Geco: A Confidentiality-Preserving and High-Performance Permissioned Blockchain Framework for General Smart Contracts

Anonymous authors

Abstract

Data confidentiality is essential for safety-critical blockchain applications. A promising approach to achieving confidentiality is using cryptographic primitives like homomorphic encryption to encrypt client data, and enforcing the correct execution of smart contracts through non-interactive zero-knowledge proofs (NIZKPs). However, existing solutions still face significant limitations in supporting general smart contracts. Firstly, these solutions cannot tolerate non-deterministic contracts whose correct execution cannot be directly proven by NIZKPs. Secondly, many of these solutions adopt cryptographic primitives that restrict arithmetic operations, such as partial homomorphic encryption, which only allows either addition or multiplication over ciphertexts.

We present Geco, a confidentiality-preserving and high-performance permissioned blockchain framework for general smart contracts. Geco supports non-deterministic contracts with general cryptographic primitives (e.g., fully homomorphic encryption), allowing arbitrary arithmetic operations. By co-designing with the multi-party trusted execution mechanism of execute-order-validate blockchains (e.g., Hyperledger Fabric), Geco generates contract logic-agnostic NIZKPs, which proves only that quorum parties produce consistent execution results, thereby ensuring the correct execution of contracts while addressing potential non-determinism. Geco detects and merges multiple conflicting transactions into a single one, minimizing the number of conflict aborts of EOV and cryptographic primitive invocations. Theoretical analysis and extensive evaluation on notable contracts demonstrate that Geco achieves the strongest confidentiality guarantee among existing systems and supports general contracts. Compared to four confidentiality-preserving blockchains, Geco achieved up to 7.14× higher effective throughput and the shortest 99% tail latency.

1 Introduction

Blockchains are extensively deployed in both industry and academia. The main reason is their support for smart contracts that enable trusted execution over a tamper-resistant ledger shared among mutually untrusted participants. However, notable blockchains like Ethereum [45] process and store client data in plaintext, raising deep concerns about data confidentiality. Such concerns are especially problematic for applications involving highly sensitive information, such as medical data [25], as they are subject to stringent privacy laws and regulations, such as the General Data Protection Regulation of the European Union [44].

Many previous work has been proposed to achieve data confidentiality on smart contracts, and can be summarized into two approaches. The first is the architecture approach, which designs new blockchain architectures for data confidentiality. However, this approach either introduces extra trust assumptions or is vulnerable to data breach caused by malicious participants. For example, Ekiden [12] uses trusted hardware. Hawk [28] relies on trusted managers, where a malicious manager can disclose clients' sensitive data.

The second is the cryptography approach. This approach employs cryptographic primitives like homomorphic encryption (HE) to encrypt client data without extra trust assumptions and mandates smart contracts to execute directly on ciphertexts [40–42], preventing sensitive data from being exposed to malicious participants. This approach relies on non-interactive zero-knowledge proofs (NIZKPs) to validate the correctness of execution results (in ciphertext), without leaking any information. Specifically, existing work employs a *re-execution method* to generate NIZKPs. In this method, the NIZKP decrypts the transaction input and result, re-executes the transaction using the decrypted input, and verifies the consistency between the decrypted and re-executed results.

However, despite these progresses, existing work of the cryptography approach still faces two intrinsic limitations. Firstly, existing work is incompatible with non-deterministic contracts, whose different executions may produce inconsistent results under the same input and initial state. Non-deterministic contracts are favorable as they can be created in general programming languages like Go and enhance performance via non-deterministic language features like multithreading [35]. However, these contracts are incompatible with NIZKPs generated by the re-execution method, which implies that identical input always produces the same result.

Secondly, existing work of the cryptography approach suffers from poor performance due to the re-execution method. This method poses challenges in achieving efficient NIZKP generation and verification, especially when integrated with general cryptographic primitives such as fully homomorphic encryption (FHE), which allows arbitrary arithmetic computations over ciphertexts. To illustrate, the combination of the popular BFV scheme [10, 18] (a FHE scheme) and the elliptic curve-based NIZKP [14] takes tens of seconds to generate or verify one NIZKP [39]. Even when employing lightweight cryptographic primitives like partial homomorphic encryption (PHE) which restricts arithmetic operations, the NIZKP operations remain cumbersome. For instance, generating a Groth16 [33] NIZKP using 2048-bit keys for the Paillier PHE scheme [34] consumes more than 256 GB memory [40], impractical for commodity desktops available today.

Overall, the re-execution method employed by the existing cryptography approach for generating NIZKPs is the root cause of their inability to support general non-deterministic contracts and their poor end-to-end performance.

In this study, our key insight to address these limitations is that, *we can re-divide the responsibility for ensuring the correct execution of contracts between the blockchain and cryptographic primitives.* Specifically, instead of relying solely on NIZKPs to prove the correct execution of contracts, we integrate NIZKPs with the quorum-based trusted execution mechanism of execute→order→validate (EOV) permissioned blockchains, such as Hyperledger Fabric (HLF) [4]. This mechanism first executes transactions on multiple executor nodes (Figure 1). The execution is considered correct if quorum nodes produce consistent results. EOV eliminates the result inconsistency caused by non-determinism, enabling the support for non-deterministic contracts. By decoupling the correctness proving logic from the contract logic, EOV significantly reduces the correctness proving overhead for contracts involving expensive cryptographic primitives.

This insight leads to GECO[1], a confidentiality-preserving and high-performance permissioned blockchain framework for general smart contract applications. GECO carries a novel Confidentiality-preserving Execute-Order-Validate (CEOV) workflow for provably correct execution of general smart contracts involving ciphertexts. CEOV executes transactions in three steps. Firstly, a client submits a transaction with plaintext input to an executor node, referred to as the *lead executor*. The lead executor employs a designated cryptographic primitive (e.g., a FHE scheme), as specified by the invoked contract, to encrypt the input. Secondly, the lead executor dispatches the transaction to multiple executors, which independently and concurrently execute the transaction over ciphertexts. Thirdly, the lead executor generates NIZKPs to prove that quorum results are consistent. By leveraging the CEOV workflow, GECO effectively preserves data confidentiality, while simultaneously guaranteeing the correctness of executions for general smart contracts.

However, GECO still faces a non-trivial performance challenge due to the transaction conflicting aborts of EOV. In particular, EOV concurrently executes all transactions for high performance and checks conflicts before committing transactions for data consistency (i.e., optimistic concurrency control [4]). As shown in Figure 1, when multiple transactions concurrently modifies the same key-value pair in the blockchain state database, only one transaction can successfully commit (e.g., tx_i); other conflicting transactions must

[1]GECO denotes GEneral COnfidentiality-preserving blockchain framework.

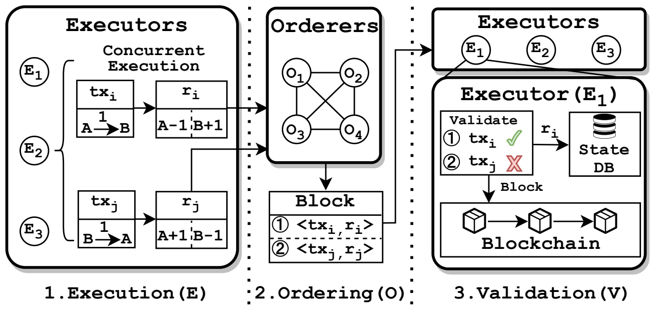


Figure 1. The EOV workflow executes transactions on multiple executors. On the executor E_2, tx_i and tx_j produce results r_i and r_j. For high performance, EOV concurrently executes tx_i and tx_j on each executor, causing conflicting aborts.

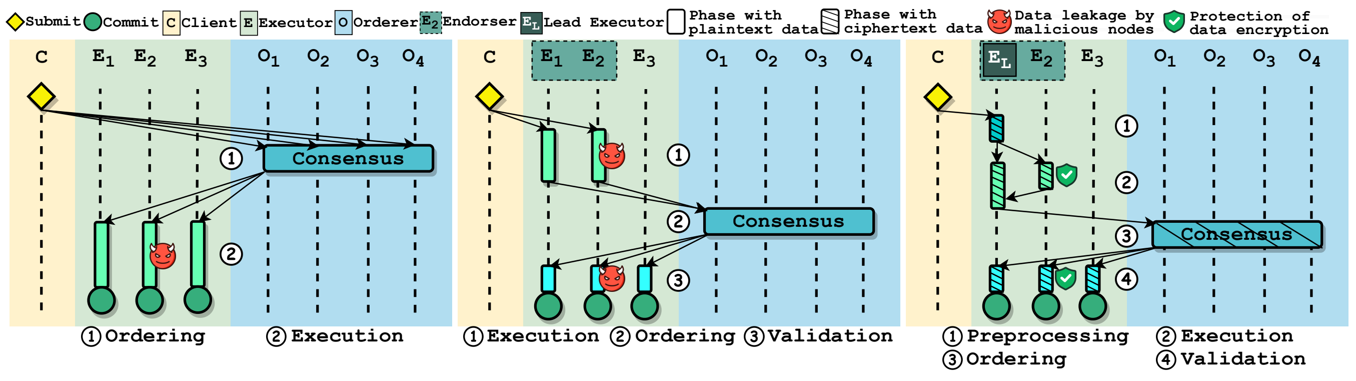
abort (e.g., tx_j). This challenge is further exacerbated in GECO due to the involvement of resource-intensive cryptographic primitives, which incur significant computation overhead. For example, Microsoft SEAL [37], a highly optimized FHE library, takes 2822 milliseconds to execute a single FHE multiplication [31]. Consequently, transaction conflicting aborts causes serious performance degradation on executor nodes.

To tackle the challenge, we propose Correlated Transaction Merging (CTM), a new concurrency control protocol for cryptographic computations. CTM exploits the similarities among conflicting transactions of blockchain applications to tackle the conflicting aborts. Specifically, conflicting transactions invoking the same contract generally modify the same keys. By merging them into a single one, we can commit all transactions with no abort and preserve the semantics of the original transactions. Consider a scenario where two transactions both attempt to perform an addition operation on the same key (e.g., two $X + 1$ operations), CTM merges and commits them into a single transaction (e.g., one $X + 2$ operation).

CTM includes two sub-protocols: *offline analysis* for smart contracts before deployment, and *online scheduling* for transactions during runtime. CTM's offline analysis determines whether multiple conflicting transactions invoking the contract can be merged into a single equivalent transaction. CTM's online scheduling tracks the read and write keys of transactions within each block to identify and merge the mergeable transactions. CTM is especially suitable for contracts that operate on ciphertexts, as it effectively reduces the waste of computing resources caused by conflicting aborts and the invocations of expensive cryptographic primitives.

We implemented GECO on top of HLF [4], a prominent EOV permissioned blockchain framework. We integrated GECO with Lattigo [31], a performant FHE library, and gnark [13], a popular NIZKP library. We evaluated GECO using the SmallBank workload [24] under different conflict ratios, and ten diverse blockchain applications, including both deterministic and non-deterministic contracts. We compared GECO with HLF and ZeeStar [40]. Our evaluation shows that:

- GECO is efficient (§6.1). GECO achieved up to 7.14× higher


Submit
Commit
Client
Executor
Orderer
Endorser
Lead Executor
Phase with plaintext data
Phase with ciphertext data
Data leakage by malicious nodes
Protection of data encryption
C
E1 E2 E3
O1 O2 O3 O4
Consensus
① Ordering ② Execution
① Execution ② Ordering ③ Validation
EL E2 E3
① Preprocessing ② Execution
③ Ordering ④ Validation

(a) Order-Execute **(b)** Execute-Order-Validate **(c)** Confidentiality-preserving Execute-Order-Validate

Figure 2. Our CEOV workflow (Figure 2c) and two existing typical permissioned blockchain workflows.

effective throughput and 14% lower end-to-end latency than ZeeStar. While GECO had a 32% drop in throughput compared to HLF, HLF does not preserve confidentiality.

- GECO is secure (§6.2). GECO can maintain high effective throughput and low end-to-end latency under attacks from malicious participants.
- GECO is general (§6.3). GECO tolerates non-deterministic contracts and supports cryptographic primitives like FHE which allows arbitrary computations over ciphertexts.

Our main contributions are CEOV, a new confidentiality-preserving blockchain workflow tailored for general smart contracts, and CTM, a new concurrency control protocol for transactions involving ciphertexts. CEOV addresses the challenge of ensuring the provably correct execution of non-deterministic contracts incorporating general cryptographic primitives (e.g., FHE), rendering GECO more secure than existing cryptography-based confidentiality-preserving blockchains. CTM enhances the effective throughput and reduces the average end-to-end latency of GECO by minimizing the conflicting aborts through the merging of conflicting transactions. Overall, GECO can benefit blockchain applications that desire both data confidentiality and high performance, such as supply chain [16] and healthcare [29]. GECO can also attract traditional non-deterministic applications, developed with general programming languages like Go and Java, to be deployed upon. GECO's code is available at https://github.com/sosp24geco/sosp24geco.

2 Background

2.1 EOV Permissioned Blockchain

A blockchain is a distributed ledger that records transactions across mutually untrusted nodes. It is either *permissionless* or *permissioned*. Permissionless blockchains (e.g., Bitcoin [32] and Ethereum [45]) are open to anyone, and their participants are mutually untrusted. In contrast, permissioned blockchains (e.g., HLF [4]) are maintained by a group of explicitly identified organizations, and only grant access to explicitly authenticated participants which are trusted within their organizations. Thanks to explicit identity authentication, permissioned blockchains can utilize fast BFT consensus protocols like HotStuff [46] to tolerate malicious nodes.

Permissioned blockchains can be further classified into two types based on their workflows: *order⟶execute* (OE) and *execute⟶order⟶validate* (EOV). As depicted in Figure 2a, the OE workflow has two phases. Firstly, the orderers establish a globally consistent transaction order (**O**); Next, all executors execute the transactions in the predetermined order (**E**). As each executors executes all transactions independently and sequentially, this workflow generally has poor performance and cannot tolerate non-deterministic transactions.

The EOV workflow (Figure 2b) incorporates a **quorum-based trusted execution mechanism** to achieve high performance and tolerate non-deterministic transactions. This mechanism decouples the trust model of applications from the trust model of blockchains. An application can define its own trust assumptions, which are conveyed through the endorsement policy and are independent of those of the blockchain's consensus protocol. Specifically, the application designates a group of executors to execute each transaction. A transaction's execution is deemed correct iff a quorum of these executors produce consistent execution results for it.

The EOV workflow has three phases. Firstly, a group of executors called *endorsers* are selected to concurrently execute each transaction (**E**). The client then collects the results of each transaction from different endorsers to check their consistency; if the execution results are consistent, the execution is deemed correct. The client sends the results to the orderers. Secondly, the orderers determine the transaction order (**O**). Finally, each executor independently validates transaction results via multi-version concurrency control (MVCC) to prevent conflicting data access within a single block (**V**).

2.2 Homomorphic Encryption

GECO encrypts sensitive client data using homomorphic encryption (HE), which allows direct computation over ciphertexts without decryption. A HE scheme has four algorithms: *KeyGen* generates key pairs necessary for other operations;

Enc(m, pk, r) uses the public key pk to encrypt a plaintext m into a ciphertext c with randomness r; *Dec(c, sk)* uses the private key sk to decrypt ciphertext c back to the original plaintext m; $\oplus$ is a binary operator taking two ciphertexts as input and produces a result ciphertext. For instance, in an additive HE scheme, the $\oplus$ operator satisfies the following property: $Enc(m_1, pk, r_1) \oplus Enc(m_2, pk, r_2) = Enc(m_1 + m_2, pk, r_3)$.

HE schemes are classified into two main types based on the supported operators: partial homomorphic encryption (PHE) and fully homomorphic encryption (FHE). Although PHE schemes have relatively low computation overhead, they are limited to only one type of arithmetic operation, either addition or multiplication. This restriction compromises the generality of smart contracts, as discussed in §1. In contrast, FHE schemes are more computation-intensive but allow for arbitrary combinations of these operations, thus capable of preserving the generality of smart contracts. Furthermore, substantial improvements have been made to improve the performance of FHE schemes since the proposal of the first plausible FHE scheme [20], making FHE a promising and practical solution for supporting general smart contracts.

2.3 Non-Interactive Zero-Knowledge Proofs

Geco co-designs the non-interactive zero-knowledge proof (NIZKP) [19, 22] with the EOV workflow to ensure the correctness of transaction execution. NIZKP is a cryptographic primitive that enables a prover P to prove knowledge of a private input s to a verifier V without revealing s. Once P generates a NIZKP using a proving key, V can verify the proof using the respective verifying key without P being present. Formally, given a proof circuit ϕ, a private input s, and some public input x, the prover P can generate a NIZKP to prove knowledge of s satisfying the predicate $\phi(s; x)$. A popular type of NIZKP is zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) [8, 23], which allows any arithmetic circuit ϕ and guarantees constant-cost proof verification in the size of ϕ, making it suitable for blockchain applications [5, 15, 41]. Geco uses the gnark [13] implementation of the Groth16 [33] system (a zkSNARKs construction) with advantages including constant proof size (several kilobytes) and short verification time (tens of milliseconds).

2.4 Related Work

We now discuss previous work on confidentiality-preserving blockchains, as illustrated in Table 1.

Architecture approach. Several previous work attempt to ensure data confidentiality by adopting specialized blockchain architectures without utilizing cryptography technology. Ekiden [12] demands hardware that supports trusted execution environments to execute transactions over private data. Hawk [28] and Arbitrum [26] delegate the contract execution to trusted managers that can peek clients' inputs. Caper [2] requires each participant party to maintain only a partial view of the global ledger and prohibits the party from owning copies of other parties' data. Qanaat [3] adopts a hierarchical data model consisting of a set of data collections that store transaction data and are accessible only to authenticated parties. However, these systems process all data in plaintext, exposing a significant attack surface for corrupted participants seeking to steal confidential data.

System	Data Encryption	Multiple Parties	General Contract	High Performance
✧ Ekiden [12]	✓	✓	✗	✓
✧ Hawk [28]	✓	✓	✗	✓
✧ Arbitrum [26]	✗	✓	✗	✓
✧ Qanaat [3]	✗	✓	✓	✓
✧ Caper [2]	✗	✓	✓	✓
♦ ZeeStar [40]	✓	✤	✗	✗
♦ SmartFHE [39]	✓	✓	✗	✗
♦ Zapper [42]	✓	✗	✗	✗
♦ FabZK [27]	✓	✗	✗	✗
♦ **Geco**	✓	✓	✓	✓

Table 1. Comparison of Geco and related confidentiality-preserving blockchains. "✧ / ♦" means that the system either takes the architecture approach (✧) or the cryptography approach (♦). "✤" means that ZeeStar supports only addition but not multiplication over ciphertexts of different parties.

Cryptography approach. Much previous work uses HE to protect data confidentiality and NIZKP to enforce the execution correctness of smart contracts.

ZeeStar [40] uses exponential ElGamal encryption [30], an additive PHE scheme. It supports ciphertext multiplication by repeating additions. However, this extension is inefficient and does not support multiplication over foreign values (i.e., ciphertexts encrypted by different parties). Moreover, ZeeStar cannot tolerate non-deterministic contracts due to its reliance on the OE workflow, as discussed in §2.1.

SmartFHE [39] is Ethereum-based and uses FHE schemes. It cannot tolerate non-deterministic contracts since it relies on the OE workflow. It does not support contracts involving foreign values as this requires a multi-key variant of SmartFHE that is currently impractical as per the authors.

Zapper [42] uses an oblivious Merkle tree construction and a NIZK processor for data confidentiality. However, Zapper shares SmartFHE's limitations: it supports neither non-deterministic contracts nor computation on foreign values.

FabZK is an HLF-based blockchain that adopts a specialized tabular ledger data structure to obfuscate the transaction-related organizations. However, this data structure significantly restricts the compatible blockchain applications, making FabZK unable to support general contracts.

3 Overview

3.1 System Model

Same as existing EOV permissioned blockchains, Geco has three types of participants: *client*, *executor*, and *orderer*. The latter two are referred to as *nodes*. In Geco, participants are grouped into *organizations*. Each organization runs multiple executors and orderers, and possesses a set of clients. We de-

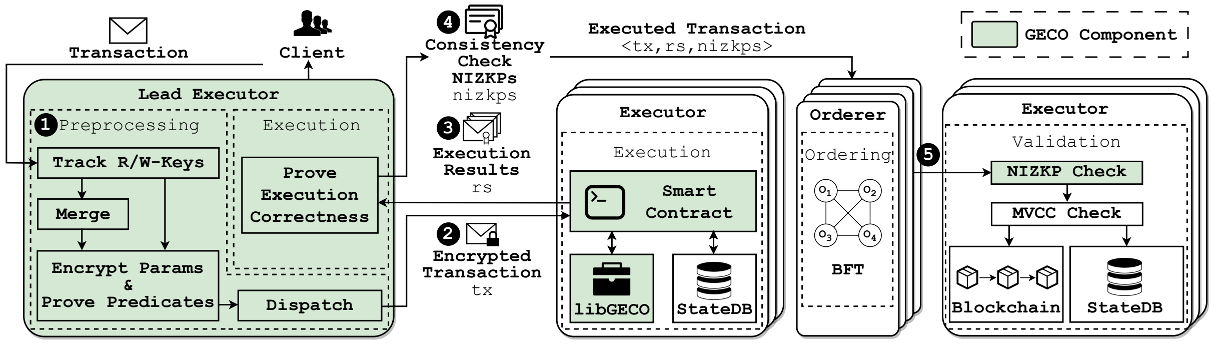


Figure 3. GECO's runtime workflow. GECO components are highlighted in green.

No.	API	Description
		LEAD EXECUTOR APIs
1	`Preprocess(tx)`	Run the runtime protocol (§4.2) to preprocess the given transaction.
2	`ProveC(rs)`	Generate NIZKPs to prove that quorum results are consistent.
3	`VerifyC(rs,nizkps)`	Verify NIZKPs to ensure that quorum results are consistent.
		SMART CONTRACT APIs
4	`Analyze(C)`	Run the offline analysis (§4.1) on the contract.
5	`Require(predicate)`	Abort the transaction if the predicate is false.

Table 2. libGECO APIs.

Data type	Description
`id`	Represent a unique identifier for a client.
`euint`	Represent an unsigned integer plaintext data, storing different ciphertexts encrypted for different clients.
Function	**Description**
`HasID(id)`	Check the existence of the given client `id`
`GetID()`	Get the `id` of the client that submits the transaction.
`GetState(key)`	Read a key-value pair from the state database.
`PutState(key,val)`	Write a key-value pair to the state database.
`Add(ct`$_1$`,ct`$_2$`)`	Perform FHE addition on the given ciphertexts.
`Sub(ct`$_1$`,ct`$_2$`)`	Perform FHE subtraction on the given ciphertexts.

Table 3. Data types and functions used in Listing 1.

scribe the functionalities of each participant type as follows:
Clients. Only clients can submit transactions to nodes for execution and commitment.
Executors. Each executor is responsible for three main tasks: executing transactions, validating results, and maintaining an up-to-date local copy of the blockchain state database.

Each organization has a designated *lead executor*, as shown in Figure 3. In addition to the aforementioned tasks, lead executors are assigned the following additional tasks:

- Detect and merge multiple conflicting transactions.
- Encrypt transaction inputs with the cryptographic primitive specified by the invoking smart contract.
- Dispatch transactions to other executors and orderers for execution and ordering, respectively.
- Generate NIZKPs to prove the correctness of execution by checking the consistency of transaction results.

Orderers. GECO orderers determine the order of transactions in blocks via a consensus protocol (e.g., PBFT [11]).

3.2 Threat model

GECO adopts the Byzantine failure model [6, 11], where orderers run a BFT consensus protocol that tolerates up to $\lfloor \frac{N-1}{3} \rfloor$ malicious orderers out of N orderers. Same as existing EOV permissioned blockchains like HLF, GECO participants trust all participants within the same organization but no participants from other organizations. We make standard assumptions on cryptographic primitives, including FHE and NIZKP.

3.3 libGECO

GECO provides a library named libGECO with five APIs (see Table 2). The lead executor APIs assist lead executors in processing transactions and ensuring the execution correctness. The smart contract APIs enable contract developers to analyze the mergeability (Definition 4.2) of the contract and enforce the validity of the given predicates involving ciphertexts. Note that the `Require()` API follows the idea of the *require* statement of ZeeStar [40]. We will further discuss how to integrate libGECO into GECO's protocol in §3.5 and §4.2.

3.4 Example Smart Contract

```
func Transfer(id src, id dst, euint val) {
  if (!HasID(src) or !HasID(dst)) {Abort();}
  if (src != GetID()) {Abort();}
  euint srcBalance = GetState(src);
  Require(srcBalance >= val);      // Invoke API 5
  euint dstBalance = GetState(dst);
  PutState(src, Sub(srcBalance, val));
  PutState(dst, Add(dstBalance, val));
}
```

Listing 1. An example contract for confidential token transfer. We denote a transaction (invoking the contract) that transfers one token from account A to B as $A \xrightarrow{1} B$.

Listing 1 is the pseudocode of a token transfer contract, which confidentially transfers `val` tokens from the sender client `src` to the receiver client `dst`. Table 3 introduces the data types and functions used in Listing 1. Note that the FHE functions `Add` and `Sub` can be implemented with any FHE scheme that allows addition and subtraction on ciphertexts, such as the BFV scheme [10, 18]. In this contract, the `Require()` API (Table 2) verifies a NIZKP that decrypts the ciphertexts involved (i.e., `srcBalance` and `val`), and checks the predicate with the plaintexts. This NIZKP is generated by the lead executor in the preprocessing phase (§4.2) and is attached to the transaction. Intuitively, this contract ensures that each client's balance is kept confidential and only accessible to the respective client, while the number of tokens being transferred is known solely to the sender and the receiver.

3.5 Geco's Protocol Overview

Geco contains two sub-protocols: (1) *offline contract analysis* and (2) *runtime transaction schedule*.

Offline contract analysis (§4.1). Before contract deployment, the developer invokes the `Analyze()` API on the contract. This API performs the offline analysis of Geco's Ctm protocol to determine whether the contract is *mergeable* (Definition 4.2). For example, the contract in Listing 1 is *additively mergeable*: multiple conflicting transactions invoking this contract with identical `src` and `dst` ($tx_1 : A \xrightarrow{1} B$ and $tx_2 : A \xrightarrow{1} B$) can be merged into one transaction ($tx_{1,2} : A \xrightarrow{2} B$) by summing their `val`, while keeping `src` and `dst` unchanged.

Runtime transaction schedule (§4.2). Geco incorporates a new Ceov workflow and the Ctm protocol to schedule transaction executions at runtime, as shown in Figure 3.

Phase 1: Preprocessing. The lead executor invokes the `Preprocess()` API to preprocess the transactions submitted by clients. Firstly, the lead executor invokes the Ctm protocol to merge transactions (❶ in Figure 3 [2]): the lead executor collects transactions submitted by clients from the same organization (§3.2), merges conflicting transactions whose parameters can be merged without changing their semantics, and leaves other transactions unchanged. Next, the lead executor encrypts all transaction's plaintext inputs of the data type `euint` using the encryption keys of corresponding organizations. In addition, for contracts using the `Require()` API, the lead executor generates NIZKPs to prove the predicates. Lastly, the lead executor selects a group of executors, known as *endorsers* in EOV [4], and dispatches the encrypted transactions to the endorsers for execution.

Phase 2: Execution. Upon receiving a transaction from a lead executor (❷), the endorser executes the invoked contract and produces a *read-write set* as the execution result. The endorser then sends the result back to the lead executor.

After receiving results from all endorsers (❸), the lead executors invoke the `ProveC()` API to generate NIZKPs, proving that quorum executors (i.e., a majority of executors) have produced consistent results for keys belonging to the same organization, thereby ensuring execution correctness.

Phase 3: Ordering. The lead executors of different organizations deliver the transactions with the NIZKPs to the Geco orderers for transaction ordering (❹). The orderers run a BFT protocol to achieve consensus on the intra-block order of transactions and disseminate the block to all executors for validation and commitment.

[2]Unless otherwise specified, ❓ refers to annotations in Figure 3.

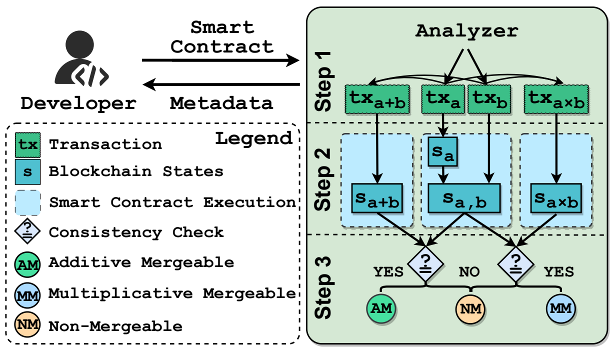


Figure 4. Geco's offline smart contract analysis protocol. tx_{a+b} and $tx_{a\times b}$ are formed by merging tx_a and tx_b via adding or multiplying their input parameters (§4.1). s_{a+b} and $s_{a\times b}$ are the states produced by executing tx_{a+b} and $tx_{a\times b}$. $s_{a,b}$ is the state produced by sequentially executing tx_a and tx_b.

Phase 4: Validation. When receiving a block from orderers (❺), the executor sequentially validates each transaction within the block in the predetermined order. Specifically, the executor invokes the `VerifyC()` API for each transaction to check the consistency of the transaction's quorum results. The executor commits only those transactions that do not conflict with previously committed transactions.

In general, the highlights of Geco arise from achieving data confidentiality and high performance for general smart contracts. We illustrate our highlights in two aspects. Firstly, existing cryptography-based confidentiality-preserving blockchains face fundamental limitations in supporting general non-deterministic contracts or general cryptographic primitives (§1). Geco's Ceov workflow tackles these limitations by leveraging the quorum-based trusted execution of EOV permissioned blockchains to generate lightweight NIZKPs, which are agnostic to the contract logic and only serve to prove the consistency of quorum results, enabling the support for general non-deterministic smart contracts. This also enables Geco to encrypt client data using cryptographic primitives that can perform arbitrary arithmetic computations over ciphertexts, such as FHE.

Secondly, in contrast to existing EOV permissioned blockchains which abort conflicting transactions [5, 40], the Geco's Ceov workflow incorporates the Ctm protocol to merge multiple conflicting transactions into a single transaction, effectively eliminating the conflicting aborts and reducing the invocation of cryptographic primitives without altering the original semantics. Therefore, Ctm leads to a considerable reduction in average end-to-end latency and prevents the wastage of computational resources.

4 Protocol Description

4.1 Offline Smart Contract Analysis

Definition 4.1 (Smart contract parameter type)**.** Smart contract input parameters are classified into two types: *key* and

Function	Description
`GetAllCFPs(C)`	Get all control flow paths of the contract.
`GenRandStateDB(C,p)`	Randomly generate a state database for the given contract and control flow path.
`GenRandParams(C,p)`	Randomly generate transaction parameters for the contract and control flow path.
`GenTX(k,v)`	Generate a transaction with the parameters generated by `GenRandParams()`.
`ExecTX(tx,state)`	Execute a transaction over a state database and return the updated state database.
`GetSize(paths)`	Get the number of control flow paths.
Variable	**Description**
C	The smart contract to be analyzed.
s, s_a, $s_{a,b}$, s_{a+b}, $s_{a\times b}$	States in blockchain database
tx_a, tx_b, tx_{a+b}, $tx_{a\times b}$	Transactions.
paths	The set of all control flow paths.
p	A control flow path.
n_{a+b}	The counter for tracking $s_{a,b} = s_{a+b}$.
$n_{a\times b}$	The counter for tracking $s_{a,b} = s_{a\times b}$.
k	Key parameters for the smart contract.
v_a, v_b	Value parameters for the smart contract.

Table 4. Functions and variables for Algorithm 1.

value. The key parameters specify the contract's accessed keys while the value parameters derive the respective values.

Definition 4.2 (Mergeable smart contract)**.** A smart contract is *mergeable* iff, for any two conflicting transactions tx_1 and tx_2 that invoke the contract with identical key parameters, the transactions can be merged into a single transaction $tx_{1,2}$, whose key parameters are the same as tx_1 and tx_2, while the value parameters are the sums or products of the respective value parameters of tx_1 and tx_2.

A smart contract is *additive mergeable* iff it is mergeable and generates new value parameters by adding the respective value parameters of the conflicting transactions. Likewise, a smart contract is *multiplicative mergeable* iff it is mergeable and generates new value parameters by multiplying the respective value parameters of the conflicting transactions. Otherwise, the contract is *non-mergeable*.

The *offline analysis* determines whether a smart contract is additive mergeable, multiplicative mergeable, or non-mergeable. The `Analyze()` API implements the offline analysis protocol. As Figure 4 shows, GECO analyzes the contract in three steps using random interpretation [1], which generates different inputs and initial states to capture all possible control flow paths (see Algorithm 1 and Table 4). The offline analysis produces a *metadata* as its output, indicating the mergeability of the contract. During runtime, lead executors inspect the metadata and carry out transaction merging solely on additive mergeable and multiplicative mergeable contracts.

Analysis step 1: Preparation. For a contract C, the GECO analyzer first checks whether each parameter is a key or value parameter. Specifically, the analyzer checks all calls to EOV's state access APIs, such as `GetState()` and `PutState()` (see Table 3). A parameter is a key parameter if it is used as a key in any of these APIs; otherwise, it is a value parameter.

Algorithm 1: Offline analysis protocol (§4.1; API 4)

```
1  n_{a+b} ← 0; n_{a×b} ← 0; paths ← GetAllCFPs(C);
2  foreach p in paths do
       // Step 1: Preparation
3      s ← GenRandStateDB(C, p);
4      k, v_a, v_b ← GenRandParams(C, p);
5      tx_a ← GenTX(k, v_a); tx_b ← GenTX(k, v_b);
6      tx_{a+b} ← GenTX(k, v_a + v_b);
7      tx_{a×b} ← GenTX(k, v_a × v_b);
       // Step 2: Execution
8      s_a ← ExecTX(tx_a, s); s_{a,b} ← ExecTX(tx_b, s_a);
9      s_{a+b} ← ExecTX(tx_{a+b}, s); s_{a×b} ← ExecTX(tx_{a×b}, s);
       // Step 3: Consistency check
10     if s_{a,b} = s_{a+b} then
11         n_{a+b} ← n_{a+b} + 1;
12     else if s_{a,b} = s_{a×b} then
13         n_{a×b} ← n_{a×b} + 1;
14 if n_{a+b} = GetSize(paths) then
15     return AdditiveMergeable
16 else if n_{a×b} = GetSize(paths) then
17     return MultiplicativeMergeable
18 else
19     return NonMergeable
```

Next, the analyzer randomly generates an initial blockchain state S, as well as two conflicting transactions tx_a and tx_b that share identical key parameters. In addition, the analyzer attempts to generate two "*merged*" transactions, namely tx_{a+b} and $tx_{a\times b}$, based on tx_a and tx_b. For tx_{a+b}, the analyzer generates new value parameters that are sums of the respective value parameters of tx_a and tx_b, while leaving the key parameters unchanged. Similarly, for $tx_{a\times b}$, the analyzer generates new value parameters that are products of the respective value parameters of tx_a and tx_b.

Analysis step 2: Random interpretation. GECO performs three independent runs of execution for C on the same initial state. In the first run, GECO executes tx_a and produces s_a. Subsequently, GECO executes tx_b on s_a, resulting in the final state $s_{a,b}$. In the second and third runs, GECO separately executes tx_{a+b} and $tx_{a\times b}$, producing s_{a+b} and $s_{a\times b}$, respectively.

Analysis step 3: Consistency check. Lastly, GECO conducts a consistency check among the three resulting states. If the equality $s_{a,b} = s_{a+b}$ holds for all control flow paths, the smart contract C is additive mergeable. Similarly, if $s_{a,b} = s_{a\times b}$, C is multiplicative mergeable. Otherwise, C is non-mergeable.

Assumptions: An analyzable contract must satisfy two requirements. For analysis step 1, the read-write set should be identified before execution, i.e., the read and write keys are explicitly identified in EOV's state access APIs. For analysis step 2, the contract should not contain recursive contract calls [1]. GECO can easily integrate more advanced analysis techniques to support contracts with fewer requirements.

For non-analyzable contracts that do not satisfy these requirements, GECO conservatively regards these contracts as *non-mergeable*. Therefore, the non-analyzable contracts can at most degrade GECO's performance.

Function	Description
`GetRWKeys(tx)`	Get all read and write keys of a transaction.
`DetectConflict(K,Q)`	Return the set of mergeable transactions and the set of non-mergeable transactions.
`Merge(txs)`	Merge the given set of mergeable transactions.
`Encrypt(txs,K)`	Encrypt the parameters of the transactions.
`ProvePredicates(txs)`	Generate NIZKPs to prove predicates if the transaction invokes the `Require()` API.
`Dispatch(txs)`	Dispatch the given transactions to endorsers.
`Execute(tx)`	Execute the transaction and return the result.
`ReplyResult(tx,r)`	Send the results to the lead executor.
`Order(tx,rs,nizkps)`	Send the transaction for ordering.
`CheckMVCC(rs)`	Perform a multi-version concurrency control check on the execution results.
`Commit(tx,rs,nizkps)`	Commit the given transaction.
`AppendBlock(blk)`	Append the given block to the local copy of the blockchain.
Variable	**Description**
`K`	All read and write keys for transactions in `Q`.
`Q`	The transaction buffering queue.
`T`	The threshold number of transactions in `Q`.
$\texttt{txs}_\texttt{m}$	The set of mergeable transactions.
$\texttt{txs}_\texttt{n}$	The set of non-mergeable transactions.
$\texttt{txs}_\texttt{e}$	The set of encrypted transactions.

Table 5. Functions and variables for Algorithm 2.

Algorithm 2: Runtime protocol of the lead executor (§4.2)

```
    // Phase 1: Preprocessing (Invoke API 1)
 1  K ← ∅; Q ← ∅;
 2  Upon reception of Transaction tx from client; do
 3      K ← K ∪ GetRWKeys(tx);
 4      Q ← Q ∪ tx;
 5  Upon timeout or |Q| ≥ T
 6      $txs_m$, $txs_n$ ← DetectConflict(K, Q);
 7      $txs_m$ ← Merge($txs_m$);
 8      $txs_e$ ← Encrypt($txs_m$, K) ∪ Encrypt($txs_n$, K);
 9      ProvePredicates($txs_e$);
10      Dispatch($txs_e$);
    // Phase 2: Execution
11  Upon reception of Transaction tx from lead executor; do
12      r ← Execute(tx);
13      ReplyResult(tx, r);
    // Phase 3: Ordering
14  Upon reception of all Results rs for Transaction tx; do
15      nizkps ← ProveC(rs);                    // Invoke API 2
16      Order(tx, rs, nizkps);
    // Phase 4: Validation
17  Upon reception of Block blk; do
18      For tx, rs, nizkps in blk; do
19          if VerifyC(rs, nizkps)∧             // Invoke API 3
              CheckMVCC(rs) then
20              Commit(tx, rs, nizkps);
21          AppendBlock(blk);
```

4.2 Runtime Transaction Schedule

GECO's *runtime protocol* (Algorithm 2 and Table 5) incorporates our new CEOV workflow to execute transactions in four phases, and enhances the performance with a CTM protocol.

Phase 1: Preprocessing. The lead executor invokes the `Preprocess()` API to merge and encrypt client transactions.

Phase 1.1: Tracking read and write keys. For each block, the lead executor buffers the transactions in a queue and keeps track of each transaction's read and write keys. For example, for a transaction ($tx : A \xrightarrow{1} B$) that invokes the contract in Listing 1, the lead executor tracks that the transaction has two read keys (A and B) and two write keys (A and B).

When a timeout occurs or the number of queuing transactions exceeds a threshold, the lead executor detects conflicting transactions according to their read and write keys. Next, the lead executor runs our CTM protocol (Phase 1.2) to merge conflicting transactions with identical key parameters and encrypts all transactions using FHE (Phase 1.3).

Phase 1.2: Correlated transaction merging (GECO's CTM protocol). For conflicting transactions with identical key parameters, the lead executor merges them by preserving their key parameters and generating new value parameters. In particular, the lead executor generates new value parameters by adding or multiplying the original value parameters of the mergeable transactions that invoke either an additive mergeable or multiplicative mergeable contract, respectively.

Phase 1.3: Encrypting transaction parameters and proving predicates. The lead executor encrypts the transaction input parameters that are of data type `euint`, namely encrypted unsigned integers (see Table 3). The lead executor analyzes the read and write keys associated with the `euint` parameters, then encrypts the parameter plaintexts using the encryption keys of the clients specified by the read and write keys. For example, for transaction $tx : A \xrightarrow{val} B$ (Listing 1), as `val` is computed together with `srcBalance` and `dstBalance`, the `euint` parameter `val` has two write keys A and B, which belongs to org_1 and org_2, respectively. Therefore, the lead executor generates two ciphertexts for `val` by encrypting `val` using org_1 and org_2's public keys, respectively.

The lead executor also generates NIZKPs for contracts that invoke the `Require()` API to validate specific predicates. For example, the contract in Listing 1 requires `srcBalance` to be not less than `val` for a successful token transfer. Thus, the lead executor generates a NIZKP which first decrypts the ciphertext (`srcBalance`) and then proves whether the predicate (`srcBalance > val`) holds. The NIZKPs are attached to the transactions for verification by other participants.

Phase 1.4: Dispatch. The lead executor disseminates transactions to the involved organizations' executors (known as "endorsers") for execution. For instance, the above tx is submitted to executors in org_1 and org_2 for executions.

Phase 2: Execution. The executors execute each transaction in two steps: (1) the endorsers produce execution results, and (2) the lead executor checks the correctness of executions.

Phase 2.1: Producing execution result. Upon receiving a transaction from a lead executor, the endorser invokes the contract specified by the transaction. The execution produces a read-write set as the result, which records the blockchain states that the transaction reads from and writes to the state database. Next, the endorser signs the result and replies it to the lead executor to check the execution correctness.

Phase 2.2: Checking the execution correctness. After collecting results from endorsers of all involved organizations, the lead executor ensures correct executions by proving that quorum executors produced consistent read-write sets. However, the read-write sets contain ciphertext data that cannot be directly compared without decryption as they involve random noise for security reasons [43]. To address this issue, the lead executor invokes the `ProveC()` API to generate a *consistency check NIZKP* that first decrypts the ciphertexts belonging to clients of the same organization, and then checks the consistency of the plaintexts. Note that the consistency check NIZKP takes the decryption key as private input, which protects the confidentiality of the decryption key. In the case where the read-write sets contain ciphertexts of multiple organizations, the lead executors of those organizations all follow the same procedure for proving the result consistency.

For instance, consider a transaction tx that modifies three keys A, B and C, belonging to different organizations, namely org_1, org_2 and org_3. To prove the consistency of tx's results from different endorsers, org_1 provides a consistency check NIZKP that different endorsers produce consistent result for key A. Similarly, org_2 provides a NIZKP that different endorsers produce consistent result for key B. NIZKPs from org_1, org_2, and org_3 are all submitted for ordering, collectively forming the correctness proof of tx.

Note that a transaction's result is considered consistent iff the same quorum of executors produces consistent read-write set results for all keys. If the endorsers of org_1 and org_2 produce consistent results for key A, while the endorsers of org_2 and org_3 produce consistent results for key B, it indicates that the consistent results for different keys are produced by different quorums of endorsers. Consequently, we cannot affirm the consistency of tx's execution results.

Phase 3: Ordering. GECO orderers run a BFT consensus protocol (e.g., PBFT [11]) to reach a consensus on the transaction order within a block. After a block is agreed upon by all orderers, they distribute it to all executors for validation.

Phase 4: Validation. Upon receiving a block from orderers, the executor sequentially validates each transaction. This validation process involves two steps: (1) invoking the `VerifyC()` API to verify the consistency check NIZKPs associated with the transactions, and (2) performing a multi-version concurrency control (MVCC) check on the results. For each valid transaction, the executor commits the result to the blockchain state database. For invalid transactions, the executor does not commit their results (i.e., transaction abort). After the executor has validated all transactions, it permanently appends the block to its local copy of the blockchain.

4.3 Defend Against Malicious Participants

The above protocol ensures data confidentiality even in the presence of malicious participants. Specifically, each GECO participant can only access the plaintext data of clients within the same organization, as participants in the same organizations are mutually trusted (§3.2). For clients from other organizations, the participant can only access their ciphertext data but not the corresponding plaintexts.

Although malicious participants are unable to compromise GECO's confidentiality, they may still significantly degrade the system's performance. In Phase 2.2, a lead executor may fail to submit the consistency check NIZKPs for specific transactions submitted by other organizations, either due to network failures or malicious intent, causing these transactions to be always aborted in Phase 4. The attack is detrimental in GECO because it causes a substantial waste of computation resources on multiple executors, especially when smart contracts involve resource-intensive FHE computations (§1).

To tackle the NIZKP omission attacks, GECO mandates the lead executors to submit the *consistency check NIZKPs* for all transactions, even if the results from different executors are inconsistent. To reduce false positives caused by network failures, GECO executors track the number of NIZKP omissions for each lead executors. A lead executor is deemed malicious only when the number of omissions caused by the lead executor exceeds a configurable threshold (by default, ten omissions). GECO executors proactively reject transactions that involve organizations of malicious lead executors in Phase 1 for a long period (by default, 10000 blocks, after which the omission number for a lead executor is reset to zero), preventing the potential waste of computation resources.

5 Analysis

This section defines and analyzes the three guarantees of GECO: correctness, liveness, and confidentiality.

5.1 Correctness

Definition 5.1 (Correctness)**.** For any agreed block with a serial number s, assuming that all executors start with the same initial blockchain state, once all valid transactions in blocks with serial numbers $s' \leq s$ are committed, all benign executors will converge to the same state (BFT safety). This state equals to the state obtained by sequentially executing all valid transactions on the same initial state (ACID).

Proof. GECO achieves equivalent BFT safety to existing EOV blockchains [4] through two properties of consistency:

- *Block consistency.* GECO treats the consensus protocol as a blackbox, as illustrated in Phase 3 of the runtime protocol (§4.2). Therefore, GECO inherits the mature consistency (safety) guarantee of existing BFT consensus protocols [7] and their implementations. This ensures that all benign executors process and append the same blocks in an identical order determined by the orderers.
- *State consistency.* Every benign executor follows the same deterministic protocol to validate and commit transactions in a given block, as demonstrated in Phase 4 of the runtime protocol (§4.2). A transaction's result is committed to the

No.	Name	⊕	⊗	Description
1	casino ♠	✓	✓	A coin flip game with biased odds: 49% chance to win, 51% chance to lose, both 0.5× stake.
2	convolution	✓	✓	Compute one-dimensional convolution on two vectors with different owners.
3	inner-product	✓	✓	Compute the inner-product of two vectors with different owners.
4	rebate	✓	✓	Currency rebate for qualifying expenditures exceeding a threshold.
5	taxing	✓	✓	Tax calculation and payment.
6	appraisal	✓		Appraise collectibles with confidential value estimates.
7	crowdfunding	✓		Raise funds from multiple owners for a single project.
8	election	✓		Determine the winner in a two-candidate election.
9	med-chain	✓		Medicine inventory management.
10	token-transfer	✓		Peer-to-peer token transfer.

Table 6. Example smart contracts used in the second workload. ⊕ and ⊗ indicate whether the contract uses FHE addition or multiplication, respectively. ♠ indicates that the contract is non-deterministic.

state database *iff* a quorum of the results are consistent and do not modify any key that has been modified by a previous transaction of the same block. This ensures that all benign executors maintain a consistent state and a consistent local copy of the blockchain.

Geco guarantees the ACID properties for transaction execution by inheriting them from the EOV workflow.

- *Atomicity.* Geco either commits or aborts each transaction.
- *Consistency.* Geco exclusively commits valid and consistent transaction results in a sequential order determined by the orderers. This leads to a consistent resulting state across benign executors given an identical initial state.
- *Isolation.* Each transaction operates independently, without interference or conflicts with other transactions.
- *Durability.* For a committed transaction, its changes to the blockchain state are permanent and irreversible.

□

5.2 Liveness

Definition 5.2 (Liveness)**.** Any valid transaction tx submitted by a benign client in the ordering phase will eventually be included in a block by Geco.

Proof. Geco inherits the BFT liveness from existing EOV permissioned blockchains [4]. Specifically, the orderers run a BFT consensus protocol, which guarantees to finalize tx into a block B (§4.2) and deliver the block to all executors. □

5.3 Confidentiality

Definition 5.3 (Confidentiality)**.** For any transaction accessing the ciphertext data ct (with the corresponding plaintext data denoted as pt) belonging to a client of organization org, Geco guarantees that any attacker from a different organization $org*$ is unable to decrypt ct and obtain pt.

Proof. Geco's confidentiality guarantee is based on two inherent confidentiality guarantees separately provided by NIZKPs and the encryption scheme employed. Firstly, the private inputs for generating NIZKPs are impossible to be revealed by verifiers [22]. This enables Geco to ensure the confidentiality of NIZKPs' private inputs such as decryption keys (i.e., private keys), as discussed in §2.3 and §4.2. Secondly, the encryption scheme employed by Geco (e.g., the BFV scheme [10, 18]) achieves *indistinguishability under chosen-plaintext attack* (IND-CPA [9]), meaning that any attacker is computationally infeasible to distinguish the plaintext of a given ciphertext without the corresponding decryption key. Specifically, it is impossible for an attacker from a different organization $org*$ to access the plaintext data corresponding to the ciphertext data belonging to org's clients. This is because org_1 keeps its decryption key confidential, and the IND-CPA makes it computationally infeasible for the attacker to gain any information about the ciphertext without any knowledge of the decryption key. □

6 Evaluation

Geco implementation. We built Geco on the codebase of HLF v2.5 [17]. Geco uses the BFV scheme [10, 18] implemented by Lattigo [31]. With default cryptographic parameters, Geco supports FHE arithmetic computation over 58-bit unsigned integers, satisfying the requirements of diverse contracts listed in Table 6. Geco uses the Groth16 NIZKP system [22] on the BN254 elliptic curve implemented by gnark [13]. Same as existing systems [5, 40] that use Groth16, our implementation requires a circuit-specific trusted setup to generate the verification key for each NIZKP circuit. This setup is a one-time procedure with negligible overhead.

Baselines. We compared Geco with four baselines: HLF [4], ZeeStar [40], Geco (w/o. Ctm), and Geco (w/o. defense). HLF is one of the most popular permissioned blockchain frameworks in both industry and academia [21, 35, 36]. ZeeStar is a notable confidentiality-preserving blockchain that encrypts sensitive data using exponential ElGamal encryption [30] (a PHE scheme) and generates Groth16 NIZKPs to ensure the execution correctness. Geco (w/o. Ctm) implements only the Ceov workflow without integrating the Ctm protocol. Geco (w/o. defense) implements both the Ceov workflow and the Ctm protocol, but lacks the defense mechanism against the NIZKP omission attacks as specified in §4.3.

Workloads. We used two workloads for evaluation. The first one (§6.1 and §6.2) is *SmallBank* [24], a widely used benchmark for evaluating blockchain performance [35, 38]. SmallBank simulates the banking scenario and provides diverse

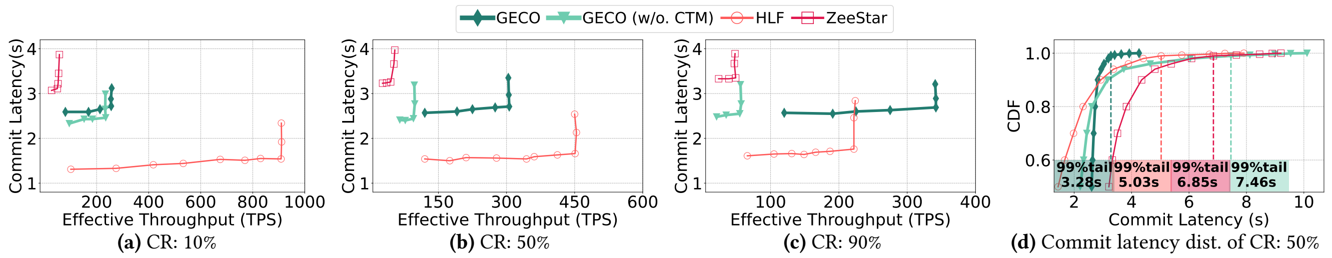


Figure 5. End-to-end performance of Geco and baselines in benign environments with different conflict ratios.

System	Average Latency (s)					99% tail
	P	E	O	V	E2E	latency (s)
ZeeStar	n/a	n/a	n/a	n/a	3.11	6.85
HLF	n/a	0.44	0.91	0.19	1.54	5.03
Geco (w/o. Ctm)	0.47	0.87	0.88	0.24	2.46	7.46
Geco	0.75	0.85	0.89	0.23	2.72	3.28

Table 7. Each phase's latency and the 99% tail latency of Figure 5b. The letters "P", "E", "O", and "V" denote the preprocessing, execution, ordering, and validation phases, respectively. The term "E2E" denotes the end-to-end latency.

contracts like token transfer. We evaluated the encrypted version of SmallBank, where each contract is re-written with APIs provided by libGeco (see Table 2) and gnark. Our second workload (§6.3) consists of ten contracts that involve FHE addition and multiplication, as shown in Table 6. In §6.3, we evaluated Geco, Geco (w/o. Ctm), and ZeeStar with these contracts to measure the performance improvement brought by the Ctm protocol in diverse applications such as gaming (`casino`) and finance (`token-transfer`).

Metrics. We measured two metrics: (1) *effective throughput*, the average number of client transactions per second (TPS) that are committed to the blockchain, excluding aborted transactions; and (2) *commit latency* (known as *end-to-end latency*), measuring the time from client submission to transaction commitment. In addition, we reported the 99th percentile commit latency (tail latency) and reported the cumulative distribution function (CDF) of the commit latency.

Testbed. We ran experiments in a cluster with 20 machines, each with a 2.60GHz E5-2690 CPU, 64GB memory, and a 40Gbps NIC. The average node-to-node RTT was 0.2 ms.

Settings. We evaluated all systems with the permissioned setting, where all participants are explicitly identified. We ran each experiment ten times and reported the average of the metrics. For each system, we created five organizations, each with two executors and one hundred clients. For Geco-based systems (i.e., Geco, Geco (w/o. Ctm), and Geco (w/o. defense)), we designated one executor as the lead executor for each organization. For HLF and Geco-based systems, we created four orderers running the PBFT [11] protocol.

In §6.1 and §6.2, we evaluated the systems' performance under conflicting transactions using the SmallBank workload. We designated 1% of the client accounts as *hot accounts*, and configured the *conflict ratio*, which denotes the probability of each transaction accessing the hot accounts.

Our evaluation focused on three primary questions.
§6.1 How efficient is Geco compared to baselines?
§6.2 How robust is Geco to malicious participants?
§6.3 How efficient is Geco under diverse applications?

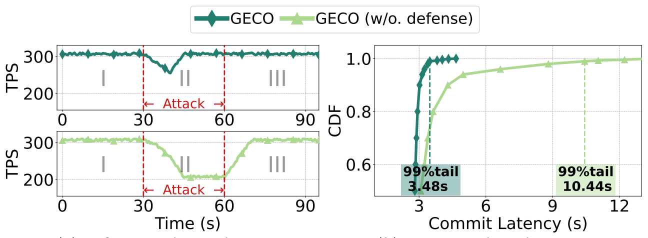
6.1 End-to-End Performance

We first evaluated the end-to-end performance in benign environments on four systems: Geco, Geco (w/o. Ctm), ZeeStar, and HLF. In the benign environment, the network was stable, and all participants were correct. We conducted three experiments using the SmallBank workload with conflict ratios of 10%, 50%, and 90%, respectively. For each experiment, the benchmark tool generated a series of *Transfer* transactions (Listing 1). In each transaction, a sender client transferred a number of tokens to a receiver client. Note that both clients were randomly selected, and these clients may belong to different organizations. For aborted transactions, we repeatedly submitted them until they were successfully committed.

Geco achieved high effective throughput, outperforming both ZeeStar and Geco (w/o. Ctm) by up to 7.14×. As shown in Figure 5, Geco achieved effective throughputs of 255, 307, and 343 TPS at conflict ratios of 10%, 50%, and 90%, respectively. In contrast, Geco (w/o. Ctm) achieved only 235, 95, and 55 TPS, indicating a notable performance gap between Geco and Geco (w/o. Ctm). This performance gap became more evident as the conflict ratios increased. ZeeStar performed even worse, achieving merely 54, 52, and 48 TPS in the three experiments, respectively. Geco achieved a low average end-to-end latency of 2.7s and the shortest 99% tail latency among all four systems. Note that Geco's average latency was 14% lower than the ZeeStar, as shown in Table 7.

We explain Geco's high performance in two aspects. Firstly, Geco's Ctm protocol (§4.2) greatly reduced conflicting aborts. Although Geco incurred an 11% extra latency compared to Geco (w/o. Ctm), it can be justified by the significant throughput gains. Secondly, Geco proved the execution correctness using the lightweight consistency check NIZKPs (§4.2), which were agnostic to the contract logic. In contrast, ZeeStar relied on the inefficient re-execution method (§1) for NIZKP generation. This enables Geco to achieve shorter commit latency than ZeeStar, as confirmed in Table 7.

Compared with HLF, which provide no data confidentiality guarantee, Geco incurred an average performance overhead of 32%. This overhead is attributed to three aspects. Firstly,



(a) Effective throughput. **(b)** Latency distributions.

Figure 6. Performance under malicious participants.

Geco introduces an extra preprocessing phase, which merges correlated conflicting transactions and encrypts transaction parameters. Secondly, in the execution phase, Geco performs costly FHE computations and NIZKP generation. Lastly, in the validation phase, Geco verifies the consistency check NIZKPs. We believe that these overheads are necessary and practical in order to fulfill Geco's confidentiality guarantee.

Geco is suitable for contending scenarios. As Figure 5 shows, when the conflict ratio increased, Geco (w/o. Ctm), ZeeStar, and HLF suffered from a notable decrease in throughput. However, Geco demonstrated an increase in throughput. Furthermore, Geco outperformed HLF with 1.54x higher throughput in the scenario with 90% conflict ratio, despite the extra overhead for achieving Geco's confidentiality guarantee. This was attributed to the Ctm protocol, which effectively tackled conflicting aborts and reduced invocations of expensive cryptographic primitives.

Overall, Geco ensures data confidentiality while achieving high performance, making Geco particularly suitable for safety-critical and performance-sensitive applications.

6.2 Robustness to Malicious Participants

We conducted NIZKP omission attacks (§4.3) on Geco and Geco (w/o. defense) with the same settings as §6.1. We set the conflict ratio as 50%. We designated four benign organizations (O_B) and one malicious organization (O_M). The lead executor of O_M conducted NIZKP omission attacks toward all transactions involving O_M clients. The experiment has three periods: (I) *pre-attack*, (II) *attack*, and (III) *post-attack*.

Our defense mechanism against the NIZKP omission attack (§4.3) is effective. Figure 6a shows that both Geco and Geco (w/o. defense) witnessed performance degradation at the onset of the attack. However, Geco quickly recovered its peak throughput, while Geco (w/o. defense)'s throughput remained poor until we terminated the attack. Figure 6b shows that Geco's tail latency barely changed during the attack (compared to Figure 5d). In contrast, Geco (w/o. defense) witnessed notable degradation in tail latency as the transactions involving O_M were repeatedly re-submitted and aborted, wasting the computation resources of executors. These differences were due to Geco's defense mechanism, which successfully detected the malicious lead executor of O_M and early rejected the transactions involving O_M. Overall, Geco achieves high performance even in the presence of malicious participants, and thus is suitable for applications with high security requirements like supply chain [35] and bidding [40].

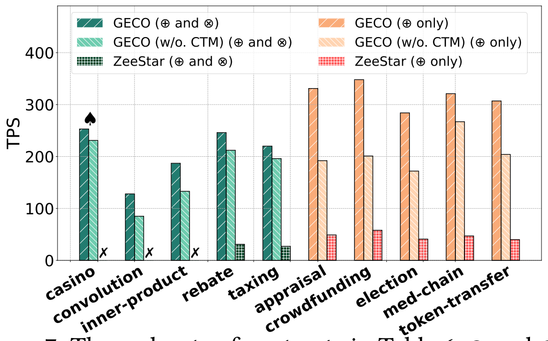


Figure 7. Throughputs of contracts in Table 6. ⊕ and ⊗ denote FHE addition and multiplication. ♠ is non-deterministic contracts. ✗ is contracts not supported by ZeeStar.

6.3 Performance under Diverse Applications

We evaluated Geco, Geco (w/o. Ctm), and ZeeStar using the second workload, which consists of ten diverse smart contracts (Table 3). As Figure 7 shows, Geco exhibited high performance in all evaluated contracts, achieving throughput ranging from 120 TPS (`convolution`) to 348 TPS (`crowdfunding`). The Ctm protocol delivered notable throughput gains, especially in contracts where conflicts are prevalent, such as `election` and `token-transfer`. Furthermore, Geco can support general applications with contracts that are non-deterministic (e.g., `casino`) and involve foreign values (e.g., `inner-product`). In contrast, existing confidentiality-preserving blockchains (e.g., ZeeStar) can only support deterministic contracts and do not support arbitrary arithmetic computations on foreign values.

In summary, Geco ensures data confidentiality while achieving high performance for general smart contracts, enabling the deployment of diverse applications developed with general-purpose programming languages like Go and Java [4]. Thanks to Geco's contract logic-agnostic trusted execution (§4.2), we can easily integrate more advanced cryptographic primitives like CKKS [43], which support floating point FHE computations, without modifying our protocol.

7 Conclusion

We present Geco, a high-performance confidential permissioned blockchain framework for general non-deterministic smart contracts and cryptographic primitives. Geco ensures the correctness of transaction executions by efficiently generating NIZKPs that are agnostic to contract logic. Moreover, our Ctm protocol effectively enhances Geco's performance in contending applications, illustrating a new approach for tackling conflicting aborts. Extensive evaluation demonstrates that Geco achieves superior performance compared to baselines, making Geco an ideal choice for a wide range of blockchain applications that desire both data confidentiality and high performance.

References

[1] 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.

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Caper: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019.

[3] Mohammad Javad Amiri, Boon Thau Loo, Divyakant Agrawal, and Amr El Abbadi. Qanaat: A scalable multi-enterprise permissioned blockchain system with confidentiality guarantees. *arXiv preprint arXiv:2107.10836*, 2021.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.5*, 2020.

[10] 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.

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

[12] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

[13] consensys. Gnark, 2023.

[14] Rafaël Del Pino, Vadim Lyubashevsky, and Gregor Seiler. Short discrete log proofs for fhe and ring-lwe ciphertexts. In *IACR International Workshop on Public Key Cryptography*, pages 344–373. Springer, 2019.

[15] 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.

[16] HyperLedger Fabric. 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.

[17] Hyperledger Fabric. Hyperledger/fabric at release-2.5, 2023.

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

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

[20] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[21] 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.

[22] 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.

[23] 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.

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

[25] Change Healthcare. Change healthcare case study, 2023.

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

[27] 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.

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

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

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

[31] 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.

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

[33] NCCGroup, 2016.

[34] 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.

[35] 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.

[36] 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.

[37] Microsoft SEAL (release 4.0). https://github.com/Microsoft/SEAL, March 2022. Microsoft Research, Redmond, WA.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.