HEOV: Achieving Privacy-Preserving and High-Performance Permissioned Blockchain via Correlated-Merged Homomorphic Encryption

Anonymous submission #5

Abstract

Data privacy is essential for privacy-sensitive permissioned blockchain applications such as the medical chain. Public-key cryptography is a widely used in permissioned blockchains to safeguard data privacy. However, current studies still face two significant challenges. Firstly, transaction executions can be extremely inefficient due to the resource-intensive nature of cryptographic operations. Secondly, existing studies are unable to efficiently verify and audit the correctness of smart contract executions on encrypted data. These challenges significantly restrict the potential of cryptography-based permissioned blockchains.

This paper introduces HEOV*, a privacy-preserving and high-performance permissioned blockchain that effectively tackles both challenges.* HEOV *incorporates two fundamental cryptography primitives: Fully Homomorphic Encryption (FHE) for direct transaction executions over ciphertext, and Non-Interactive Zero-Knowledge (NIZK) proof for ensuring the correctness of transaction executions. By incorporating an correlated-merging protocol,* HEOV *minimizes the invocations of resource-intensive FHE computation, thereby ensuring data confidentiality without incurring substantial computation overhead. Capitalizing on the security guarantee offered by the blockchain,* HEOV *efficiently generates lightweight NIZK proofs, enabling* HEOV *to seamlessly incorporate FHE to support general blockchain applications. We implemented* HEOV *on the codebase of Hyperledger Fabric, one of the most popular permissioned blockchain framework. Our evaluation demonstrated that* HEOV *delivered high performance, with a throughput of up to 343 transactions per second and an average end-to-end latency of 2.59 seconds.* HEOV *attains robust data privacy with exceptional performance, rendering it suitable for deploying diverse blockchain applications.*

1. Introduction

A permissioned blockchain is an immutable ledger among explicitly identified mutually untrusted participant clients and nodes. Unlike permissionless blockchains, the permissioned blockchain is decoupled from the cryptocurrency [35] and can run traditional high-performance Byzantine-fault-tolerant (BFT) consensus protocols (e.g., BFT-SMaRT [9]) to tolerate malicious participants. Therefore, permissioned blockchains are ideal for deploying performance-sensitive and security-critical enterprise applications, such as citizen information management [31] and medical chain [32].

The execute-order-validate (EOV) workflow, introduced by

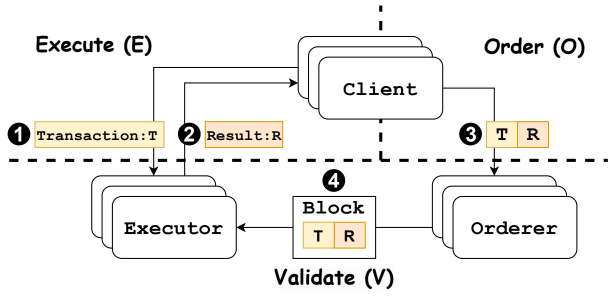


Figure 1: The EOV workflow

Hyperledger Fabric (HLF) [5], is one of the most notable permissioned blockchain workflows [5, 41, 24]. As illustrated in Figure 1, the EOV workflow processes client transactions in three phases. First, in the *execute* phase, the client submits transactions to the executor nodes for policy-based executions. A transaction's execution result is correct if certain executor nodes (specified in the application-layer endorsement policy [5]) execute the transaction and produce consistent execution results. Then, in the *order* phase, orderer nodes reach a consensus on the order of execution results. Finally, in the *validate* phase, executor nodes validate the ordered transactions and their results, commit valid ones, and abort the invalid ones.

Two advancements in the EOV workflow make it especially suitable for deploying enterprise applications. First, the EOV workflow concurrently executes all transactions to achieve a high performance [5, 41, 43]. Second, the consensus-on-result feature enables EOV to tolerate non-deterministic transactions, because all executors can commit a consistent execution result. Therefore, EOV can support applications written in general-purpose multi-threaded programming languages (e.g., Golang [?]). These unique advancements make EOV be extensively used in various enterprise applications. For example, Singapore deployed a stock exchange system on HLF [5]; Walmart built its supply chain system [1] on HLF.

However, existing EOV permissioned blockchains often involve the trade-off in data privacy. Specifically, a client must submit each transaction's plaintext to the executor nodes for execution, which may jeopardize the privacy of user data. This is particularly problematic for applications that handle sensitive data, such as patients' medical records on a medical chain [32]. Systems that handle such sensitive data are typically subject to strict privacy laws and regulations, such as the General Data Protection Regulation in Europe [52]. Therefore, ensuring data privacy in EOV permissioned blockchains is a critical issue that needs to be resolved.

Recent studies [46, 48, 44, 47, 30] have revealed the potential of homomorphic encryption (HE) and non-interactive zero-knowledge (NIZK) proof as powerful tools for safeguarding data privacy in blockchain systems. The notable study ZeeStar [46] leverages additive HE and NIZK proof to enable encrypted, privacy-preserving transaction executions among mutually untrusted participants. Specifically, blockchain nodes first execute transactions on data owned by different participants using homomorphic encryption, and then generate NIZK proofs to verify the correctness of the executions on other blockchain nodes.

Unfortunately, this innovative approach still faces the dilemma between the generality of supported applications and achieving high performance. On the one hand, adopting the fully homomorphic encryption (FHE [51]), which offers both addition and multiplication operations, can support general blockchain applications. However, to prove the correctness of transaction executions, FHE leads to intricate NIZK proofs and prohibitively high costs for generating these proofs. For instance, generating a Groth16 zk-SNARK for Paillier encryption requires over 256 GB of RAM, rendering it impractical for commodity desktop machines [46]. Moreover, generating a single proof takes several tens of seconds. Such prolonged latency is unacceptable for blockchain applications, which generally require a commit latency of several seconds [**?**]. On the other hand, adopting only partial homomorphic encryption (PHE) can achieve high performance. For instance, ??? supports only additions of data with different owners, and achieves a ???$\times$ better latency than FHE when generating the NIZK proofs [46]). However, PHE hampers the support for multiplications of data owned by different participants, thus limiting the system's generality. Overall, existing approaches can achieve only either high performance or application generality but not both.

In this study, our key insight to tackle the dilemma is that, we can leverage EOV's policy-based execution, which enables the blockchain nodes to efficiently verify the correctness of transaction executions, to obviate the necessity of verifying the correctness of executions using the NIZK proofs. With EOV's policy-based execution, each transaction is homomorphically executed on multiple executor nodes, and the correctness of the executions is validated by comparing the consistency of execution results from executor nodes. As a result, the system can seamlessly integrate with fully homomorphic execution schemes to support a wide array of general blockchain applications while still maintaining high performance.

This insight leads to HEOV, the first high-performance privacy-preserving EOV permissioned blockchain framework that supports general blockchain applications. HEOV clients leverage FHE to encrypt sensitive user data, while executor nodes execute transactions on encrypted ciphertext. By synergizing the strengths of EOV and HE, HEOV ensures the correctness of transactions' executions by embracing EOV's policy-based executions, and preserves data confidentiality by preventing executors from accessing sensitive user data in plaintext.

The integration of the EOV workflow with FHE presents two unique challenges. The first challenge is tolerating non-deterministic transactions in EOV permissioned blockchains. For a non-deterministic transaction, different executors may produce inconsistent execution results, and EOV only commits transactions with consistent execution results. However, in HEOV, the executors cannot directly compare the FHE-encrypted execution results like existing EOV permissioned blockchains, due to the random noise that FHE inserts into the ciphertexts to defend against ciphertext attacks [21]. Consequently, the executors cannot determine whether a transaction is non-deterministic or not by checking the consistency of the transaction's execution results, as the ciphertexts of execution results are inconsistent even for a deterministic transaction.

The second challenge is the potentially disastrous performance degradation when integrating FHE with EOV, compared to plain text executions. First, fully homomorphic encryption incurs a high computation overhead. Existing studies [51] have reported that FHE-encrypted transactions take 10^6x longer time to execute, which we confirmed in our evaluation (§8). Worse, the random noise that FHE inserts into ciphertext accumulates as the number of homomorphic computations increases. The ciphertext can be only decrypted when the noise is below a threshold, limiting the number of consecutive HE computations (e.g., multiplication) on the same ciphertext before decryption fails. This issue can be addressed by resetting the noise using relinearization, which can take several minutes even in efficient implementations [51].

This performance issue is further exacerbated by EOV's conflicting aborts. Due to EOV's concurrent transaction execution, when multiple concurrently executed transactions access the same key and at least one of them writes to the key (conflicting transactions), only one transaction can commit and other transactions will abort. Such conflicting transactions are prevalent in typical blockchain applications [43]. For example, HLF achieved only 24.5% of its conflict-free peak throughput in the stock trading application, as confirmed in our evaluation in §4. These two factors are intertwined, as conflicting transactions result in a large number of HE-encrypted ciphertext computations on the same key, leading to frequent relinearizations.

To tackle the first challenge, HEOV clients generate NIZK proofs to prove the consistency of execution results of different executor nodes. Instead of generating heavy-weight NIZK proof to prove the correctness of the entire transaction execution process as existing studies [46], HEOV generates only light-weight NIZK proof to prove the results' plaintext consistency without introducing prohibitive proof generation overhead. This enables HEOV to integrate FHE for supporting general blockchain applications with both addition and multiplication operations. Furthermore, the NIZK proof safeguards against malicious participants falsifying the consistency of results, ensuring compatibility with BFT safety.

System	**Data Privacy**	**Computation On Ciphertext**	**Computation On Foreign Data**	**Tolerate Non-Deterministic Smart Contract**
HLF [5]	✗	✗	✓	✓
ZeeStar [46]	✓	✓	✓	✗
Zapper [48]	✓	✗	✗	✗
smartFHE [44]	✓	✓	✗	✗
FabZK [30]	✓	✗	✓	✗
HEOV	✓	✓	✓	✓

Table 1: Compare HEOV with existing privacy-preserving blockchain systems.

To tackle the second challenge, our basic idea is to exploit the conflicting transactions, which are ubiquitous in permissioned blockchain applications, to reduce the invocation number of homomorphic computations and relinearizations. We propose a new secure computing abstraction called correlated-merged homomorphic encryption for HEOV. This approach enhances the online schedule of transactions with an offline analysis of the transaction code. During the offline analysis, HEOV analyzes the transaction code and identifies whether multiple conflicting transactions invoking the code can be merged into a single transaction. Based on the results of the offline analysis, HEOV's online schedule merges conflicting transactions and performs the necessary HE-encrypted ciphertext computations on the merged transactions. For example, when two transactions both add to the same key (e.g., two $X+1$ operations), HEOV combines the two adding transactions into a single add transaction on that key (e.g., one $X+2$ operation). This reduces the number of HE computations and relinearization required while preserving the semantics of the original transactions. Moreover, in contrast to existing permissioned blockchains with EOV consensus (e.g., HLF [5]), where conflicting transactions are resolved by committing only one and invalidating the rest, HEOV can commit all conflicting transactions without any aborts.

We implemented HEOV on the codebase of HLF, and compared HEOV with state-of-the-art privacy-preserving blockchain systems [46]. Our evaluation and analysis results show that:

- HEOV is efficient. HEOV's throughputs were up to 343 TPS and 255 TPS (§ 8) under conflict-heavy and conflict-free environments, respectively.
- HEOV achieved even higher throughput as the conflict ratio increases (§ 8.1).
- HEOV is able to counter targeted conflicting attack (§ 5.1) and no-resubmission attack (§ 5.2).

HEOV makes a pioneering effort in identifying the challenges posed by the integration of HE and EOV, achieving the best of both worlds. HEOV integrates EOV with HE to achieve data confidentiality without introducing significant performance overhead or sacrificing the generality in supporting diverse blockchain applications. The implication behind HEOV 's synergy of conflicting transactions and homomorphic encryption extends beyond the scope of the blockchain and can serve as a general transaction scheduling mechanism for other exciting research areas for distributed systems, such as the Internet of Things [54] and databases [50], where transactions often conflict, and each execution of the distributed transaction commitment on the Internet leads to substantial latency.

2. Background And Related Works

In this section, we present a background overview of HEOV and review related works in the field of privacy-preserving blockchains, which are compared in table 1. This section aids in our understanding of the contribution and significance of HEOV.

2.1. Execute-Order-Validate Permissioned Blockchain

Permissioned [5, 16] and permissionless [45, 23] blockchains are maintained by multiple mutually untrusted organizations, but they differ significantly in their protocols. In a typical permissioned blockchain, nodes are incentivized to follow the protocol using cryptocurrency because the protocol requires certain types of proof from the nodes. For example, proof-of-work [35] demands a high level of computation power, while proof-of-stake [22, 53] requires a large number of holdings in the associated cryptocurrency. In contrast, a typical permissioned blockchain employs a BFT consensus protocol across multiple consensus nodes to tolerate a certain number of malicious nodes, instead of mandating nodes to offer proof of work or stake. For example, HLF is usually deployed with dozens of orderer nodes [45], whereas Cosmos operates with 175 validator nodes [16]. Thus, permissioned blockchains generally exhibit better performance compared to permissionless blockchains, owing to the superior performance of BFT protocols, such as PBFT [13] and HotStuff [2], in contrast to the consensus protocols used by permissionless blockchains. HEOV, inherited from HLF, can concurrently run one or multiple orderer nodes forming a BFT ordering service that can determine the order of transactions for each block within a few hundred milliseconds.

Permissioned blockchains can be further categorized into two sub-types based on their workflows: order$\rightarrow$execute (OE) and execute$\rightarrow$order$\rightarrow$validate (EOV). The OE workflow takes a pessimistic approach: (**O**) first determining the order of transactions and (**E**) then sequentially executing them on all nodes. One benefit of the OE approach is that transactions are always committed without aborts. However, the bottleneck in the ordering phase can make inefficiency a serious problem for OE blockchains, and even trigger security issues such as buggy or malicious contracts that cause significant financial losses [29]. In contrast, the EOV workflow is an optimistic approach. (**E**) Executor nodes execute various transactions submitted by various clients in parallel for high performance without caring about their order and send back execution results to the clients for sanity checks. (**O**) Clients then submit transactions with results to orderer nodes in order to reach a consensus on the order of transactions received within a specific time frame. (**V**) After determining the order of transactions, orderers assemble them into a block and dispatch it to all executors. Executors then validates each transaction in the block sequentially, committing valid ones and aborting invalid ones. In short, the EOV workflow achieves high throughput at the cost of aborting conflicting transactions. HEOV adopts the EOV workflow for parallelism and applies the correlated-merging protocol (i.e., pre-submission transaction merging) to minimize potential conflicting aborts, laying a solid foundation for the high performance of HEOV.

2.2. Homomorphic Encryption

Homomorphic encryption (HE) is an extension of public-key cryptography [4] that supports direct computation over encrypted data without exposing the underlying plaintext. The capability is based on the homomorphism between plaintext and ciphertext spaces. In algebra, a homomorphism is a structure-preserving map between two algebraic structures of the same type (e.g., two rings) that preserves the operations of the structures. For instance, if $f : X \rightarrow Y$ is a map between two sets X and Y equipped with the same structure, and $\oplus$ is an operation of the structure (suppose $\oplus$ is a binary operation for simplicity), then f satisfies the property

$$f(a \oplus b) = f(a) \oplus f(b)$$

for every pair of elements a, b of X, and $a \oplus b$ of Y. It is often said that f preserves the operation or is compatible with the operation.

There are different types of HE schemes depending on the classes of arithmetic operation they support. Two common types are (1) partially homomorphic encryption (PHE), which allows only one specific kind of operation (e.g., addition or multiplication), and (2) fully homomorphic encryption (FHE), which allows arbitrary combinations of multiple kinds of computation and is the strongest notion of homomorphic encryption. Despite simplicity, PHE is restricted by its limited computation capability and thus is not adopted by HEOV. For example, the Paillier cryptosystem [38, 39] only supports homomorphic addition. In contrast, FHE schemes incur higher computation overhead but offer powerful general computation capability, aligning with HEOV's objective of supporting general smart contracts. Among several popular FHE schemes (e.g., BFV [11, 19], BGV [12], and CKKS [14]), HEOV adopts the BFV scheme because it provides a good trade-off between security and performance. HEOV employs the lattigo [34] library, whose BFV implementation can perform an arithmetic computation on 58-bit unsigned integers within tens of milliseconds under 128-bit security, ensuring that HE computations do not significantly jeopardize HEOV performance.

2.3. Non-Interactive Zero-Knowledge Proofs

Zero-knowledge proof (ZKP) is a cryptographic primitive that enables a prover P to prove to a verifier V that P knows a secret without revealing that secret. Non-interactive zero-knowledge proof (NIZKP) [20, 25] is a specialized version of ZKP that eliminates the interaction between P and V. In other words, once P generates an NIZKP using a proving key, V can verify the proof as long as V has access to the corresponding verifying key. P need not be online during the verification process. Formally, given a proof circuit ϕ, a private input w, and some public input x, P can use an NIZKP to prove knowledge of w satisfying a predicate $\phi(w; x)$.

Zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) [10, 26, 55] are a type of generic NIZKP construction that allows any arithmetic circuit ϕ and guarantees constant-cost proof verification in the size of ϕ, making them suitable for blockchain applications [47, 6, 17, 37, 3]. Among many zkSNARK constructions, Groth16 [25] is a circuit-specific preprocessing general-purpose construction that has become a de facto standard used in several blockchain projects [46, 40] due to the constant size of proof (several kilobytes) and short verifying time (tens of milliseconds).

HEOV adopts Groth16 as its NIZKP solution and the gnark library [15] as its NIZKP implementation. The adoption of NIZKP is crucial for two reasons. Firstly, NIZKP are necessary to enable conditional branching in the context of encrypted data. Since direct comparison between two ciphertexts is currently infeasible, if-statements that are common in many high-level programming languages cannot be applied in this case. NIZKP offer a solution to this problem. Secondly, NIZKP are needed to prove the correctness of transaction results from various executors, which helps prevent malicious clients from forging transaction results.

2.4. Related Works

We now review previous works related to HEOV.

Smart Contract Privacy. Privacy is a critical aspect in the world of smart contracts, and several previous works have accomplished different levels of privacy.

Both SmartFHE [44] and ZeeStar [46] achieve *data privacy*, meaning that the value of specific data can only be revealed by those who have access to the decryption key, typically the data owner. Other participants, whether benign or not, cannot learn anything about the underlying plaintext except the identity of the data. SmartFHE [44] proposes a framework to support private smart contracts using FHE and NIZKP, while ZeeStar adopts ElGamal [33] as an additive homomorphic scheme and has developed a domain-specific language based on Solidity and a compiler that takes a public smart contract as input and outputs a semantically equivalent one that can operate on private inputs.

Zapper [48] and FabZK [30] go beyond *data privacy* and also hide the identity of the data being accessed, a property known as *key privacy*. This is the strongest notion of security in systems related to HEOV. Zapper achieves *key privacy* by leveraging a novel combination of an oblivious Merkle tree construction and an NIZK processor, but this comes at the cost of sacrificing the expressiveness of smart contracts, which will be discussed shortly. FabZK achieves *key privacy* by using ZKP, verifiable Pedersen commitments, and a special data structure called shared tabular structured ledger, which stores only encrypted data from each transaction (e.g., payment amount) and conceals the relevant members of transactions (e.g., payer and payee).

Smart Contract Generality. Smart contract generality is a major concern in the field of privacy-preserving smart contract research, and several previous works have achieved different levels of smart contract generality using various approaches, as compared in table 1.

ZeeStar realizes general-purpose smart contracts by supporting a tailored version of Ethereum [53] smart contracts. However, ZeeStar only allows additive homomorphic encryption and partially implements multiplication for limited scenarios by iteratively executing a specific number of homomorphic additions. This seriously undermines the applicability of ZeeStar and is not aligned with its claim to support general smart contracts. Furthermore, ZeeStar uses NIZKP to prove the correctness of offline HE computation, which requires significant time for proof generation, as confirmed in its evaluation. HEOV addresses these two pain points in the following ways. First, HEOV adopts FHE, specifically the BFV scheme [11, 19], that enables both homomorphic addition and multiplication. Second, HEOV simplifies the NIZKP by harmonizing the EOV workflow and HE, as discussed in §4.2, reducing the proof generation time and, therefore, the end-to-end latency.

Zapper sacrifices the generality of smart contracts, hindering its applicability. Firstly, Zapper prohibits the use of control flow, loops, or any operation that modifies instructions at runtime. Secondly, Zapper can only perform computation on self-owned data, which means a Zapper transaction cannot involve data belonging to multiple users. These limitations pose serious restrictions on the generality of Zapper's smart contracts.

SmartFHE currently only implements the single-key variant where all private inputs for a transaction must be owned by the same party. The multi-key variant that supports computation among private inputs owned by different parties is currently impractical, according to its authors, which is exactly the second problem that hinders Zapper's generality. Therefore, SmartFHE also suffers from limited expressiveness of smart contracts.

FabZK [30] is based on HLF [5] and achieves a stronger smart contract generality than ZeeStar, Zapper, and SmartFHE. However, FabZK faces a serious scalability problem because every transaction causes an update of the values of all organizations, which puts heavy pressure on both the transaction end-to-end latency and the space consumption. Additionally, it is difficult to add new organizations to the FabZK network on-the-fly due to the restriction of the tabular structured ledger.

3. Overview

3.1. System Model

HEOV consists of three types of participants: clients, executors, and orderers. The latter two are often referred to as nodes because they act as "servers" in an HEOV network that clients interact with. An HEOV network is typically maintained by multiple organizations, each consisting of clients and nodes. Members of the same organization are mutually trusted, but those who belong to various organizations do not trust each other.

Clients. Clients are end-users who submit transaction proposals to executors for execution and to orderers for ordering. Additionally, clients are responsible to generate NIZKP to prove the consistency of transaction results.

Executors. All executors in HEOV are responsible for evaluating transactions and maintaining a local copy of the blockchain, similar to HLF. HEOV executors utilize libHEOV (§7.1), which offers a set of API that support NIZKP and HE operations, to perform execution of transactions with homomorphic encrypted data. libHEOV is tailored for EOV contract execution and comes with a set of pre-defined parameters for HE, therefore smart contract developers do not need to be cryptography experts to efficiently incorporate smart contracts with HE operations.

Besides, a *lead executor* is elected for each organization to implement the correlated-merging protocol, as show in figure 2. Lead executors are not different from other executors in terms of transaction execution and blockchain maintenance, but they are assigned with the following extra tasks.

- Receive transaction proposals from clients of its organization.
- Optimize transactions by merging multiple transactions targeting the same set of keys into a single transaction.
- Dispatch transaction proposals to other executors and orderers for execution and ordering, respectively.
- Pass transaction responses to clients.

Note that clients are not required to directly submit their transaction proposals to a lead executor as other executors will finally forward the proposals to it.

Orderers. Orderers in HEOV are the same as in HLF. Specifically, they determine the order of transactions via some consensus protocol (e.g., BFT) and pack them into individual blocks which are ultimately validated and recorded by relevant executors.

HEOV identifies all participants through a membership mechanism similar to HLF, where each participant is issued a pair of public/secret keys that encode its identity information. This mechanism enables access control to HEOV resources and, more importantly, makes all operations on HEOV auditable and traceable since any transaction is signed by a specific participant with the corresponding privilege.

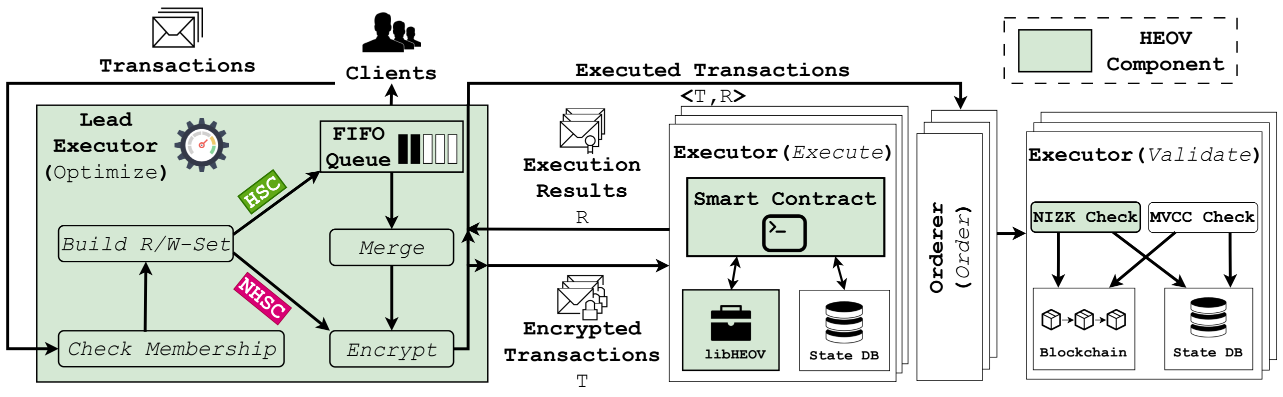


Figure 2: The HEOV Runtime Workflow. HEOV components are highlighted in green. An "HSC" transaction invokes a Homomorphic Smart Contract, while the "NHSC" one does not.

3.2. Threat model

HEOV adopts the Byzantine failure model [13, 8], where malicious orderers run BFT consensus protocols that tolerate up to f malicious orderers out of $3f+1$ total orderers.

HEOV groups participants (i.e., clients, optimizers, executor nodes, and orderer nodes) into organizations. Participants are mutually trusted only when they belong to the same organization. This implies that clients and nodes are mutually untrusted, as an organization either only contains clients or nodes. Any clients or nodes can be malicious, in which case they are called attackers (denoted as A).

3.3. Workflow Overview

The runtime workflow of HEOV can be summarized into several steps, as shown in Figure 2.

Step 1: optimization. Unlike HLF where executors execute transactions on reception, HEOV elects one lead executor for each organization to be the agent between clients and nodes. For each block, the lead executor collects transaction proposals from clients, temporarily store them in a First-In-First-Out (FIFO) queue, and detects whether there are two or more consecutive proposals invoking the same homomorphic smart contract and modifying the same sets of keys. If found, these proposals are tagged as *mergeable* proposals. At the moment of block generation, the lead executor would merge consecutive mergeable proposals into one proposal, while leaving non-mergeable proposals as is.

Step 2: encryption. Before dispatching proposals to other executors, the lead executor encrypts the proposal arguments with all the relevant organizations' public keys and signs each merged transaction with its certificate. This is indispensable because existing BFV computation is only possible on ciphertexts that are encrypted with the same key. When involving multiple organizations (i.e., data encrypted with different keys), the plaintext must be encrypted into different ciphertexts with the relevant organizations' keys to ensure BFV computation correctly conducted.

Step 3: dispatch. The HEOV lead executor dispatches the newly signed transactions to other executors. To prevent the loss of clients' signatures, they are gathered as metadata of the newly signed transactions. Additionally, for auditability purposes, a client is notified of the corresponding transaction ID when receiving a response from the lead executor.

Step 4: execution. On receiving a proposal from a lead executor, the HEOV executor invokes the corresponding smart contract, signs the proposal, and sends it back to the lead executor along with the executed result. Note that an HEOV executor has no knowledge of the foreign data involved (i.e., the data that are not owned by that executor's organization), as it has no access to the corresponding decryption key (i.e., private key), which is guarded by the respective organization.

Step 5: proof generation. Before submitting a transaction for ordering, the lead executor generates an NIZKP to prove the plaintext consistency of the results produced by different executors. The validity of the NIZKP implies the correctness of the execution result, as guaranteed by the correctness 6.1 of HEOV. More importantly, the NIZKP can be generated without re-running the smart contract to separately check the correctness of each data update, reducing the NIZKP complexity and therefore the generation time.

Step 6: consensus. HEOV orderer nodes run an instance of a BFT protocol (e.g., BFT-SMaRT [9]) to reach a consensus on the order of transactions in one block. Same to HLF, HEOV treats the BFT protocol as a blackbox, allowing HEOV to enjoy the mature safety and performance optimizations from existing BFT protocols.

Step 7: commit. HEOV executor nodes commit a transaction only after a block is delivered by orderers and the transaction result is valid, meaning it does not overlap in keys with previous valid transactions. Invalid transactions do not affect the state of HEOV but are still included in the block for integrity and future auditability.

The highlights of the HEOV runtime workflow can be summarized as below.

- NIZKP is generated to prove the result correctness.
- The correlated-merging protocol is used to optimize merge-

able transactions, reducing the number of conflicting transactions and HE operations.
- Lead executors safeguard the private keys for their organizations, ensuring data confidentiality.
- HE computation and NIZKP operations are performed with the help of libHEOV.

4. Protocol

This section describes the two protocols ruling the normal execution of HEOV: (1) the offline homomorphic smart contract analysis protocol (§4.1), which determines whether a smart contract is homomorphic and therefore optimizable, and (2) the runtime protocol (§4.2), which specifies the mechanism for HEOV runtime execution.

4.1. Offline Homomorphic Smart Contract Analysis

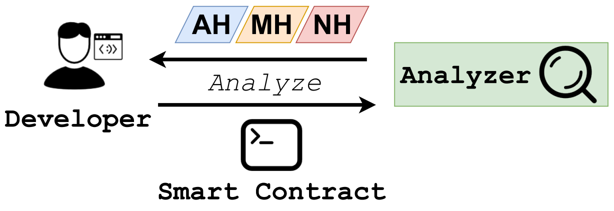


Figure 3: Offline Analysis Protocol. "AH", "MH", and "NH" means additive, multiplicative, and non-homomorphic, respectively.

HEOV's offline analysis determines whether a smart contract *SC* is homomorphic. *SC* is homomorphic *iff* multiple *SC* transactions modifying the same set of states can be merged into a single transaction without tampering with the execution results. Before smart contract deployment, a smart contract analyzer is run to conduct the offline analysis. As shown in figure 3, after analysis, the analyzer generates metadata representing whether the analyzed smart contract is additive homomorphic (**AH**), multiplicative homomorphic (**MH**), or non-homomorphic (**NH**). The metadata are critical because without them, the lead executors are not permitted to perform transaction merging.

HEOV analyzes smart contracts with random interpretation [3], which generates a different set of inputs and initial states to capture the behavior of all possible control flow paths. As a first step, two random input sets I_1 and I_2, along with a random initial memory state M, are generated for the candidate *SC*. The memory state contains the smart contract outputs and all keys written by the smart contract. Then, the smart contract is interpreted in two different ways. First, HEOV executes *SC* twice consecutively. On the initial state M, the first execution takes I_1 as input and produces the states M_{I_1} as output. Then, M_{I_1} is used as the initial state of the second execution, which takes I_2 as input and produces the final state M_{I_1,I_2} as output. Second, in parallel to this, HEOV executes the smart contract once on the same initial state M with input I_1+I_2 and produces $M_{I_1+I_2}$. If $M_{I_1,I_2}=M_{I_1+I_2}$ for all control flow paths, the smart contract is additive homomorphic. Similarly, if $M_{I_1,I_2}=M_{I_1\times I_2}$, the smart contract is multiplicative homomorphic.

4.2. Runtime

This section describes the runtime protocol that specifies the rules for transaction execution. The whole process is visualized in figure 2.

Phase 1: Submission. The first step of the runtime protocol is client submitting a transaction proposal to an executor of its organization. In the event that the executor receiving the proposal is not a lead executor, it will proceed to forward the proposal to the designated lead executor. The process of submission adopts mutual TLS, offering dual benefits of safeguarding the confidentiality of proposal arguments and requiring both parties to authenticate themselves by presenting certificates to establish their identities.

Phase 2: Pre-execution preparation. Upon receiving a transaction from a client, the lead executor undertakes a series of sequential preparation tasks: (**phase 2.1**) verifying the client's membership, (**phase 2.2**) constructing the read-write set, (**phase 2.3 and 2.4**) merging homomorphic transactions, and (**phase 2.5**) encrypting transaction arguments.

Phase 2.1: Membership verification. After receiving a proposal, the lead executor first verifies whether the client belongs to the same organization by examining the client's TLS certificate. If the client's certificate is expired or fails to demonstrate the client's membership in the lead executor's organization, the lead executor promptly rejects the transaction proposal and notifies the client of the failure. To counteract against Denial-of-Service (DoS) attacks, the lead executor employs an IP-based blacklist strategy. This approach involves maintaining a record of IP addresses associated with attackers who have submitted a certain number of failed or invalid proposals. By employing this strategy, the lead executor effectively filters out traffic related to DoS attacks, thereby safeguarding the smooth execution of subsequent steps.

Phase 2.2: Read-write set construction. The read-write set for a transaction consists of two sets: the read set and the write set which encompass the states being read and written, respectively. The purpose of constructing a read-write set is to identify the owner of the value being written. This enables the selection of the appropriate public key required to encrypt the value into ciphertext.

Phase 2.3: Transaction diversion. During phases 2.3 and 2.4, lead executors handle transactions differently based on the homomorphic property of the smart contracts they invoke. The determination of this property is carried out through the offline homomorphic smart contract analysis protocol, as referenced in §4.1. For non-homomorphic smart contracts, correlated-merging is not applicable so that proposals for these contracts undergo direct encryption specified in phase 2.5. Conversely, for homomorphic smart contracts, a distinct approach is adopted. Instead of immediate encryption, proposals are temporarily collected into a FIFO queue, which stores all the proposals that will be packaged into the same block, thereby facilitating potential merging of these proposals.

Phase 2.4: Correlated Merging. The lead executor proceeds to iterate over the FIFO queue, performing the merging of consecutive correlated proposals that invoke the same homomorphic smart contract and exhibit overlap in their read-write sets. Through the merging process, the number of transaction proposals may be reduced, as certain proposals become integrated into newly merged ones. To illustrate, let's consider an example involving two correlated $transfer$ transactions, namely TX_1 and TX_2. These transactions share two identical relevant parties and possess respective inputs, denoted as I_1 and I_2. Following the merging operation, a newly merged transaction, denoted as TX_{1+2}, is created. This merged transaction incorporates the combined input of $I_1 + I_2$ and effectively replaces both TX_1 and TX_2.

Phase 2.5: Homomorphic encryption. Before submission, the arguments of each proposal undergo encryption using public keys that correspond to the identities listed in the write set. This encryption step is crucial because each argument may be utilized on different ciphertexts owned by clients from different organizations. For example, suppose *Sam* transfers M tokens to *Rachel*, and their organizations are Org_1 and Org_2, respectively. In this case, the plaintext value M must be encrypted into two ciphertext, $Ct_{M,1}$ and $Ct_{M,2}$, using Org_1's and Org_2's public keys, respectively.

Phase 3: Dispatch. The lead executor proceeds to sign the proposal and dispatch it to other executors in accordance with the endorsement policy. The client transaction ID and signatures of the proposal are preserved as metadata to the proposal, allowing clients to validate that their proposals have been accurately executed.

Phase 4: Execution. Similar to HLF, an HEOV executor carries out the necessary smart contract execution and then sends an execution response to the lead executor. However, unlike HLF, HEOV smart contracts leverages the capabilities of libHEOV, which enpowers them to perform HE and NIZKP operations. Additionally, HEOV smart contracts use NIZKP as an alternative to the common *if* statements found in most programming languages. In HEOV, *if* statements are neither recommended nor allowed when ciphertext is involved in the condition expression, because HE ciphertext does not support comparison operations such as $<$, $>$, or $=$, and executors are not authorized to access private keys, making it impossible to decrypt ciphertext and compare the underlying plaintext. To address this, HEOV smart contracts generate NIZKP that validate the execution results. These proofs are included alongside the transaction results and stored on the blockchain for future reference.

Phase 5: Correctness NIZKP Generation. Before submitting a transaction for ordering, the lead executor is responsible for collecting results from the previously selected executors who executed the transaction. Each result comprises ciphertext and endorsement information, which includes the acknowledgment of the transaction. The lead executor proceeds to decrypt and compare all the collected ciphertexts to ensure the *plaintext consistency* of the result. This decryption process is performed by the lead executors of respective ciphertext owners. If a ciphertext belongs to another organization, the lead executor requests the corresponding organization to validate the plaintext consistency of the ciphertext results. To consider a result valid, there must be a quorum, typically a majority, of ciphertexts referencing the same plaintext. Only transactions with valid results are eligible for submission for ordering. To ensure the integrity and prevent forgery of results, the lead executor generates an NIZKP to prove the plaintext consistency. This proof is also stored on the blockchain and is publicly verifiable. By employing this approach, the lead executor is unable to create a fake result or tamper with the on-chain data, as the verification process ensures the accuracy and integrity of the results.

Phase 6: Ordering. HEOV orderers run a BFT consensus protocol (e.g., BFT-SMaRT [9]) to reach a consensus on the order of transactions in one block. The block is then distributed to all executors for validation once finalized by orderers.

Phase 7: Validation. Upon receiving a block from orderers, an executor follows a sequential process to validate each transaction within the block, such as verifying the consistency NIZKP associated with the transactions. If validation succeeds, the executor applies the transaction result to the local world state database, ensuring the changes made by the transactions are accurately reflected in the executor's local database. Once all the transactions in the block have been validated and processed, the block is permanently appended to the local blockchain. As all executors take deterministic and consistent actions for the same block, all local copies of blockchain will maintain consistent, therefore ensuring the consistency of HEOV.

Phase 8: Response. As a final step, the lead executor sends a response to the client who proposed the transaction. This response informs the client whether their transaction has been accepted or rejected.

5. Attacks and Countermeasures

This section defines two potential attacks (§5.1 and §5.2) and proposes the countermeasure of HEOV.

5.1. Defense Against Targeted Conflicting Attack

The targeted conflicting attack is a type of Byzantine performance attack that aims to cause aborts on a targeted key, seriously degrading the performance of EOV blockchains. The attack workflow typically involves the attacker A choosing a victim key K_V and constantly submitting a series of attack transactions T_A accessing K_V. As a result, a benign transaction T_B submitted by a benign client will be marked as conflicted and thus invalid unless T_B is executed earlier than all T_A.

The targeted conflicting attack has two notable features. First, such attacks are easy to conduct and have a high success rate. Suppose the attacker A submits T attack transactions for a victim transaction T_B, and each transaction shares the same probability of being the first transaction to be executed. The success rate of such attacks is as high as $\frac{N}{N+1}$, seriously undermining the valid throughput of benign clients.

Second, it may be difficult to detect such attacks. There is no notable distinction between normal aborts and aborts caused by attacks. Furthermore, benign transactions are still possible, albeit with a low probability, to be accepted. These make detection of such attacks a complex job.

The optimization technique used by HEOV addresses targeted conflicting attacks by eliminating their basis. Specifically, correlated merging merges multiple transactions that access identical keys into one transaction before submitting them to executors. As a result, aborts are significantly reduced, leaving no space for targeted conflicting attacks.

5.2. Defense Against No-Resubmission Attack

No-resubmission attack is another type of performance degradation strategy that exploits a feature of the EOV workflow. In HEOV, lead executors are responsible for collecting the executed transaction received from executors and resubmitting them to orderers for ordering. To conduct a no-resubmission attack, the attacker A simply skips the re-submission process. Although simple, the attack is detrimental because it wastes a significant amount of computation power of executors, especially when smart contracts perform heavy HE computations.

Distinguishing between a no-resubmission attack and a legitimate action can be difficult. A benign transaction can be executed with different results from different executors, especially for non-deterministic smart contracts. In this case, the results of this round of execution are dropped, and no-resubmission is a required action.

To reduce the damages from no-resubmission attacks, HEOV adopts a per-executor counter-based protocol where each executor maintains a counter for each lead executor. The counter records the number N_{nr} of no-resubmission instances of its corresponding lead executor within a certain period. If the number exceeds a configured exceeds a configured upper bound N_{ub}, then the executor rejects the execution of transactions from that lead executor for a specified time T_r. The values of N_{nr}, N_{ub}, and T_r are configurable in HEOV. By controlling these parameters, HEOV still allows for benign no-resubmission actions while minimizing the number of successful no-resubmission attacks.

6. Analysis

This section defines and proves the three guarantees of HEOV: correctness, liveness, and confidentiality.

6.1. Correctness

Definition 1 (Correctness). *For any two benign nodes $Node^1$ and $Node^2$, they hold the same world state with the n-th block as the latest block. Let $Block^1_{n+1}$ and $Block^2_{n+1}$ be the $(n+1)$-th block received by $Node^1$ and $Node^2$, respectively. $Block^1_{n+1}$ and $Block^2_{n+1}$ are consistent and deterministic.*

HEOV ensures the correctness of smart contract execution by satisfying the properties of *ACID* and BFT safety, which ensures the correctness of HEOV in transaction level and block level, respectively.

The transaction execution of HEOV satisfies the *ACID* properties, which are the foundation of all reliable failure-tolerant transaction processing systems, and are indispensable as concurrent transaction execution is possible and common in EOV workflow.

- *Atomicity*. HEOV executes and commits (or aborts) each transaction as a whole.
- *Consistency*. HEOV only commits valid transactions with correct execution results (conform to the smart contract's logic) that satisfy the application-level endorsement policy 4.2. Specifically, the execution results of a transaction are considered valid as long as enough executors digitally sign the result.
- *Isolation*. HEOV inherits the isolation property from the EOV workflow. In other words, each HEOV transaction operates as if it is the only transaction executing, without interference or conflicts with other transactions.
- *Durability*. Same as existing EOV permissioned blockchains [5], HEOV inherits the durability guarantee from EOV's consensus protocol. Executors can retrieve the blockchain's transactions from their local copies of blockchain.

HEOV achieves equivalent BFT safety as existing EOV blockchains [5] through the presence of two crucial properties.

- *Block consistency*. HEOV treats the consensus protocol as a "black box", thereby enjoying the mature consistency (safety) guarantee of existing BFT consensus protocols [9] and their implementations. This ensures that all executors process and append the same set of blocks in an identical order determined by orderers.
- *State consistency*. Every executor follows the same deterministic protocol (§4.2) to validate each transaction in a given block. A transaction's result is committed to the state database *iff* all of its results produced by various executors are consistent and do not modify any key that has already been modified by a previous transaction in the same block. In this way, all executors maintain a consistent state and a local copy of blockchain after validation of a newly appended block.

6.2. Liveness

Definition 2 (Liveness). *Let C be a benign client and T be a well-formed transaction submitted by C and accepted by the lead executor LE. T will eventually be committed to* HEOV*, even in the event of a crash of C or a reboot of LE.*

To successfully commit T to HEOV, two liveness requirements need to be met.

Firstly, the orderers must include T in a publicly agreedupon clock to guarantee its proper ordering and inclusion in the blockchain. This requirement is satisfied as HEOV inherits the BFT liveness guarantee from its BFT protocol [9] running in EOV's ordering phase.

Secondly, T must be submitted to and executed by executors during the execution phase. This is addressed by the combined liveness guarantees of HEOV lead executor and the inherited EOV workflow. The liveness guarantee of lead executor, denoted as LE, ensures that once T has been accepted, it is stored in a FIFO queue that persists data to an underlying non-volatile storage. In the event of an unexpected crash, T is not lost and will be finally submitted to executors after LE reboots. The inherited EOV workflow adopts the endorsement policy [5], an application-level agreement protocol among participating organizations. This policy ensures that a certain number of malicious executors cannot tamper with the final correct result, as long as there are a sufficient number of benign organizations executing the transactions. For example, in a typical token transfer application with three organizations (*orgBankA*, *orgBankB*, *orgBankC*), the endorsement policy is *majority(orgBankA, orgBankB, orgBankC)*, which requires that every T must be endorsed by at least two of the three organizations and ensures that at most one organization can act maliciously.

6.3. Confidentiality

Definition 3 (Confidentiality). *Suppose C is a smart contract following the guideline of §7.2. With this setup, data owners can store and process the ciphertext CT without compromising the confidentiality of the corresponding plaintext PT. An active attacker A cannot deduce PT from CT (storage confidentiality). Moreover, even if A can observe the execution of C and knows the type of C being executed, A is unable to infer any specific details about the changes made to PT (computation confidentiality).*

HEOV is instantiated with two specific cryptographic systems: (1) the BFV [11, 19] scheme which achieves IND-CPA security based on the Decisional Ring Learning With Errors (D-RLWE) problem, and (2) Groth16 [25], which is a computationally sound and perfectly NIZKP system. Suppose a probabilistic polynomial time attacker A corrupts a participating organization and can observe the execution of all transactions. HEOV provides *storage confidentiality* because A is unable to computationally distinguish the plaintext of a given ciphertext for which A has no corresponding private key (decryption key), thanks to the IND-CPA security of the BFV encryption. Furthermore, transaction arguments are always hidden in ciphertext form throughout the transaction execution, making it probabilistically indistinguishable for any two various transactions that refer to the same set of keys. In other words, A cannot learn any information beyond the identities of relevant keys during transaction execution, thereby achieving *computation confidentiality*.

7. Implementation

We implement HEOV on the codebase of HLF v2.2 [5]. HEOV adopts the same smart contract syntax as HLF, with the difference being that smart contracts need to be rewritten to use the libHEOV API in order to leverage the HE operations and NIZKP support, which will be discussed in §7.1.

7.1. libHEOV

HEOV provides a developer-friendly toolkit called libHEOV that eliminates the need for cryptography expertise and simplifies the use of HE and NIZKP functionalities. libHEOV harmonizes two mature libraries, lattigo [34] and gnark [15], which encapsulate the arithmetic computations required by HE and NIZKP operations, respectively.

Cryptography Arguments. libHEOV comes with a set of pre-defined cryptography arguments out-of-the-box (which will be discussed shortly) that strike a balance between security, speed, and memory consumption. Additionally, libHEOV offers high configurability, enabling developers to customize these arguments to achieve their desired trade-off between security and performance.

Homomorphic Encryption. libHEOV employs the BFV scheme [11, 19] as its fully homomorphic encryption solution. The BFV scheme used in HEOV ensures 128-bit security in the classic setting and supports 58-bit unsigned integer arithmetic. In modern commodity servers, an HE multiplication under this setting only takes tens of milliseconds [34]. Furthermore, the 58-bit unsigned integer is large enough to cover the value ranges of many general applications without worrying about potential overflow issues.

Non-Interactive Zero-Knowledge Proof. libHEOV uses Groth16 [25] on the BN254 elliptic curve as its NIZKP solution. Like other systems rely on Groth16 zk-SNARKs [46, 6], our implementation requires a circuit-specific trusted setup, which can be executed using secure multi-party computation [7]. Although the trusted setup process incurs an overhead, we consider it trivial as it is a one-shot procedure and therefore not included in the later evaluation section.

7.2. Rewrite Smart Contract With libHEOV

To enable HE and NIZKP support, a typical HLF smart contract needs to be rewritten in accordance with the specifications of HEOV stated below. This rewriting process should be performed manually by developers as each smart contract may possess ambiguous semantics that might confuse a transpiler, making automated transpiling a challenging task.

- **Use encrypted variable type *euint* for private data**. An *euint* variable is the ciphertext encrypted from its underlying plaintext. Throughout the execution of HE computation, the plaintext of an *euint* variable is never revealed without access to the relevant private key, ensuring the confidentiality of private data.
- **Use libHEOV functions to express HE and NIZKP oper-

ations. For HE computation, libHEOV provides *Add* and *Mul* functions that can be used to replace the standard addition and multiplication logic in the source smart contract. For NIZKP operations, developers can utilize the *ProofGen* and *ProofVerify* functions to generate new proof instances and verify a proof with specific public input.

- **Use NIZKP instead of if-statements**. Although most modern programming languages support if-statements or similar structures to control program flow based on conditional expressions, these statements cannot handle ciphertext directly due to the lack of support for direct comparison. As an alternative, HEOV employs NIZKP to simulate if-statements on ciphertext. However, it is important to note that NIZKP cannot act as a perfect replacement for if-statements because, if a proof is verified to be false, the program flow will exit without executing the remaining instructions.

```
function transfer(sender Account, receiver Account, amount
     int) {
  if db.HasKey(sender) || db.HasKey(receiver) {
    return error("incomplete sender's or receiver's
     information");
  }

  sBalance = db.GetState(sender)
  if sBalance < amount {
    return nil
  }

  rBalance = db.GetState(receiver)
  sBalance -= amount;
  rBalance += amount;

  db.SetState(sender, sBalance)
  db.SetState(receiver, rBalance)
}
```

Listing 1: Transfer Smart Contract (Not-Rewritten)

```
function transfer(sender Account, receiver Account, amount
     euint) {
  if db.HasKey(sender) || db.HasKey(receiver) {
    return error("incomplete sender's or receiver's
     information");
  }

  lessThanBalanceProof = db.GetProof(sender); ❶
  libHEOV.ProofVerify(lessThanBalanceProof, amount);

  sBalance = db.GetState(sender)
  rBalance = db.GetState(receiver)
  sBalance = libHEOV.Sub(sBalance, amount) ❷
  rBalance = libHEOV.Add(rBalance, amount)
  db.SetState(sender, sBalance)
  db.SetState(receiver, rBalance)

  senderProof = libHEOV.ProofGen(proveKey,
     lessThanBalanceCS, sBalance)
  receiverProof = libHEOV.ProofGen(proveKey,
     lessThanBalanceCS, rBalance)
  db.SetProof(sender, senderProof) ❸
  db.SetProof(sender, receiverProof)
}
```

Listing 2: Transfer Smart Contract (Rewritten)

Listing 1 and Listing 2 depict an example smart contract before and after rewriting, respectively. Both versions of the contract aim to transfer *amount* tokens from the *sender* to the *receiver*. Compared to listing 1. there are three notable distinctions in listing 2.

Firstly, HEOV uses NIZKP to compare two ciphertexts instead of direct comparison, as mentioned earlier. Before the token transfer takes place, the *transfer* smart contract checks whether the balance of the *sender* is sufficient to be deducted (❶). This is accomplished by retrieving the *sender*'s proof from the executor's database and verifying its validity.

Secondly, *transfer* performs HE computation (i.e., *Add* and *Sub*) on ciphertexts (❷). The executors evaluating this transaction do not have access to the actual values, ensuring data confidentiality.

Lastly, *transfer* generates new NIZKP for later reference (❸), ensuring that future executions always use the latest values.

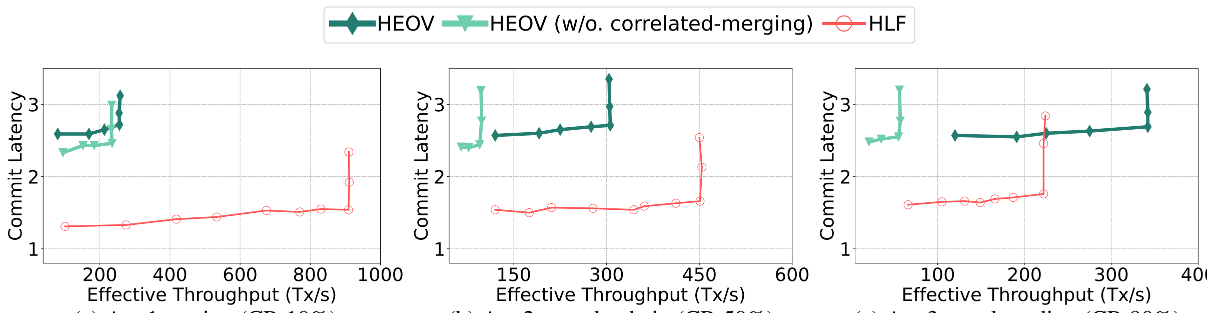
8. Evaluation

We ran all experiments in a cluster with 20 machines, each equipped with a 2.60GHz E5-2690 CPU, 64GB memory, and a 40Gbps NIC. Each node, optimizer, and client was run in a separate docker container. All participant containers were orchestrated by a multi-host Docker Swarm network. The average node-to-node RTT was about 0.2ms.

Baselines. We compared HEOV with two baselines: HEOV (w/o. correlated-merging) and HLF [5]. HEOV (w/o. correlated-merging) can be seen as a strawman approach in the evaluation, as it is essentially HEOV without adopting the correlated-merged protocol. HLF is widely regarded as the most popular permissioned blockchain framework in both industry and academia, serving as a common baseline for HEOV and many other permissioned blockchains. [41, 24, 42]. While the comparison between HEOV and HLF presented the practicality of HEOV, the comparison between HEOV and HEOV (w/o. correlated-merging) showcased the advantages of the correlated-merged protocol.

Workloads. We evaluated all three blockchains (HEOV, HEOV (w/o. correlated-merging), and HLF) using 2 workloads.

The first workload is *SmallBank* [27], which is a popular benchmark for evaluating blockchain systems. SmallBank simulates typical bank business logic and represents a common usage scenario for blockchain systems, making it an ideal workload that is widely used in notable blockchain studies [41, 43]. We implemented two versions of the SmallBank workload: one using encryption and one without encryption. For HLF, SmallBank's computation was done in the plaintext field, without any homomorphic encryption (HE) operations involved. For HEOV and HEOV (w/o. correlated-merging), we created a homomorphic version of the original plaintext SmallBank using libHEOV APIs. (§7.1). At the beginning of each experiment, we created 10 organizations and 100 accounts for each



(a) App1: voting (CR:10%) (b) App2: supply chain (CR:50%) (c) App3: stock trading (CR:90%)

Figure 4: End-to-end performance of HEOV and HLF under applications with different conflict ratios.

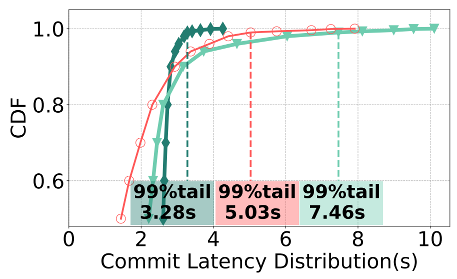


Figure 5: Commit latency under CR=50% (App2).

organization, with each account initialized with the same balance. We then generated a set of *transfer* transactions where a randomly selected account $Acct_1$ transferred a certain amount of tokens to another randomly selected account $Acct_2$. Note that $Acct_1$ and $Acct_2$ may or may not belong to the same organization. The SmallBank workload was used to measure both the end-to-end performance (§8.1) and the three systems' performance under attack (§8.2).

Our second workload is a microbenchmark designed to showcase the performance improvement introduced by the correlated-merged protocol. It consists of two smart contracts, SC_A and SC_M, which respectively execute homomorphic additive and multiplicative transactions. The ratio of these two types of transactions can be configured to simulate various scenarios. This workload was evaluated exclusively on HEOV and HEOV (w/o. correlated-merging) (§8.3), as HLF does not support on-ciphertext computation and is therefore not relevant for this evaluation.

Metrics. We evaluated two types of metrics: *effective throughput* and *commit latency* (also called *end-to-end latency*). The effective throughput is the average number of valid client transactions committed per second, excluding any conflicting aborted transactions. The commit latency is the duration between when a client submits a transaction and when it is committed. Note that if a transaction fails to commit, HEOV's optimizer will resubmit the transaction until it is finally committed. We also reported the 99th percentile commit latency and the cumulative distribution function (CDF) of the transactions' commit latency.

Evaluation methodology. We developed a distributed benchmark using Tape [28], an efficient benchmark tool for HLF. Our benchmark spawned 128 clients across multiple servers to ensure that the benchmarking tool is not a bottleneck.

Each system ran with five executors and four orderers, using BFT-SMaRT [9] as the consensus protocol. The default block size was 100 transactions, which is a common setting used in notable studies [42, 43]. We designated 1% of accounts as *Hot Accounts* and defined the *Conflict Ratio* as the probability of each transaction accessing the hot accounts, with a default value of 50%. We conducted ten runs for each evaluation and reported the average of each metric. Our evaluation focused on three main questions.

§8.1 How efficient is HEOV's correlated-merging?

§8.2 How robust is HEOV's performance to targeted conflicting attacks and no-resubmission attacks?

§8.3 How efficient is HEOV under diverse applications?

8.1. End-to-End Performance

To evaluate the effectiveness of HEOV, we first conducted experiments in benign environments where the network was stable and all participants behaved correctly. We evaluated all systems with different conflict ratios of 10%, 50%, and 90%, respectively.

HEOV outperformed the HEOV (w/o. correlated-merging) in effective throughput, with only a slightly higher commit latency as the trade-off. As shown in figure 4, HEOV recorded a throughput of 255 TPS, 307 TPS, and 343 TPS when the conflict ratio was 10%, 50%, and 90%, respectively. In contrast, HEOV (w/o. correlated-merging) achieved only 235 TPS, 95 TPS, and 55 TPS, demonstrating a larger performance gap compared to HEOV with more conflicting transactions. The average end-to-end latency of HEOV and HEOV (w/o. correlated-merging) were approximately 2.7s and 2.5s, respectively. The slightly higher commit latency observed with HEOV was due to the pending time for merging transactions introduced by the correlated-merged protocol, as discussed in §4.2. It's worth noting that the extra latency overhead of HEOV was merely 8%, but the gain of throughput could be up to 5.2×.

HEOV is particularly suitable for applications with conflicting transactions, thanks to our correlated-merged proto-

System	Average Latency (s)					99% tail
	P	E	O	V	E2E	latency (s)
HEOV	0.75	0.85	0.89	0.23	2.72	3.28
HEOV (w/o. correlated-merging)	0.47	0.87	0.88	0.24	2.46	7.86
HLF	0.03	0.41	0.91	0.19	1.54	5.03

Table 2: Latency of systems under test in figure 4b and 5. The "P" phase represents the preparation period before execution.

col (§4.2). Surprisingly, in figure 8, HEOV significantly outperformed HLF with 1.54x higher throughput with 90% conflict ratio, even with the extra overhead caused by homomorphic encryption and NIZKP generation. In less conflicting scenarios, HEOV was not as performant as HLF. This is the inevitable price HEOV has to pay for ensuring data confidentiality. Nonetheless, HEOV still achieved a higher throughput than HEOV (w/o. correlated-merging) in all three settings.

The performance of HEOV is not undermined as the conflict ratio increases, as shown in Figure 4. Surprisingly, HEOV exhibited an increasing trend in throughput as the conflict ratio increased, performing even better in highly conflicting scenarios. In contrast, both HEOV (w/o. correlated-merging) and HLF suffered from a significant drop in throughput. This can be attributed to HEOV's correlated-merged protocol, which eliminates a substantial portion of potential conflicts. For example, two consecutive *Transfer* transactions from Alice to Bob can be merged into one transaction, reducing the number of HE and NIZKP operations and the number of re-submissions caused by conflicting aborts. However, HLF and HEOV (w/o. correlated-merging) are not equipped with countermeasures for this case, and only allows one valid transaction when multiple transactions in a block modify the same key. Therefore, HLF and HEOV (w/o. correlated-merging) would face performance degradation as the conflict ratio increased, while HEOV's throughput remained high due to its ability to handle conflicts efficiently.

Figure 5 shows that HEOV's average latency (2.7s) was not as competitive as that of HLF (1.6s), due to the extra overhead introduced by the HE and NIZKP operations. However, this higher latency is fully justified by HEOV's confidentiality guarantee for general smart contracts. Additionally, it is worth noting that HEOV achieved an even shorter 99%-tail latency than HLF, and all its transaction latencies were concentrated within a narrow range. This suggests that the correlated-merged protocol adopted by HEOV can generally reduce the number of transaction re-submission.

Table 2 provides a more detailed insight into the latency of each phase. Among the four phases (preparation **P**, execution **E**, ordering **O**, and validation **V**), we observed that the extra overhead of HEOV was mainly concentrated on the preparation and execution phases. In the preparation phase, a transaction needs to be encrypted into ciphertext and may need to wait for merging. In the execution phase, there are several expensive operations, such as HE evaluation, NIZKP generation, and verification. The validation phase also witnesses a slight increase in latency due to the need to verify NIZKP. However, the ordering phase shows no variance in latency, which is expected given that HEOV does not modify this phase. This leaves the performance of the ordering phase unchanged. Overall, the results in Table 2 highlight the trade-off between the confidentiality guarantee provided by HEOV and the extra overhead introduced by the use of FHE and NIZKP operations.

Overall, HEOV achieves high performance while providing a strong security guarantee, making it particularly suitable for applications that are sensitive about data privacy and may involve conflicting transactions.

8.2. Robustness to Attacks

Targeted conflicting attacks (§5.1). We conducted the targeted conflicting attacks on all three systems, as shown in Figure 6. The experiments involved four benign organizations (O_B) and one malicious organization (O_M). A victim organization (O_V) was selected from O_B, and O_M created conflicting transactions to specifically attack the transactions submitted by O_V. To evaluate the behavior of each system under real-world attacks, we divided each experiment into three periods: (*Period I: pre-attack*), during which the systems were running in benign environments; (*Period II: attack*), during which O_M conducted the attack; and (*Period III: post-attack*), during which the attack was over and the environment resumed its benign state.

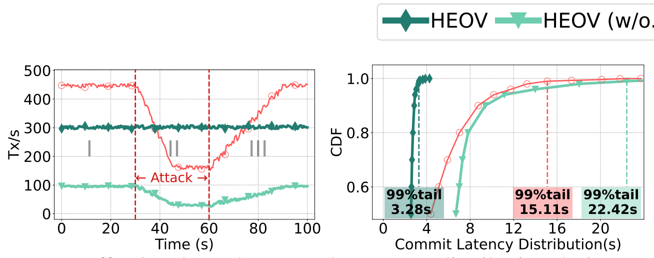
Figure 6a shows the real-time throughput of HEOV throughout the targeted conflicting attack experiments. The results demonstrate that HEOV's throughput was unaffected by the attack. This is because HEOV's correlated-merged protocol can eliminate conflicting aborts (§4.2). In contrast, HLF and HEOV (w/o. correlated-merging) experienced significant drop in throughput during *Period II*. This matches our expectations, as neither HLF nor HEOV (w/o. correlated-merging) are equipped with countermeasures for targeted conflicting attacks.

Figure 6b shows the per-transaction latency distribution of the three systems during *Period II* under the targeted conflicting attacks. HEOV exhibited notably shorter average latency and 99%-tail latency, which were almost the same as when no attacks were conducted. In contrast, the latency of HLF or HEOV (w/o. correlated-merging) was high. The transaction resubmissions caused by aborts seriously prolonged the duration between a transaction's first proposal and its final commitment, resulting in much higher latency compared to HEOV.

In summary, the correlated-merged protocol of HEOV is not only an effective measure for improving throughput, but also an important countermeasure for mitigating targeted conflicting attacks.

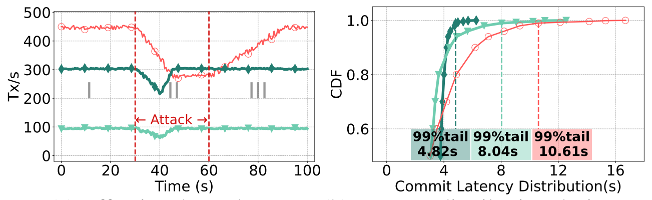
No-resubmission attacks (§5.2). To measure the performance of the three system under no-resubmission attacks, we set up experiments similar to the targeted conflicting attacks experiments (§ 5.1). The metrics are demonstrated in Figure 7.

Figure 7a shows that all three systems experienced different



(a) Effective throughput (b) Latency distribution during targeted conflicting attack

Figure 6: Performance under targeted conflicting attack



(a) Effective throughput (b) Latency distribution during no-resubmission attack

Figure 7: Performance under no-resubmission attack

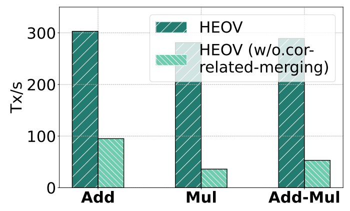


Figure 8: HEOV's optimization improvement under different combinations of HE computation.

levels of performance degradation at the beginning of the no-resubmission attack. However, HEOV's throughput quickly resumed to the level of *Period I*, while HLF's throughput remained at a low level. This demonstrates the effectiveness of the counter-based countermeasure for no-resubmission attacks, where the executors of the benign organizations O_B block malicious clients from O_M. The countermeasure helps to prevent malicious clients from monopolizing the blockchain system and ensures fair access to the system for all clients.

In Figure 7b, both HEOV and HEOV (w/o. correlated-merging) exhibited much better average latency than HLF during the no-resubmission attack. This is because they were able to detect the malicious attackers and reject executing their transactions, thus reserving the computation power for the benign clients. This helped to mitigate the impact of the attack on the system's latency.

Overall, the results in Figure 7 highlight that HEOV is particularly suitable for safety-critical applications that may involve malicious participants. HEOV incorporates countermeasures to mitigate both targeted conflicting attacks and no-resubmission attacks, ensuring fair access to the system and reducing latency.

8.3. Performance under Diverse Applications

The *SmallBank* workload is a representative of a broad spectrum of real-world applications, such as token transfers. However, it only performs homomorphic addition and does not perform any HE multiplication. Therefore, to fully benchmark the performance improvement brought by correlated-merged on addition and multiplication, we developed a workload that comprised both additive homomorphic SC_A and multiplicative homomorphic SC_M smart contracts. SC_A adds the value of two designated accounts (i.e., acc_1 and acc_2, identified through the input parameters), and then writes the result back to acc_1. SC_M multiplies the value of two accounts and writes the result back to acc_1.

We conducted three experiments to evaluate the performance of HEOV and HEOV (w/o. correlated-merging): (1) (**Add**) all transactions only invoked SC_A; (2) (**Mul**) all transactions only invoked SC_M; and (3) (**Add-Mul**) 50% of transactions invoked SC_A, while the other 50% invoked SC_M. As shown in Figure 8, the results demonstrate that HEOV's correlated-merging protocol can support both addition and multiplication operations. This makes HEOV an ideal solution for a variety of real-world applications that require one or both of these types of operations, such as cryptocurrency [18] and finance [49].

Figure 8 demonstrates that HEOV significantly outperformed HEOV (w/o. correlated-merging) in all three groups of experiments. We recorded throughput enhancements of 3.2x, 5.4x, and 7.8x for addition-only, multiplication-only, and addition-multiplication-hybrid applications, respectively. These results highlight the effectiveness of HEOV's correlated-merged protocol on both addition and multiplication operations, particularly when both types of operations are involved. Furthermore, the results suggest that the correlated-merged protocol is especially effective when multiplication constitutes the major HE computation of the business logic.

In short, HEOV's correlated-merged protocol has delivered substantial performance advantages to HEOV without changing the transaction semantics. This breakthrough enables HEOV to safeguard the privacy of sensitive user data with cryptographic guarantees while still achieving high performance.

8.4. Lessons Learned

HEOV has two limitations. First, HEOV is specifically designed for blockchain applications with conflicting transactions, such as token transfers [18]. For these applications, HEOV's correlated-merged protocol significantly reduces the ratio of conflicting aborts and minimizes the expensive HE computations. However, for conflict-free applications, HEOV achieves similar performance to the baseline. It is worth noting that conflicts are common in typical real-world blockchain applications (e.g.,), and achieving conflict-free applications needs significant modifications to the application logic, such as rewriting the smart contract using CRDT [36], which is often not possible or practical for popular blockchain applications, such as financial trading [36].

Second, HEOV's current implementation supports only HE computations on bounded unsigned integers due to the FHE scheme it adopts (i.e., BFV [11, 19]). However, integrating other more capable FHE schemes, such as CKKS [14]), would allow HEOV to support HE computation on floating-point numbers easily. This would expand the range of applications that HEOV can support and increase its flexibility and usability for real-world blockchain applications.

9. Conclusion

We present HEOV, the first privacy-preserving EOV permissioned blockchain that efficiently achieves both BFT safety and data confidentiality. HEOV harmonizes the best of homomorphic encryption, non-interactive zero-knowledge proof and the EOV workflow, while circumventing their drawbacks via a novel correlated-merged homomorphic encryption protocol. Extensive evaluation results on typical blockchain applications with diverse computations show that HEOV is general, highly efficient, and secure compared to the baselines. HEOV is open sourced and its code is released on https://github.com/ccs23p123/heov.

References

[1] Case Study:How Walmart brought unprecedented transparency to the food supply chain with Hyperledger Fabric. https://www.hyperledger.org/learn/publications/walmart-case-study, 2022.

[2] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil, 2018.

[3] Farhana Aleen and Nathan Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. *ACM Sigplan Notices*, 44(3):241–252, 2009.

[4] Mohamed Alloghani, Mohammed M Alani, Dhiya Al-Jumeily, Thar Baker, Jamila Mustafina, Abir Hussain, and Ahmed J Aljaaf. A systematic review on the status and progress of homomorphic encryption technologies. *Journal of Information Security and Applications*, 48:102362, 2019.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, New York, NY, USA, 2018. ACM.

[6] Nick Baumann, Samuel Steffen, Benjamin Bichsel, Petar Tsankov, and Martin T. Vechev. zkay v0.2: Practical data privacy for smart contracts, 2020.

[7] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304, 2015.

[8] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 1730 Massachusetts Ave., NW Washington, DCUnited States, 2014. IEEE, IEEE Computer Society.

[9] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[10] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 326–349, New York, NY, USA, 2012. ACM.

[11] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 868–886, Heidelberg, 2012. Springer, Springer Berlin.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[13] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, New York, NY, USA, 1999. ACM.

[14] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437, Hong Kong, China, 2017. Springer, Springer.

[15] consensys. Gnark, 2023.

[16] Cosmos. validators overview, 2023.

[17] Jacob Eberhardt and Stefan Tai. Zokrates-scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091, Halifax, Nova Scotia, Canada, 2018. IEEE, IEEE.

[18] Ethreum. Erc-20 token standard, 2023.

[19] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. https://eprint.iacr.org/2012/144.

[20] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple noninteractive zero knowledge proofs under general assumptions. *SIAM Journal on computing*, 29(1):1–28, 1999.

[21] Caroline Fontaine and Fabien Galand. A survey of homomorphic encryption for nonspecialists. *EURASIP Journal on Information Security*, 2007:1–10, 2007.

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, Portland, OR, USA, 2019. IEEE.

[24] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20 000 transactions per second. *International Journal of Network Management*, 30(5):e2099, 2020.

[25] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[26] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 581–612, Cham, 2017. Springer International Publishing.

[27] H-Store. H-store: Smallbank benchmark, 2013.

[28] Hyperledger-TWGC. Hyperledger-twgc/tape: A simple traffic generator for hyperledger fabric, 2023.

[29] Peter Kacherginsky, 2022.

[30] Hui Kang, Ting Dai, Nerla Jean-Louis, Shu Tao, and Xiaohui Gu. Fabzk: Supporting privacy-preserving, auditable smart contracts in hyperledger fabric. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 543–555, Portland, Oregon, USA, 2019. IEEE, IEEE.

[31] Yang Liu, Debiao He, Mohammad S Obaidat, Neeraj Kumar, Muhammad Khurram Khan, and Kim-Kwang Raymond Choo. Blockchain-based identity management systems: A review. *Journal of network and computer applications*, 166:102731, 2020.

[32] Medicalchain. Medicalchain. https://medicalchain.com, 2022.

[33] Andreas V Meier. The elgamal cryptosystem. In *Joint Advanced Students Seminar*, 2005.

[34] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, pages 6. 64–70, ., 2020. HomomorphicEncryption.org.

[35] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[36] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. Fabriccrdt: A conflict-free replicated datatypes approach to permissioned blockchains. In *Proceedings of the 20th International Middleware Conference*, pages 110–122, 2019.

[37] NCCGroup, 2016.

[38] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[39] Pascal Paillier and David Pointcheval. Efficient public-key cryptosystems provably secure against active adversaries. In Kwok-Yan Lam, Eiji Okamoto, and Chaoping Xing, editors, *Advances in Cryptology - ASIACRYPT'99*, pages 165–179, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[40] B. O. Peng, Yongxin Zhu, Naifeng Jing, Xiaoying Zheng, and Yueying Zhou. Design of a hardware accelerator for zero-knowledge proof in blockchains. In Meikang Qiu, editor, *Smart Computing and Communication*, pages 136–145, Cham, 2021. Springer International Publishing.

[41] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, and Heming Cui. Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 18–34, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 543–557, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: The case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 105–122, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Ravital Solomon, Rick Weber, and Ghada Almashaqbeh. smartfhe: Privacy-preserving smart contracts from fully homomorphic encryption. *Cryptology ePrint Archive*, 2021.

[45] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput BFT for blockchains, 2019.

[46] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 179–197, San Francisco, California, USA, 2022. IEEE.

[47] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. Zkay: Specifying and enforcing data privacy in smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1759–1776, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Samuel Steffen, Benjamin Bichsel, and Martin Vechev. Zapper: Smart contracts with data and identity privacy. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2735–2749, 2022.

[49] taXchain. taxchain provides a faster, better, cheaper way to complete eu tax forms using hyperledger fabric, 2023.

[50] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[51] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1092–1108, San Francisco, CA, USA, 2021. IEEE.

[52] Paul Voigt and Axel Von dem Bussche. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 10(3152676):10–5555, 2017.

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

[54] Kimchai Yeow, Abdullah Gani, Raja Wasim Ahmad, Joel JPC Rodrigues, and Kwangman Ko. Decentralized consensus for edge-centric internet of things: A review, taxonomy, and research issues. *IEEE Access*, 6:1513–1524, 2017.

[55] zcash. What are zk-snarks?, Jun 2022.