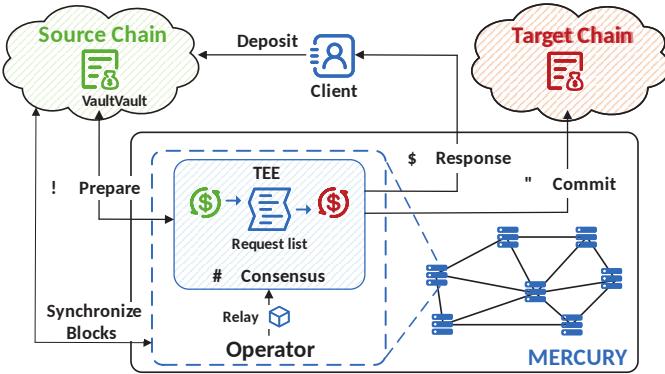


Figure 2: An architectural overview of MERCURY. A client transfers currencies from the chain  $\mathcal{S}$  to the chain  $\mathcal{T}$ .

**Cryptographic Primitives.** Our protocol utilizes a multi-signature scheme (GENSIG, SIGN, VERIFY), and a secure hash function  $\text{HASH}(\cdot)$ . All message sent within our protocol are signed by the sending party. A message signed by party  $p$  is denoted as  $\text{SIGN}(mr; sk_p) \rightarrow \sigma$ . Multi-signature scheme allows multiple parties to collaboratively generate a digital signature for a message by two main functions: SIGN and VERIFY. Specifically, given a set of users  $G$  and the message  $m$ , the function  $\text{SIGN}(mr, \{sk\}_{g \in G}) \rightarrow \sigma$  takes the private key  $sk$  and the hash of the message  $mr$  as input and return the signatures  $\sigma$ . The function  $\text{VERIFY}(\{pk\}_{g \in G}, mr, \sigma) \rightarrow 0/1$  takes the public key  $pk$ ,  $mr$  and  $\sigma$  and returns the result of whether the signature was generated by  $G$ .

**Automated Market Maker (AMM).** AMM [57] algorithm automatically provides liquidity to a decentralized exchange by continuously adjusting the exchange rate of the traded currencies based on some predefined mathematical formulas. Specifically, Constant Function Market Maker (CFMM) [58] can keep the product of the quantities of currencies as constant. For instance, in a system which holds X (resp., Y) amount of currency A (resp., B), if a client wishes to exchange  $x$  amount of currency A for currency B, the quantity of currency B available for exchange satisfies  $(X+x)*(Y-y) = X*Y = k$ , where  $k$  is a constant. The function  $y \leftarrow \text{AMM.exchange}(x)$  calculates the exchange rate from A to B. The currencies in the liquidity pool are supplied by Liquidity Providers (LPs) [59]. LPs contribute to the liquidity pool by depositing both currencies A and B.

#### IV. MERCURY DESIGN

##### A. Overview

Fig. 2 illustrates an architectural overview of MERCURY, which enables secure currency exchanges between any two chains: the blockchain  $\mathcal{S}$  and the blockchain  $\mathcal{T}$ . The vaults act as the on-chain interactions for clients, while operators are the entities that handle cross-chain exchanges.

MERCURY includes five key components: system initialization (Sec. IV-B), normal-case operation of the cross-chain exchange (Sec. IV-C), challenge-response mechanism (Sec. IV-D), transaction verification inside TEE (Sec. IV-E),

##### Algorithm 1 The Partial Functions Executed by vault

```

1: Function INIT(committee)           ▷ Initialize Requests
2:    $Dep \leftarrow \emptyset$              ▷ Set deposits list to empty
3:
4: Function DEPOSIT( $s_{addr}, v$ ) 
5:   Require  $v > 0$ 
6:    $id \leftarrow \text{HASH}(s_{addr}, v, \text{block.timestamp})$ 
7:    $Dep_{id}.sender \leftarrow s_{addr}$ 
8:    $Dep_{id}.value \leftarrow v$ 
9:    $Dep_{id}.underChal \leftarrow \text{false}$ 
10:  Trigger Deposit event
11:
12: Function TRANSFER( $tranSet, signs$ )
13:   Require  $\text{VERIFYMUTISIG}(signs)$ 
14:   For each  $tx \in tranSet$ 
15:      $tx.receiver.transfer(v)$ 
16:   End For
17:
18: Function STARTCHALLENGE( $id, value, pledge$ )
19:   Require  $pledge \geq Pledge_C$ 
         $\wedge value = Dep_{id}.value$ 
         $\wedge Dep_{id}.underChal = \text{false}$ 
20:    $Dep_{id}.underChal \leftarrow \text{true}$ 
21:    $Dep_{id}.startChal \leftarrow \text{block.timestamp}$ 
22:   Trigger Challenge event
23:
24: Function RESLOVECHALLENGE( $id$ )
25:   Require  $Dep_{id}.underChal = \text{true}$ 
         $\wedge \text{block.time} - Dep_{id}.startChal > \tau_w$ 
26:   REFUND( $id$ )
27:   DELETE( $Dep_{id}$ )
28:
29: Function UPDATECHECKPOINT( $idSet, signs$ )
30:   Require  $\text{VERIFYMUTISIG}(signs)$ 
31:   For each  $id \in idSet$ 
32:     DELETE( $Dep_{id}$ )
33:   End For
```

and rewards (Sec. IV-F). Specifically, system initialization involves the registration of operators on the respective blockchains; normal-case operations are responsible for transaction processing; challenge-response mechanism ensures clients redeem their deposits on the chain; transaction verification allows enclaves to synchronize with blockchain  $\mathcal{S}$  and  $\mathcal{T}$ ; and incentive mechanisms to incentivize liquidity provision for the vaults and to sustain services, equitably distributing transaction fees among LPs and operators.

##### B. System Initialization

During the initialization, *vaults* are deployed on different blockchains by the service providers. Algorithm 1 gives the main functions executed in *vault*. Then, the operators join the system by registering on *vaults*. Thus, *vaults* are jointly controlled by the operators, whose public keys are recorded on the *vaults*. The registration process involves the attestation and block synchronization verification for the enclave, and recording the enclave's public key on the *vaults*.

There are  $n = 2f + 1$  operators that register in the  $vaults$  during the initialization. Here, we present an exemplary representation of an operator  $p_i$  registering on  $vault_S$  for clarity. The operator  $p_i$  equipped with TEE instructs his TEE to install the program  $prog$  for MERCURY and creates a new enclave  $\eta_i$ . The registration process has three steps:

- ❶ The enclave  $\eta_i$  in operator  $p_i$  generates a unique pair of private and public keys  $\langle sk_i, pk_i \rangle$  for the  $vault_S$  on the blockchain  $S$ . The secret key  $sk_i$  is securely stored within  $\eta_i$ , accessible only by the program running in  $\eta_i$ , while the public key  $pk_i$  is returned as output to the operator  $p_i$ . Subsequently,  $\eta_i$  generates an attestation  $\rho_i$  claiming that the program  $prog$  runs in  $\eta_i$  and controls the secret key  $sk_i$ .
- ❷ The operator  $p_i$  transmits the header of the latest  $k$  finalized block to the enclave  $\eta_i$ , where  $k$  is set as the upper bound the enclave may lag.  $\eta_i$  verifies the block header's consistency (refer to Sec. IV-E), generating a proof  $\rho_i^{bc}$  (*i.e.*, the header of the latest block signed by the  $\eta_i$ ).
- ❸ Then, the operator registers  $\eta_i$  by invoking  $REGISTER(\eta_i, \rho_i, \rho_i^{bc}, pk_i)$  on the  $vault_S$ . The  $vault_S$  verifies that  $\rho_i$  is a valid attestation [55, 60] and  $\rho_i^{bc}$  refers to one of the latest  $k$  blocks on the blockchain  $S$ . Upon successful verification,  $vault_S$  adds the operator  $p_i$  with  $pk_i$  to the operators list.

The registration process guarantees the execution of the  $prog$  for all registered enclaves and the confidentiality of the secret key  $sk_i$ . Consequently, there is no need to re-attest enclaves in subsequent protocol steps. Clients can access the  $vault_S$  by its smart contract address on the blockchain  $S$ .

### C. Cross-Chain Exchange

To safeguard the liveness of MERCURY, no more than  $f$  TEEs operated by a group of  $n$  operators may crash. To ensure that operators can generate enough votes and that merely  $f$  crash operators cannot collect valid numbers of votes, we design the system such that  $n = 2f + 1$ , and a minimum of  $f+1$  votes is required for passing the verification on smart contract. As mentioned in Sec. I, we implement RAFT protocol to the group of TEE-equipped operators. In the normal-case operation of a cross-chain exchange, where a client seeks to trade its currencies between the blockchain  $S$  and the blockchain  $T$ , the process encompasses five phases as shown in Fig. 2 and explained below.

- ❶ The client initiates an exchange request by depositing its currency  $s$  to  $vault_S$  on blockchain  $S$ . Specifically, it creates a  $deposit$  transaction as a tuple of  $\langle s_{addr}, v \rangle_\sigma$  (cf. Algorithm 1 line 4), where  $s_{addr}$  is the client's address on the blockchain  $S$ , and  $vault$  is the transferred value of currency  $s$  (greater than zero).  $vault_S$  calculates a unique  $id$ , which is the hash value of the  $deposit$  transaction, records  $\langle id, s_{addr}, v \rangle$ , and locks the  $vault$  amount of currency  $s$ . Note that each  $deposit$  corresponds to a unique on-chain identifier  $id$ , which prevents replay attacks (*i.e.*, clients cannot initiate more than one  $request$  using the same  $deposit$ ).

Then, the client sends a  $request$  as a tuple  $\langle REQUEST, id, s_{addr}, id_T, r_{addr} \rangle_\sigma$  to all the operators, where  $id_T$  is the identifier of the blockchain  $T$ , and  $r_{addr}$  is

the receiving address of the client on  $T$ . Note that the message  $request$  contains the information about the blockchain  $T$ , which is not included in the  $deposit$  transaction to reduce on-chain transaction size and associated fees.

- ❷ Upon receiving the client's  $request$ , the enclave  $\eta_i$  verifies whether the  $deposit$  transaction is finalized on the blockchain  $S$  by synchronizing with  $S$  (details in Sec. IV-E). Upon successful verification, operators check whether  $s_{addr}$  and  $vault$  of the  $deposit$  match the information in  $request$ . Then, they compute the amount of currency  $t$  by  $AMM.exchange(v)$ , where  $vault$  (resp.,  $vault_T$ ) is the amount of currency  $t$  (resp.,  $s$ ). Then, the enclave  $\eta_i$  generates a  $transfer$  transaction for the blockchain  $T$ , denoted as  $\langle id_T, v_T, r_{addr} \rangle$ , adds it to the memory pool, and awaits consensus.
- ❸ The enclaves in operators run a RAFT protocol to achieve agreement on the transaction  $\langle TRANSFER, id, s_{addr}, v, id_T, r_{addr} \rangle$ . Note that transactions, aiming at the same blockchain  $T$ , are batched and signed by operators and subsequently submitted to the  $vault_T$ . Thus,  $vault_T$  can process several transactions through one-time signature verification, optimizing on-chain computing cost. During the consensus,  $\eta_i$  signs batches with its private key for the blockchain  $T$ , and broadcasts the signature  $sig_i$  to other TEEs. This private key is generated during the initialization and the public key is recorded in the  $vault_T$ .
- ❹ Upon collecting more than  $f$  signatures,  $\eta_i$  generates a  $m\text{-}n$  multi-signature  $\sigma$  for the batch of transactions, where  $m = f + 1$ . Then,  $\eta_i$  submits these transactions together with the multi-signature  $\langle txs \rangle_\sigma$  to the blockchain  $T$ . Miners on the blockchain  $T$  verify the multi-signature and execute the transactions. For each transaction  $transfer$ , the designated receiver account, as specified by the client, receives the transfer from  $vault_T$ .
- ❺ After  $transfer$  is finalized on the blockchain  $T$ , operators return the execution result to the respective clients associated with these transactions.

### D. Safe Redemption

In this section, we propose a practical challenge-response mechanism deployed in  $vault_S$  to address the safe redemption issue. As mentioned in Sec. III, TEEs controlled by Byzantine operators cannot ensure availability. Thus, there exists a potential scenario wherein transaction  $tx_S$  on the blockchain  $S$  is finalized, while transaction  $tx_T$  on the blockchain  $T$  remains unsubmitted due to the unavailability of the TEE, which breaks the atomicity of cross-chain exchange. The challenge-response mechanism provides an on-chain method in this scenario, which allows clients to withdraw their deposit by initiating a challenge on  $vault_S$ .

To implement the challenge-response mechanism, we made some adjustments to the normal case operation. Upon the client's submission of a  $deposit$ , the currency is not directly transferred to the  $vault_T$ ; instead, it remains pending. Thus, the client can withdraw the  $deposit$  if the transaction on the blockchain  $T$  fails. Only after the operators process the transaction and receive the confirmation on the blockchain  $T$  can the  $deposit$  be locked (*i.e.*, can not be withdrawn).

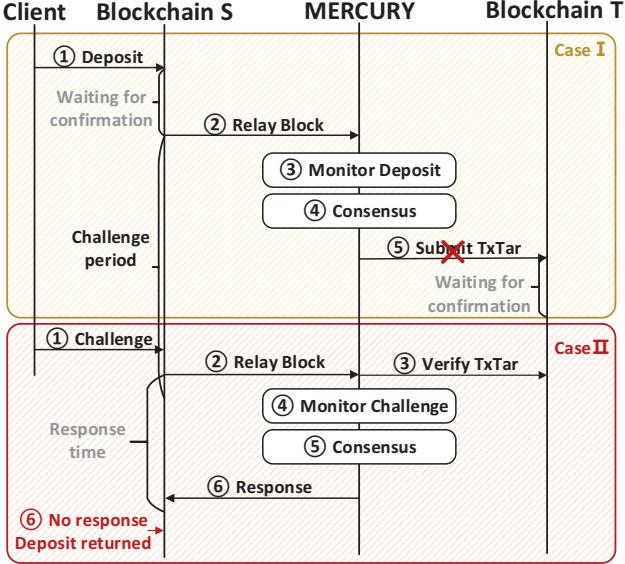


Figure 3: The design of the challenge-response protocol. Case I represents the normal-case operation, whereas case II represents the case of failures. The red cross represents the process that cannot be completed when operators fail.

Fig. 3 illustrates the challenge process. After creating the *deposit* transaction, a client sets a timer of  $\tau_c$ . We can set  $\tau_c$  to  $2\delta_S + 2\delta_E + \delta_T$ , where  $\delta_S$  (resp.,  $\delta_T$ ) is the transaction confirmation time on the blockchain  $\mathcal{S}$  (resp., blockchain  $\mathcal{T}$ ), and  $\delta_E$  is the time for the operators to process the exchange request. If  $transfer_T$  is finalized on the blockchain  $\mathcal{T}$ , operators will send  $\langle \text{CONFIRM}, id \rangle_\sigma$  to  $vault_S$ . Then,  $vault_S$  set the state of  $tx_S$  to be confirmed and can no longer be withdrawn. Else if no  $tx_T$  on the blockchain  $\mathcal{T}$  is finalized before the timer is triggered, the client invokes `STARTCHALLENGE()` (cf. Algorithm 1 line 18) on  $vault_S$ .

Once the client initiates a challenge on  $vault_S$ , a timer  $\tau_w$  waiting for a response is set up. To give the operators enough response time,  $\tau_w \gg 2\delta_S + \delta_E$ . Note that  $\tau_w$ , as the waiting time for clients to refund in case of failure, will not affect the transaction latency under normal cases. Thus, in practice,  $\tau_w$  is set to be sufficiently long, providing the operators with ample time to resubmit transactions and respond. When a challenge against  $deposit_i$  is triggered in  $vault_S$ , the response falls into the following two cases.

- **Operators are available.** When the operators witness the challenge from the blockchain  $\mathcal{S}$ , they verify whether  $transfer_i$  (*i.e.*, the corresponding transaction for the  $deposit_i$  in the blockchain  $\mathcal{S}$ ) on the blockchain  $\mathcal{T}$  has been finalized. If  $transfer_i$  has not been carried out, the operators can resubmit it on the blockchain  $\mathcal{T}$ . Once the transaction is finalized, the operators return the result to  $vault_S$ , thereby successfully concluding the challenge. The resolution of challenges similarly requires consensus, with operators reaching consensus on  $\langle \text{CHALLENGE}, id \rangle$  and collecting more than  $f$  signatures. Then, the operators hand up the  $deposit_i$  and the multi-signature of operators.  $vault_S$  verifies the multi-signature and deletes  $deposit_i$  if validated.

- **Operators are unavailable.** The operators are unable to respond to the challenge on the blockchain  $\mathcal{S}$ . Once  $T_w$  has elapsed, the client can claim a refund on  $vault_S$  by invoking `RESOLVECHALLENGE(id)` (cf. Algorithm 1 line 24). Then, the  $vault_S$  refunds the  $deposit_i$  to the client if the waiting time  $\tau_w$  has elapsed.

To counteract potential malicious challenges, the client is required to provide a pledge while starting the challenge. If the challenge is successful, the client receives a refund along with the pledge; otherwise, the pledge is forfeited. Besides, to reduce on-chain storage, our approach retains only outstanding *deposits* on  $vault_S$ , ensuring that completed ones are removed. After  $transfer_i$  has been finalized on the blockchain  $\mathcal{T}$ , operators delete  $deposit_i$  from  $Dep$  on the blockchain  $\mathcal{S}$ , and the associated *request* cannot be challenged anymore. However, this approach increases the interactions between operators and the blockchain  $\mathcal{S}$ . Thus, we set up checkpoints to batch-confirmed transactions. Operators upload checkpoints after acquiring a fixed number of completed cross-chain exchanges or after a designated period. Specifically, operators upload a checkpoint through `UPDATECHECKPOINT(idSet, signs)` (cf. Algorithm 1 line 29), where  $idSet$  is the list of cross-chain deposits to be confirmed and  $signs$  is the  $f+1$  multi-signature of operators.

#### E. Transaction Verification inside TEE

To verify transactions on the blockchain, operators need to verify transactions on the blockchain  $\mathcal{S}$  and blockchain  $\mathcal{T}$  inside TEE. However, given the potential for malicious operators, there exists a risk for enclaves to receive forged blocks. Besides, the current storage requirement for a full node of Ethereum exceeds 600GB [61], which is impractical for enclaves. Inspired by the light-client design [62], we design a verification strategy in enclave  $\eta_i$  that checks the validity of received blocks without withholding the whole blockchain. Ethereum simplifies the verification for the light clients by introducing a sync committee consisting of 512 randomly selected validators. The members of the sync committee are expected to sign every block header at each slot (*i.e.*, the time interval for block proposal in Ethereum), among which there is a different node assigned to propose a new block. Besides, the current and the next sync committee can also be observed within the finalized blocks. Thus, we adapt the design of the sync committee.

During the initialization phase,  $\eta_i$  first synchronizes the existing  $k$  finalized blockchain headers BHF and related block data.  $b_h$  is the latest finalized block with height  $h$ .  $\eta_i$  verifies that (i) BHF itself is consistent, (ii) the latest finalized block ( $b_h$ ) has enough signatures of the sync committee. Then,  $\eta_i$  generates a proof  $\rho_i^{bc} \leftarrow \langle b_h.bh \rangle_\sigma$ , which is the header of  $b_h$  signed by enclaves, and hand out to the vault. Vault verifies  $\rho_i^{bc}$  and decides whether enclave  $\eta_i$  has gotten the latest  $k$  finalized block  $b_h$  with blockchain, where  $k$  is set as the upper bound to the time an enclave may lag behind. If passed, the enclave  $\eta_i$  is considered to be initially synchronized successfully. Then, enclave  $\eta_i$  can discard block data and retain current and next sync committee members, along with BHF storing finalized block headers.

After the initialization,  $\eta_i$  continuously receives finalized blocks. Once a block is finalized [63] and verified by  $\eta_i$ , its header, along with all its ancestors, is placed in BHF. To verify that a block  $B$  is finalized,  $\eta_i$  needs to verify the validity of the sync committee and their signatures. By comparing the sync committee with the expected one conveyed in the historical finalized block,  $\eta_i$  can verify the validity of the committee as well as these signatures. Therefore, enclave  $\eta_i$  only needs to store historical block headers since initialization.

#### F. Vault Liquidity and Rewards

As mentioned in Sec. IV-B, we deploy *vaults* jointly controlled by the operators, allowing any parties to provide *vaults* with liquidity. To motivate operators to diligently handle transactions and LPs to provide liquidity, we distribute transaction fees among LPs and operators. The idea behind this design is that incentive mechanisms such as transaction fees and block rewards play an important role in securing blockchain systems [64–66].

Suppose the transaction fee for one transaction is  $F$ . The LP provides an amount  $L$  of liquidity to *vault*, the total liquidity in *vault* is  $T$ . The reward for the LP is then proportional to the fraction of total liquidity it provided, expressed as:  $F_L = (L/T)Fr$ , where  $r$  is a pre-determined parameter representing the proportion of rewards in transaction fees for LPs. For each completed transaction, operators that participate in the final  $m - n$  votes of the consensus process receive rewards. For a transaction with  $m$  signatures, each TEE participating in its consensus can obtain  $F_T = (1/m)F(1 - r)$ .

Furthermore, to incentivize continued service provision, each operator is required to deposit upon registration. If MERCURY is deactivated, settlement is executed on the blockchain, and any losses incurred by MERCURY due to TEE unavailability will be borne by the TEE providers.

## V. SECURITY ANALYSIS

In this section, we conduct a security analysis of MERCURY. As described in Sec. III, MERCURY is *trust-minimized* since there is no trust assumption of operators in the exchange. Then, we give a proof of Atomicity and No online-client requirement. Recall that  $tx_S$  is a *deposit* transaction on the blockchain  $\mathcal{S}$  and  $tx_T$  is a *transfer* transaction. We establish several technical lemmas.

**Lemma 1.** *If  $tx_S$  is confirmed on the blockchain  $\mathcal{S}$ ,  $tx_T$  must be confirmed on the blockchain  $\mathcal{T}$ .*

*Proof.* Proof by contradiction. Suppose that  $tx_T$  is not confirmed on the blockchain  $\mathcal{T}$ . Thus, the request that confirm the associated  $tx_S$  in  $\text{vault}_S$  is invalid. As mentioned in Sec. III, programs inside TEE are protected and can not be compromised, *i.e.*, this invalid request can not be signed by the secret key inside the operators' TEE. Then, the invalid request without  $f + 1$  signatures from operators' TEE will not be accepted by  $\text{vault}_S$  and no one can confirm  $tx_S$  in  $\text{vault}_S$ . Therefore, after a period of  $\tau_c$ , the client can issue a challenge and cancel the  $tx_S$ . This contradicts the fact that  $tx_S$  is confirmed on the blockchain  $\mathcal{S}$ .  $\square$

**Lemma 2.** *If  $tx_T$  is confirmed on the blockchain  $\mathcal{T}$ ,  $tx_S$  must be confirmed on the blockchain  $\mathcal{S}$ .*

*Proof.* Proof by contradiction. Suppose that  $tx_S$  is not confirmed on the blockchain  $\mathcal{S}$ . Then, we have two cases as follows.

- **Case 1.**  $tx_S$  is not even submitted to the blockchain  $\mathcal{S}$ . Thus, the associated transaction  $tx_T$  in  $\text{vault}_T$  is invalid. In this case,  $tx_T$  can not be signed by  $f + 1$  operator's TEE and can not be confirmed on the blockchain  $\mathcal{T}$ .
- **Case 2.**  $tx_S$  is pending on the blockchain  $\mathcal{S}$  (but not confirmed). According to Sec. IV-D, operators will generate the associated  $tx_T$  and submit it to the  $\text{vault}_T$ . If transaction  $tx_T$  is not confirmed on blockchain  $\mathcal{T}$ , the assertion is proven. Conversely, if  $tx_T$  is confirmed on blockchain  $\mathcal{T}$ , operators will observe  $tx_T$  and subsequently confirm the corresponding transaction  $tx_S$  on  $\text{vault}_S$ . Therefore,  $tx_S$  will be confirmed, a contradiction.

$\square$

**Theorem 1** (Atomicity). *Both  $tx_S$  and  $tx_T$  are either confirmed or not confirmed at all.*

*Proof.* It follows immediately from Lemma 1 and Lemma 2.  $\square$

**Theorem 2** (No online-client Requirement). *If the client is offline at any finite time  $\tau$  after initiating a cross-chain transaction, the atomicity can ultimately be guaranteed.*

*Proof.* The client initiates the cross-chain transaction by submitting  $tx_S$  to the  $\text{vault}_S$ . If the  $tx_S$  fails, cross-chain transactions will not start at all, and both  $tx_S$  and  $tx_T$  are not confirmed. If the  $tx_S$  is executed on the blockchain  $\mathcal{S}$ , the  $tx_S$  remains pending on the  $\text{vault}_S$ . Then the execution of  $tx_T$  falls into the following two cases. In the first case,  $tx_T$  is confirmed on the blockchain  $\mathcal{T}$ ,  $tx_S$  will also be confirmed by the operators, whether the client is online or not. At this time, both  $tx_S$  and  $tx_T$  are confirmed. In the second case,  $tx_T$  is failed. By Lemma 1, the  $tx_S$  in  $\text{vault}_S$  will not be confirmed, and the client can challenge it at any time. Thus, as the finite offline time  $\tau$  pass, the client can initiate the challenge and cancel the  $tx_S$ , *i.e.*, both  $tx_S$  and  $tx_T$  are not confirmed.  $\square$

## VI. EVALUATION

In this section, we evaluate MERCURY in terms of on-chain verification costs and off-chain transaction throughput and latency. We compare MERCURY with three state-of-the-art counterparts, XClaim [21], ZK-Bridge [22] and Tesseract [17]. This section answers the following three questions:

- **Q1:** What is the breakdown of on-chain costs in MERCURY?
- **Q2:** How does MERCURY perform in terms of throughput and latency?
- **Q3:** How does MERCURY compare with counterparts?

### A. System Implementation and Setup

We build a prototype of MERCURY to evaluate its performance, employing the Ethereum test network Sepolia [29],

Table III: System Parameters.

Parameters	Values
Number of operators	5, <b>10</b> , 15, 20
Batch size for RAFT consensus	500, 1000, 1500, <b>2000</b>
Number of transactions submitted on-chain	1, 10, <b>20</b> , 50, 100

which is consistent with the version of the Ethereum main network. We use Solidity 0.8.0 [27] to develop the smart contract (*i.e.*, *vaults*). We deploy MERCURY on AWS EC2 machines with one m6a.xlarge instance per operator. All instances run with the Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz and 16G RAM running Ubuntu Linux 22.04, with AMD SEV as the TEE component [67]. The consensus protocol among operators relies on the implementation of RAFT [26]. We conduct experiments in a LAN environment with  $0.5 \pm 0.03$  ms and a WAN environment with  $20 \pm 0.1$  ms, each machine was equipped with a private network interface with 100MB bandwidth.

We vary parameters in Table III with default values in bold to evaluate system gas cost, throughput, and latency. The experiment focuses on the cost on the chain and the throughput and delay of off-chain transaction processing. Precisely, cost measures the gas cost for executing functions of the smart contract (*i.e.*, *vault*), some of which can include multiple transactions simultaneously, thus reducing the shared cost for each transaction. Throughput measures the number of transactions that operators can handle per second. Meanwhile, latency measures the time cost for operators to vote and process transactions.

#### B. On-chain Cost of Functions

Cross-chain exchange costs on the blockchain are measured in “gas”, which serves as a unit of measurement that gauges the computational effort needed to perform operations on Ethereum. To make these costs more understandable, we convert gas into USD with a gas price of 20 Gwei ( $2 \times 10^{-8}$  ETH) and an ETH price of 2850 USD on 1 May 2024.

**Cost breakdown.** Table IV presents the on-chain costs of MERCURY for each function involving 10 operators. Our measurements cover the entire process, including DEPOSIT, TRANSFER (see IV-C), and CHALLENGE (see IV-D) on Ethereum. The TRANSFER function is particularly noteworthy as one of the most gas-consuming, approximately three times that of the Deposit method (84K gas). This higher cost is attributed to the verification of the multi-signature of TEEs, which demands a substantial amount of computation. However, with a batch size of 20 transactions, the per-transaction fee is amortizing to 14K gas (0.82 USD), less than a simple ETH transfer<sup>2</sup>. A successful challenge-response involves both the STARTCHALLENGE and RESOLVECHALLENGE functions, totaling 85K gas (4.17 USD). The UPDATEREPOINT function consumes 260K gas. However, as we implement a checkpoint update every 20 transactions, the cost per transaction for UPDATEREPOINT is 1.48 USD. Therefore,

<sup>2</sup>Sample ETH transfer is an example of sending ETH in Ethereum, commonly employed for comparing gas fees [68].

Table IV: **On-chain cost.** The costs are in USD as per exchange rates of 1 May 2024: ETH/USD 2850 and gas price is 20 Gwei ( $2 \times 10^{-8}$  ETH).

Functions	Total (gas)	Average (gas/tx)	USD (\$)
DEPOSIT	84,404	84404	4.81
TRANSFER	289,018	14451	0.82
STARTCHALLENGE	47,351	47351	2.70
RESOLVECHALLENGE	37,364	37364	2.13
UPDATEREPOINT	259,814	25981	1.48
Simple ETH transfer	21,000	21000	1.20

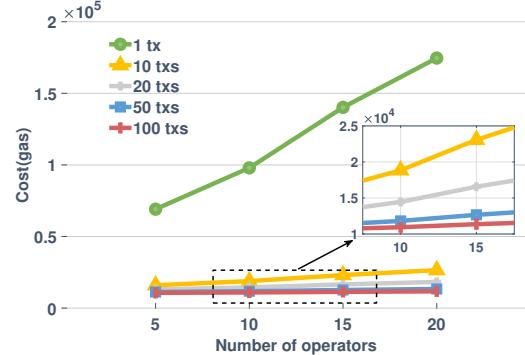


Figure 4: On-chain cost of the TRANSFER function with varying numbers of operators and transactions.

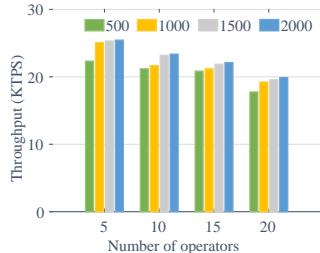
the total cost of a cross-chain transaction, including deposit, transfer and update checkpoint, is 125K gas per transaction.

**Cost of the TRANSFER function.** As mentioned in Sec. IV, TRANSFER function submits more than one transaction at a time. Thus, we provide detailed insights into the average gas requirements for the TRANSFER function, which is the most gas-consuming function in MERCURY. Fig. 4 illustrates transfer costs in MERCURY across varying numbers of operators and batch sizes. The observed trend indicates that the cost of TRANSFER grows with an increasing number of operators, reaching 175K gas when there are 20 operators without batching (*i.e.*, 1 transaction per batch). This substantial cost underscores the necessity of implementing batching to reduce the per-transaction expenses. As the batch size increases, there is a significant reduction in average costs. As the batch size further increases to 100 transactions, the cost diminishes to just 12K gas (0.67 USD).

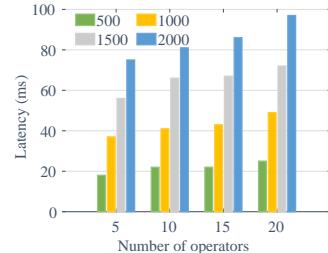
#### C. Performance Evaluation

We evaluate two performance metrics: 1) throughput, measured in thousands of transactions per second (KTPS), representing the number of transactions executed per second, and 2) latency, measured in millisecond (ms), denoting the average end-to-end delay from the moment operators get the transactions until the submission of the transaction.

**Throughput and latency.** Fig. 5 illustrates the throughput and latency in MERCURY across varying numbers of operators and batch sizes in LAN and WAN environments. In the LAN environment, throughput shows a modest decrease while latency increases as the number of operators rises, peaking at



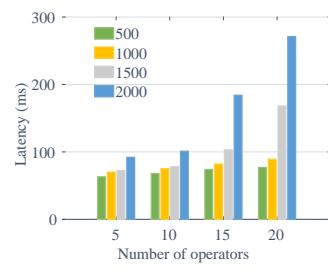
(a) Throughput in LAN



(b) Latency in LAN



(c) Throughput in WAN



(d) Latency in WAN

Figure 5: Transaction processing performance with varying different numbers of operators and batch size in LAN and WAN.

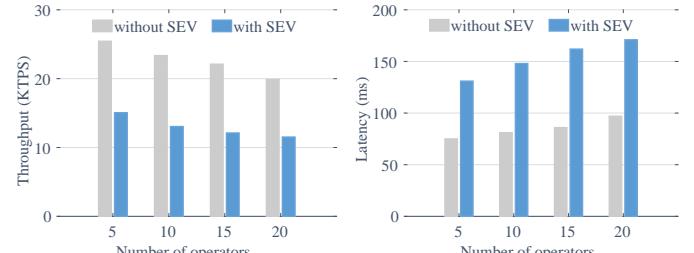
38 ms with a batch size of 1000 and 20 operators. This occurs because the increase in the number of operators leads to a rise in messaging overhead, causing bandwidth inefficiencies. Throughput exhibits an upward trend with increasing batch size, while latency decreases, reaching 10 ms at a batch size of 1000 with 5 operators. This is because, with larger batch sizes, MERCURY can process more transactions concurrently.

In the WAN environment, throughput shows a modest decrease while latency increases as the number of operators rises. This trend is more pronounced than in the LAN environment because the increased communication delay in WAN amplifies the impact of adding more operators on system performance. Additionally, both throughput and latency increase with larger transaction batch sizes.

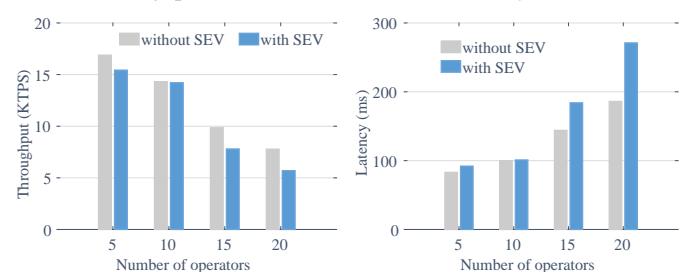
**TEE overhead.** We evaluate the throughput and latency of MERCURY with and without SEV across 5-20 operators in both LAN and WAN environments. In the LAN environment, the introduction of SEV results in a halving of throughput and nearly doubles the latency. This degradation occurs because operations executed inside a TEE require encryption and context switching from the host machine’s regular execution environment, which hampers performance. In the WAN environment, the introduction of SEV also leads to a decline in throughput and an increase in latency, although the trend is less pronounced. In this case, the inherent communication delay of the WAN environment becomes the primary factor affecting throughput and latency.

#### D. Comparison with Counterparts

**Comparison with XClaim.** A cross-chain exchange for XClaim [21] consists of the locking of the currencies on the backed chain (*i.e.*, source blockchain), the issuing of



(a) Throughput in LAN



(b) Latency in LAN

Figure 6: System performance with/without SEV in LAN and WAN.

Table V: Comparison with state-of-the-art counterparts.

	XClaim	ZK-Bridge	Tesseract	MERCURY
Cost (gas/tx)	>388882	>227000	239181	124836
USD (\$)	>22.17	>12.9	13.63	7.11

cryptocurrency-backed assets (CBAs) on the issuing chain (*i.e.*, target blockchain). As shown in Table IV, the cost for processing one cross-chain exchange in MERCURY is about 125K gas. In contrast, the on-chain cost of XClaim for a cross-chain exchange is about 389K gas (Table V), representing a 67.87% increase compared to MERCURY. Besides, to synchronize the state of the backed chain to the issuing chain, XClaim incurs a daily cost of 28380K gas (1401.97 USD) to relay blocks, which is a significant additional expense that cannot be overlooked.

**Comparison with ZK-Bridge.** ZK-Bridge [22] utilizes zero-knowledge proofs that are generated off-chain and verified on-chain. In contrast, MERCURY employs off-chain transaction processing and verifies multi-signatures on-chain. We specifically focus on two fundamental steps common to both protocols: the generation of proofs/multi-signatures, and the on-chain verification. ZK-Bridge’s transaction proof generation takes approximately 18s, while, as depicted in Fig. 5, MERCURY’s latency is less than 300ms, showcasing significant efficiency improvements. The gas consumption for on-chain verification of multi-signatures in MERCURY is illustrated in Fig. 4, which is comparable with the 227K gas (Table V) required for ZK-Bridge verification.

**Comparison with Tesseract.** Tesseract [17], a TEE solution for cross-chain exchange, generates transaction pairs with the hash time lock. For a fair comparison, we deployed the HTLC on Ethereum to measure the on-chain cost [69]. A cross-chain transaction with Tesseract involves two phases: lock and withdraw. The on-chain costs for the two phases are 165K gas

and 74K gas, respectively, resulting in a total cost of 239K gas (13.63 USD). By contrast, MERCURY’s cost is 125K gas, which declines by 47.70%.

## VII. BEYOND MERCURY

In this paper, MERCURY is built atop blockchains that support smart contracts. However, it can be easily extended to blockchains (*e.g.*, Bitcoin [1]) that do not support smart contracts. The main extension is to replace the challenge-response mechanism with hash time-lock to resolve redemption issues caused by TEEs’ unavailability. We now introduce it in detail.

**Hash time-lock.** A hash time-lock is a transactional agreement used in the cryptocurrency industry to facilitate conditional payments. It incorporates both a hashlock and a timelock. The hash lock ensures that the transaction includes a hashed secret, and it can only be successfully executed by revealing this secret (*i.e.*, the preimage of the hash). The time lock sets a time limit  $T$ , specifying that the transaction can only be executed after this time has passed.

**Contract Design.** We consider the case, in which a client who expects to exchange currency  $s$  on blockchain  $\mathcal{S}$  for currency  $t$  on blockchain  $\mathcal{T}$ . First, the client sends an exchange request to the operators, who then generate a pair of transactions  $tx_S$  and  $tx_T$ , while the preimage is stored inside the TEE. The execution of the transaction  $tx_S$  falls into two cases.

- $tx_S^1$ : If the time exceeds  $T$ , the client’s money will be returned to the original address.
- $tx_S^2$ : If a secret preimage is revealed, the client’s money will be transferred to an address controlled by MERCURY.

The client signs the transaction  $tx_S$  and publishes it on the blockchain  $\mathcal{S}$ . If, within time  $T$ , the operator detects that transaction  $tx_T$  has been successfully executed, it will reveal the hash preimage on blockchain  $\mathcal{S}$ , resulting in the successful execution of transaction  $tx_S^2$ . Otherwise, after time  $T$ , transaction  $tx_S^2$  will be successfully executed, and the client will receive a refund on blockchain  $\mathcal{S}$ . As we can see, the hash time-lock ensures the atomicity of the cross-chain transaction when the operators are unavailable.

## VIII. CONCLUSION AND FUTURE WORK

In this paper, we propose MERCURY, a trust-minimized and efficient cross-chain exchange without online-client requirement. MERCURY employs a group of operators protected by TEEs as the trust root. The cost-efficient challenge-response mechanism safeguards the atomicity of cross-chain exchanges, even when all TEEs are unavailable. Furthermore, MERCURY employs a lightweight transaction verification mechanism and simultaneously incorporates multiple optimizations to reduce on-chain costs. The prototype implementation of MERCURY delivers promising results, surpassing existing state-of-the-art approaches in terms of on-chain cost and off-chain processing time. There are several future directions to improve MERCURY. First, MERCURY can be extended to support Proof-of-Work (PoW)-based blockchains like Bitcoin that do not have smart contracts. Second, more financial functions can be built on top of MERCURY, *e.g.*, lending and borrowing.

## REFERENCES

- [1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2008.
- [2] “Ethereum,” <https://ethereum.org/en/>.
- [3] D. Schwartz, N. Youngs, A. Britto *et al.*, “The ripple protocol consensus algorithm,” *Ripple Labs Inc White Paper*, vol. 5, no. 8, p. 151, 2014.
- [4] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, “Algorand: Scaling byzantine agreements for cryptocurrencies,” in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.
- [5] F. Fang, C. Ventre, M. Basios, L. Kanthan, D. Martinez-Rego, F. Wu, and L. Li, “Cryptocurrency trading: A comprehensive survey,” *Financial Innovation*, vol. 8, no. 1, p. 13, 2022.
- [6] Y. Liu, A. Tsyvinski, and X. Wu, “Common risk factors in cryptocurrency,” *The Journal of Finance*, vol. 77, no. 2, pp. 1133–1177, 2022.
- [7] “Coinmarketcap,” <https://coinmarketcap.com>.
- [8] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, “A survey on blockchain interoperability: Past, present, and future trends,” *ACM Computing Surveys*, vol. 54, no. 8, pp. 1–41, 2021.
- [9] A. Augusto, R. Belchior, M. Correia, A. Vasconcelos, L. Zhang, and T. Hardjono, “SoK: Security and privacy of blockchain interoperability,” *Authorea Preprints*, 2023.
- [10] “EOS network foundation,” <https://eosnetwork.com>.
- [11] M. Herlihy, “Atomic cross-chain swaps,” in *Proceedings of the 2018 ACM symposium on principles of distributed computing*, 2018, pp. 245–254.
- [12] S. Hu, M. Li, J. Weng, J.-N. Liu, J. Weng, and Z. Li, “IvyRedaction: Enabling atomic, consistent and accountable cross-chain rewriting,” *IEEE Transactions on Dependable and Secure Computing*, 2023.
- [13] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sánchez, A. Kiayias, and W. J. Knottenbelt, “SoK: Communication across distributed ledgers,” in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*. Springer, 2021, pp. 3–36.
- [14] Z. Yin, B. Zhang, J. Xu, K. Lu, and K. Ren, “Bool network: An open, distributed, secure cross-chain notary platform,” *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 3465–3478, 2022.
- [15] A. Xiong, G. Liu, Q. Zhu, A. Jing, and S. W. Loke, “A notary group-based cross-chain mechanism,” *Digital Communications and Networks*, vol. 8, no. 6, pp. 1059–1067, 2022.
- [16] Y. Chen, J.-N. Liu, A. Yang, J. Weng, M.-R. Chen, Z. Liu, and M. Li, “PACDAM: Privacy-preserving and adaptive cross-chain digital asset marketplace,” *IEEE Internet of Things Journal*, 2023.
- [17] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, “Tesseract: Real-time cryptocurrency exchange using trusted hardware,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications*

- Security*, 2019, pp. 1521–1538.
- [18] “Lightning network.” <https://lightning.network>.
- [19] S. A. Thyagarajan, G. Malavolta, and P. Moreno-Sánchez, “Universal atomic swaps: Secure exchange of coins across all blockchains,” in *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022, pp. 1299–1316.
- [20] J. Xu, D. Ackerer, and A. Dubovitskaya, “A game-theoretic analysis of cross-chain atomic swaps with HTLCs,” in *2021 IEEE 41st international conference on distributed computing systems (ICDCS)*. IEEE, 2021, pp. 584–594.
- [21] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. Knottenbelt, “XClaim: Trustless, interoperable, cryptocurrency-backed assets,” in *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019, pp. 193–210.
- [22] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, “ZKbridge: Trustless cross-chain bridges made practical,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3003–3017.
- [23] “Cosmos: A network of distributed ledgers,” <https://cosmos.network/whitepaper/>.
- [24] “Mercury (mythology),” [https://en.wikipedia.org/wiki/Mercury\\_\(mythology\)](https://en.wikipedia.org/wiki/Mercury_(mythology)).
- [25] W. Wang, S. Deng, J. Niu, M. K. Reiter, and Y. Zhang, “Engraft: Enclave-guarded Raft on Byzantine faulty nodes,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’22, New York, NY, USA, 2022, p. 2841–2855.
- [26] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.
- [27] “Solidity documentation,” <https://docs.soliditylang.org/en/v0.8.7/>.
- [28] “Golang,” <https://go.dev/>.
- [29] “Sepolia,” <https://github.com/eth-clients/sepolia>.
- [30] “Buy, trade, and hold 350+ cryptocurrencies on Binance,” <https://www.binance.com/en>.
- [31] “Coinbase exchange,” <https://www.coinbase.com/exchange>.
- [32] “Centralized exchange: Convenient or risky?” <https://www.binance.com/en/feed/post/419747>.
- [33] “How safe is your crypto on centralized exchanges?” <https://beincrypto.com/how-safe-is-your-crypto-on-centralized-exchanges/>.
- [34] K. Ren, N.-M. Ho, D. Loghin, T.-T. Nguyen, B. C. Ooi, Q.-T. Ta, and F. Zhu, “Interoperability in blockchain: A survey,” *IEEE Transactions on Knowledge and Data Engineering*, 2023.
- [35] L. Deng, H. Chen, J. Zeng, and L.-J. Zhang, “Research on cross-chain technology based on sidechain and hash-locking,” in *Edge Computing—EDGE 2018: Second International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25–30, 2018, Proceedings 2*. Springer, 2018, pp. 144–151.
- [36] L. Yin, J. Xu, and Q. Tang, “Sidechains with fast cross-chain transfers,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3925–3940, 2021.
- [37] “BTCRelay,” <http://btcrelay.org>.
- [38] “Simplified payment verification.” [https://wiki.bitcoinsv.io/index.php/Simplified\\_Payment\\_Verification](https://wiki.bitcoinsv.io/index.php/Simplified_Payment_Verification).
- [39] T. Gauthier, S. Dan, M. Hadji, A. Del Pozzo, and Y. Amoussou-Guenou, “Topos: A secure, trustless, and decentralized interoperability protocol,” *arXiv preprint arXiv:2206.03481*, 2022.
- [40] Y. Li, J. Weng, M. Li, W. Wu, J. Weng, J.-N. Liu, and S. Hu, “Zerocross: A sidechain-based privacy-preserving cross-chain solution for monero,” *Journal of Parallel and Distributed Computing*, vol. 169, pp. 301–316, 2022.
- [41] X. Luo, K. Xue, Q. Sun, and J. Lu, “CrossChannel: Efficient and scalable cross-chain transactions through cross-and-off-blockchain micropayment channel,” *IEEE Transactions on Dependable and Secure Computing*, 2024.
- [42] “Polkadot: Vision for a heterogeneous multichain framework,” <https://github.com/w3f/polkadot-white-paper/raw/master/PolkaDotPaper.pdf/>.
- [43] Y. Hei, D. Li, C. Zhang, J. Liu, Y. Liu, and Q. Wu, “Practical AgentChain: A compatible cross-chain exchange system,” *Future Generation Computer Systems*, vol. 130, pp. 207–218, 2022.
- [44] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, “Town crier: An authenticated data feed for smart contracts,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 270–282.
- [45] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, “Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts,” in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 185–200.
- [46] “Remote attestation.” <https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html>.
- [47] “AMD SEV remote attestation protocol.” <https://enarx.dev/docs/technical/amd-sev-attestation>.
- [48] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, “ROTE: Rollback protection for trusted execution,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1289–1306.
- [49] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, “Rollback and forking detection for trusted execution environments using lightweight collective memory,” in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 157–168.
- [50] J. Niu, W. Peng, X. Zhang, and Y. Zhang, “Narrator: Secure and practical state continuity for trusted execution in the cloud,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*,

- ser. CCS '22, 2022, p. 2385–2399.
- [51] W. Peng, X. Li, J. Niu, X. Zhang, and Y. Zhang, “Ensuring state continuity for confidential computing: A blockchain-based approach,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–14, 2024.
- [52] E. Kindler, “Safety and liveness properties: A survey,” *Bulletin of the European Association for Theoretical Computer Science*, 1994.
- [53] S. Zhang and J.-H. Lee, “Analysis of the main consensus protocols of blockchain,” *ICT express*, vol. 6, no. 2, pp. 93–97, 2020.
- [54] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, “Smart contract development: Challenges and opportunities,” *IEEE transactions on software engineering*, vol. 47, no. 10, pp. 2084–2106, 2019.
- [55] R. Pass, E. Shi, and F. Tramer, “Formal abstractions for attested execution secure processors,” in *Advances in Cryptology—EUROCRYPT 2017: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30–May 4, 2017, Proceedings, Part I* 36. Springer, 2017, pp. 260–289.
- [56] “Automata: Modular attestation.” <https://github.com/automata-network/automata-dcap-v3-attestation>.
- [57] V. Mohan, “Automated market makers and decentralized exchanges: A DeFi primer,” *Financial Innovation*, vol. 8, no. 1, p. 20, 2022.
- [58] H. Adams, N. Zinsmeister, M. Salem, R. Keefer, and D. Robinson, “Uniswap v3 core,” Uniswap.org.
- [59] A. A. Aigner and G. Dhaliwal, “Uniswap: Impermanent loss and risk profile of a liquidity provider,” *arXiv preprint arXiv:2106.14404*, 2021.
- [60] “On-chain remote attestation verification.” <https://github.com/PufferFinance/rave/>.
- [61] “Ethereum.” <https://en.wikipedia.org/wiki/Ethereum>.
- [62] “Minimal light client.” <https://github.com/ethereum/annotated-spec/blob/master/altair/sync-protocol.md#introduction>.
- [63] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, “Combining GHOST and Casper,” *arXiv preprint arXiv:2003.03052*, 2020.
- [64] I. Eyal and E. G. Sirer, “Majority is not enough: Bitcoin mining is vulnerable,” *Communications of the ACM*, vol. 61, no. 7, pp. 95–102, 2018.
- [65] C. Feng and J. Niu, “Selfish mining in Ethereum,” in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1306–1316.
- [66] J. Niu, F. Gai, R. Han, R. Zhang, Y. Zhang, and C. Feng, “Crystal: Enhancing blockchain mining transparency with quorum certificate,” *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 5, pp. 4154–4168, 2023.
- [67] A. SEV-SNP, “Strengthening VM isolation with integrity protection and more,” *White Paper, January*, vol. 53, pp. 1450–1465, 2020.
- [68] T. Frassetto, P. Jauernig, D. Koisser, D. Kretzler, B. Schlosser, S. Faust, and A.-R. Sadeghi, “Pose: Practical off-chain smart contract execution,” *arXiv preprint arXiv:2210.07110*, 2022.
- [69] “Ebaas hash time locked cross chain samples.” <https://github.com/ehousecy/htlc-samples>.