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

ABSTRACT

Multi-party 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 the encrypted data.

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 three notable confidential blockchains, Geco achieved up to 7.14× higher effective throughput and 13% lower end-to-end latency on average.

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 participant nodes and clients. However, notable blockchains like Hyperledger Fabric (HLF) [5] and Ethereum [47] 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 the medical data [28], as they are subject to stringent privacy laws and regulations, such as the General Data Protection Regulation of the European Union [46].

Many previous work has been proposed to achieve data confidentiality on smart contracts, and existing work can be summarized into two distinct approaches. The first is the architecture approach. It works by introducing new blockchain architectures to impose strict access controls on data without resorting to cryptography primitives [3–5]. For example, HLF uses channels [6] and private data collections [22] to determine which participants are eligible to access certain data. However, this approach still processes and stores client data in plaintext, and is not designed for handling the sensitive client data leakage caused by malicious nodes that can access the data.

The second is the cryptographic approach. This approach employs cryptographic primitives like homomorphic encryptions to encrypt client data for confidentiality and mandates the smart contract to execute directly on the encrypted data [30, 42–44], preventing the exposure of sensitive data to potential malicious nodes. This approach relies on the non-interactive zero-knowledge proof (NIZKP) to validate the correctness of contracts' execution results (in ciphertext), without leaking any sensitive data. Specifically, existing studies employ a *re-execution method* to generate NIZKPs. In this method, the NIZKP decrypts the ciphertext transaction input and result, re-executes the transaction using the decrypted input, and verifies the consistency between the decrypted and re-executed results. For example, ZeeStar [42] employs a partial homomorphic encryption (PHE) scheme to encrypt client data and uses the re-execution method to generate NIZKPs for proving the execution correctness.

However, despite these great advancements, existing work of the cryptographic approach still faces two fundamental limitations. Firstly, existing work is incompatible with non-deterministic smart contracts, whose different executions may produce inconsistent results even with the same input and initial state [5]. Non-deterministic contracts are gaining popularity as developers can create contracts in general programming languages like Java and C++, and achieve high performance through non-deterministic language features like multi-threading [37]. However, these contracts are incompatible with NIZKPs generated with the above re-execution method, which implies that identical input will always produce the same result.

Secondly, the existing cryptographic 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 [12, 20] (i.e., an FHE scheme) and the elliptic-curve based NIZKP [15] takes tens of seconds to generate or verify one NIZKP [41]. Even when employing lightweight cryptographic primitives like PHE which restrict arithmetic operations, the NIZKP operations remain cumbersome. For instance, generating a Groth16 [35] NIZKP using 2048-bit keys for the Paillier PHE encryption scheme [36] consumes more than 256 GB memory [42], which is impractical for commodity desktops available today.

Overall, we believe the re-execution method employed by existing cryptographic approaches for generating NIZKPs is the root cause of their inability to support general non-deterministic contracts and their poor end-to-end performance, which is often a small fraction of the requirements for typical blockchain applications (e.g., 1k txns/s for trading [37]).

In this study, our key insight to address these limitations is that,

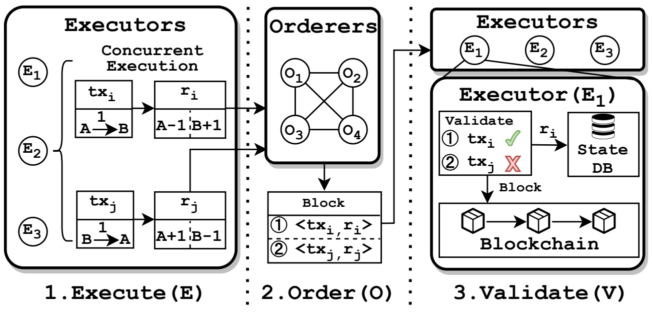


Figure 1: The EOV workflow executes transactions on multiple executors for trusted execution. On the executor E_2, tx_i and tx_j produce execution results r_i and r_j. For high performance, EOV concurrently executes tx_i and tx_j on each executor, causing conflicting aborts.

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 cryptographic primitives like 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 (e.g., Hyperledger Fabric [5]). The mechanism first executes transactions (*endorsing*) on multiple executor nodes (Figure 1). The execution is considered correct if quorum nodes produce consistent results. EOV eliminates the potential result inconsistency caused by non-determinism, enabling the support for non-deterministic smart contracts. By decoupling the proving logic of execution correctness from the smart contract logic, EOV significantly reduces the correctness proving overhead for smart 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 encrypted data. Ceov executes transactions in three steps. Firstly, a client sends 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., an FHE scheme), as specified by the invoked contract, to encrypt the input data. Secondly, the lead executor relays the transaction to multiple executors, which independently and concurrently execute the transaction over the encrypted data. Thirdly, the lead executor generates a NIZKP 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.

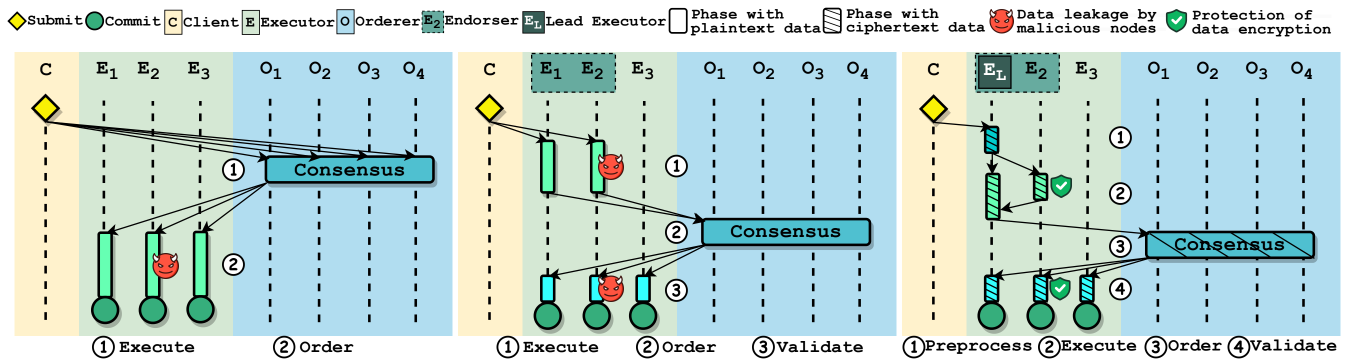
However, Geco still faces a non-trivial performance challenge due to the conflicting transaction aborts of EOV. In particular, EOV concurrently executes all transactions for high performance and checks conflicts before committing transactions for data consistency (i.e., optimistic concurrency control [5]). As shown in Figure 1, when multiple transactions concurrently access the same key-value pair in the blockchain state database, and at least one of them modifies the value, only one transaction can successfully commit (e.g., tx_i); other conflicting transactions must abort (e.g., tx_j). Such transaction conflicts are prevalent in typical blockchain applications. For example, in stock exchange [40], Hyperledger Fabric achieved only 24.5% of its peak throughput, as confirmed in our evaluation (Figure 5). This challenge is further exacerbated in Geco due to the involvement of resource-intensive cryptographic primitives, which incur significant computation overhead. Microsoft SEAL [39], a highly optimized FHE library, takes 2822 milliseconds to execute a single homomorphic multiplication [33]. Consequently, transaction conflicting aborts cause a significant waste of computing resources and performance degradation on executor nodes.

To tackle this challenge, we propose a Correlated Transaction Merging (Ctm) protocol, which exploits the similarities of conflicting transactions in permissioned blockchain applications to tackle the conflicting aborts. Specifically, conflicting transactions invoking the same contract generally modifies the same key with the same operation. By merging these transactions into a single one, we can commit all transactions without any aborts while preserving the semantics of the original transactions. For instance, consider a scenario where two transactions both attempt to perform an addition operation on the same key (e.g., two $X+1$ operations). In such cases, Ctm merges these two addition transactions into a single transaction (e.g., one $X+2$ operation). Ctm comprises two sub-protocols: *offline analysis* for smart contracts prior to deployment, and *online scheduling* for transactions during runtime. Before deployment, a smart contract undergoes Ctm's offline analysis, which determines whether multiple conflicting transactions invoking this contract can be merged into a single equivalent transaction. During runtime, Ctm's online scheduling constructs a transaction dependency graph on a per-block basis to identify mergeable conflicting transactions and merge them into a single one. Ctm is especially suitable for smart contracts that operate on homomorphically encrypted data. It effectively reduces the waste of computing resources stemming from conflicting aborts with a moderate overhead, and decreases the frequency of invocations for expensive cryptographic operations.

We implemented Geco on the codebase of HLF [5], the most notable EOV permissioned blockchain framework. We integrated Geco with Lattigo [33], a highly efficient FHE library, and gnark [14], a popular NIZKP library. We evaluated Geco using the SmallBank workload [27] (the de facto blockchain benchmark) under different conflict ratios, and ten diverse blockchain applications, including both deterministic and non-deterministic smart contracts. We compared Geco with HLF and ZeeStar [42], covering both the access-control and cryptography baselines for data confidentiality. Our evaluation shows that:

- Geco is efficient (§6.1). Geco achieved up to 7.14× higher effective throughput and 13% lower end-to-end latency than the ZeeStar (cryptography). Although Geco witnessed a 32% drop in throughput compared to HLF (access-control), HLF does not preserve confidentiality with malicious nodes.
- Geco is secure (§6.2). Geco can achieve high effective throughput under attacks from malicious participants and conflicting transactions.
- Geco is general (§6.3). Geco tolerates non-deterministic smart

[1]Geco stands for GEneral COnfidentiality-preserving blockchain framework


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
Consensus
① Execute ② Order
① Execute ② Order ③ Validate
① Preprocess ② Execute ③ Order ④ Validate

(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. Endorsers are executors selected for executing a transaction (§3.1).

contracts and supports cryptography primitives like FHE which allows arbitrary computations over encrypted data.

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 encrypted data. 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 transactions with data races. Overall, Geco can benefit blockchain applications that desire both data confidentiality and high performance, such as supply chain [18] and healthcare [31]. Geco can also attract broad traditional non-deterministic applications, developed with general-purpose programming languages like Golang and Java, to be deployed upon. Geco's code is available at https://github.com/sigmod25geco/sigmod25geco.

2 BACKGROUND

2.1 EOV Permissioned Blockchain

A blockchain is a distributed ledger that records transactions across mutually untrusted nodes. It can be summarized into two main categories: *permissionless* and *permissioned.* Permissionless blockchains (e.g., Bitcoin [34] and Ethereum [47]) are open to anyone, and their participants are mutually untrusted. In contrast, permissioned blockchains (e.g., HLF [5]) 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 [1] to tolerate malicious nodes.

Typical permissioned blockchains can be further classified into two types according to their workflows: *order→execute* (OE) and *execute→order→validate* (EOV). As depicted in Figure 2a, the OE workflow involves two steps: Firstly, the orderer nodes establish a globally consistent transaction order (**O**); Subsequently, all executor nodes sequentially execute the transactions in the predetermined order (**E**). This workflow leads to significant performance issues and cannot tolerate non-deterministic transactions since each executor node independently executes all transactions.

In contrast, the EOV workflow (shown in Figure 2b) incorporates a **quorum-based trusted execution mechanism** to achieve high performance and to tolerate non-deterministic transactions.

This mechanism decouples the trust model of applications from the trust model of the blockchain. 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 for executing each transaction. The transaction's execution is deemed correct when a quorum of these executors produce consistent execution results for the transaction.

The EOV workflow has three phases. Initially, a group of executors called *endorsers* are selected to concurrently execute each transaction (**E**). Then, the client collects the executed results of each transaction from different endorsers to check their consistency; if the execution results are consistent, the execution is regarded as correct. The client sends the results to the orderers. Subsequently, 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**).

Geco's new Ceov workflow (Figure 2c) extends EOV's quorum-based trusted execution mechanism to preserve data confidentiality while supporting general non-deterministic contracts and cryptographic primitives (FHE and NIZKP), Geco inherits the advancements of EOV while tackling its limitations. On the one hand, Geco concurrently executes transactions to achieve high performance and tolerates general non-deterministic contracts. On the other hand, Geco tackles EOV's conflicting aborts caused by EOV's concurrent executions with a new Ctm protocol (§4.2).

2.2 Homomorphic Encryption

Homomorphic encryption (HE) is a family of encryption schemes that allow direct computation over encrypted data without decryption. A HE scheme consists of four algorithms: *KeyGen* generates key pairs necessary for other operations; *Enc(m, pk, r)* uses the public key *pk* to encrypt a plaintext message *m* into a ciphertext *c* along with randomness r; *Dec(c, sk)* uses the private key *sk* to decrypt ciphertext c back into 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 can be summarized into two main categories based on the supported operators: partial homomorphic encryption (PHE) and fully homomorphic encryption (FHE). PHE schemes are limited in their support to only one specific type of arithmetic operation, either addition or multiplication. In contrast, FHE schemes allow for arbitrary combinations of these operations. Although PHE schemes have relatively low computation overhead, they are limited in their computational capabilities. This restriction compromises the generality of smart contracts, as discussed in §1. FHE schemes typically demand higher computation overhead, but they are 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 [23], making FHE a promising and practical solution for supporting general smart contracts. Geco utilizes the BFV scheme [12, 20] (a FHE scheme) implementation of Lattigo [33], which is an optimized FHE library. This implementation enables Geco to perform FHE computations on 58-bit unsigned integers within tens of milliseconds, striking a balance between performance and smart contract generality.

2.3 Non-Interactive Zero-Knowledge Proofs

Non-interactive zero-knowledge proof (NIZKP) [21, 25] 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 an NIZKP using a proving key, V can verify the proof using the corresponding verifying key without P being present. Formally, given a proof circuit ϕ, a private input s, and some public input x, prover P can generate an NIZKP to prove knowledge of s satisfying the predicate $\phi(s; x)$. One popular type of NIZKP is called zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) [10, 26, 48], which allows any arithmetic circuit ϕ and guarantees constant-cost proof verification in the size of ϕ, making them suitable for blockchain applications [2, 7, 16, 35, 43]. Specifically, Geco employs the Groth16 [35] (a zkSNARKs construction) implementation of the gnark library [14] 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.

Access-control-based approach. Several previous work attempt to achieve data confidentiality by adopting specialized blockchain architectures without utilizing cryptography technology. HLF [5] enforces access control to client data on the inter-contract and the intra-contract levels through two architectures, channels [6] and private data collection [22], respectively. Caper [3] requires each participant party to maintain only a partial view of the global ledger and prohibits them from owning copies of other parties' data. Qanaat [4] 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 do not utilize cryptography primitives, but process and store 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
✧ HLF [5]	✗	✓	✓	✓
✧ Qanaat [4]	✗	✓	✓	✓
✧ Caper [3]	✗	✓	✓	✓
♦ ZeeStar [42]	✓	✓	✗	✗
♦ SmartFHE [41]	✓	✓	✗	✗
♦ Zapper [44]	✓	✗	✗	✗
♦ FabZK [30]	✓	✗	✗	✗
♦ **Geco**	✓	✓	✓	✓

Table 1: Comparison of Geco and related confidentiality-preserving blockchain systems. "✧ / ♦" means that the system either takes the access-control approach (✧) or the cryptography approach (♦).

Cryptography-based approach. Unlike the aforementioned solutions, many prior work use cryptographic primitives to protect confidentiality and enforce the correct execution of smart contracts.

ZeeStar [42] uses exponential ElGamal encryption [32], which is an additive PHE scheme, for encrypting transaction data. It claims to support on-ciphertext multiplication by repeatedly performing the addition operation. However, this extension is highly inefficient and only applicable to limited scenarios. Moreover, ZeeStar cannot tolerate non-deterministic smart contracts due to its reliance on the OE workflow, as discussed in §1. Consequently, these factors result in that ZeeStar does not support general smart contracts.

SmartFHE [41] is an Ethereum-based blockchain that uses FHE schemes to preserve confidentiality. However, two limitations impede its support for general smart contracts. Firstly, SmartFHE is also unable to tolerate non-deterministic smart contracts since it relies on the OE workflow. Secondly, SmartFHE does not support smart contracts involving foreign values (i.e., ciphertexts encrypted by different parties), because this requires a multi-key variant of SmartFHE that is currently impractical according to the authors.

Zapper [44] uses an NIZK processor and an oblivious Merkle tree construction to provide confidentiality for both its users and the objects they interact with. However, Zapper faces the same limitations with SmartFHE: Zapper neither can tolerate non-deterministic smart contracts, nor support computation on foreign values. These significantly undermine the compatibility of Zapper with general smart contracts.

FabZK is a HLF-based blockchain that adopts a specialized tabular ledger data structure comprising $N + 3$ columns, with N representing the participating organizations. This structure is designed to obfuscate the transaction-related organizations. However, this data structure significantly restricts the compatible blockchain applications, compromising its capability to support general smart contracts. Additionally, as the number of organizations increases, the performance of FabZK is severely compromised since each transaction needs to update all columns in the ledger.

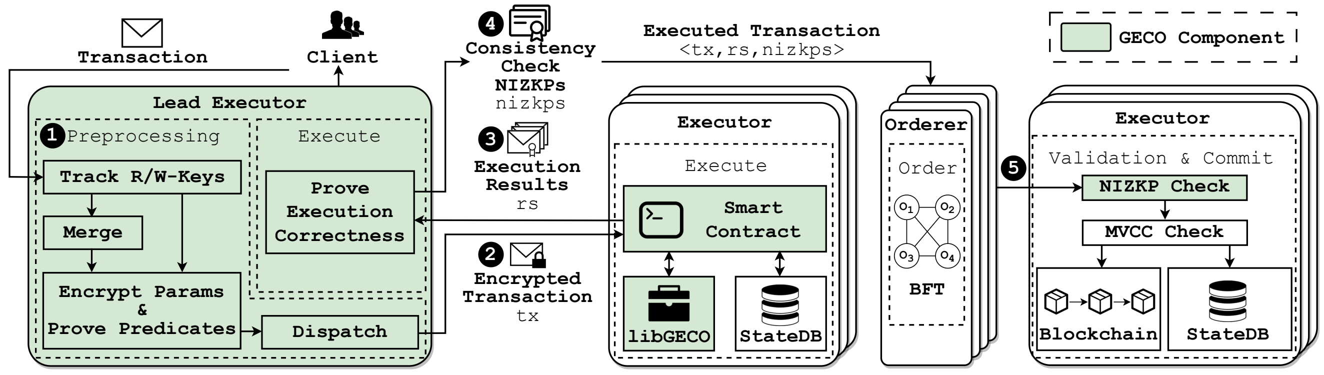


Figure 3: Geco's runtime workflow. Geco components are highlighted in green.

3 OVERVIEW

3.1 System Model

Same as existing EOV permissioned blockchains, Geco contains 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. The roles and functionalities of each participant type are described as follows:

Clients. In Geco, clients are the only participant type that can submit transactions to nodes for execution and commitment. Other participant types, namely executors and orderers, are not permitted to submit transactions.

Executors. In Geco, all executors are responsible for three main tasks: executing transactions, validating results, and maintaining an up-to-date local copy of the blockchain state database. To support these tasks, executors leverages libGeco (§3.3), a library that we developed to offer a range of NIZKP functionalities, which are essential for the correct functioning of Geco.

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

- Detect multiple conflicting transactions and merge them into a single transaction.
- 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 [13]).

3.2 Threat model

Geco adopts the Byzantine failure model [8, 13], 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 typical permissioned blockchains such as HLF, a participant trusts 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

API	Description
	Lead executor APIs
$\boxed{1}$ Preprocess	Preprocess the given transaction according to the runtime transaction schedule protocol (§4.2).
$\boxed{2}$ ProveConsistency	Generate *consistency check NIZKPs* to prove the consistency of the quorum results.
$\boxed{3}$ VerifyConsistency	Verify the given *consistency check NIZKPs* to ensure that the quorum results are consistent.
	Smart contract APIs
$\boxed{4}$ Analyze	Perform the offline analysis protocol (§4.1) on the given smart contract.
$\boxed{5}$ Require	Require that the given predicate for a contract is true; otherwise, abort the transaction execution.

Table 2: libGeco APIs.[2]

Geco provides a library named libGeco with five APIs (Table 2). The lead executor APIs assist lead executors in preprocessing transactions and ensuring the correctness of transaction execution. 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 encrypted data. Note that the Require API follows the idea of the *require* statement of ZeeStar [42]. 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 (!HasClientID(src) or !HasClientID(dst)) {Abort();}
  if (src != GetClientID()) {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 smart 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

[2]Unless otherwise specified, $\boxed{?}$ refers to libGeco APIs in Table 2.

Data type	Description
`id`	Represent a unique identifier for a participant client in Geco.
`euint`	Represent a single unsigned integer plaintext data, but stores different ciphertext data encrypted for different clients.
Function	**Description**
`HasClientID`	Check the existence of the given client `id`
`GetClientID`	Get the unique `id` associated with the sender client who submits the transaction.
`GetState`	Read a state from the blockchain state database.
`PutState`	Write a state to the blockchain state database.
`Add`	Perform FHE addition on the given input ciphertexts.
`Sub`	Perform FHE subtraction on the given input ciphertexts.

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

`Sub` are implementation-independent; they can be implemented with any FHE scheme that supports addition and subtraction on encrypted data, such as the BFV scheme [12, 20]. In this contract, the `Require` API (introduced in Table 2) verifies a NIZKP that decrypts the encrypted data involved (i.e., `srcBalance` and `val`), and checks the validity of the predicate applied to the decrypted data. This NIZKP is generated by the lead executor during the preprocessing phase (§4.2) and is attached to the transaction. Intuitively, this smart contract achieves the following confidentiality guarantee: each client's balance is kept confidential and only accessible to the respective client, while the specific number of tokens being transferred remains 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). Prior to smart contract deployment, the developer invokes libGeco `Analyze` API (Table 2) on the contract. The `Analyze` 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 that invoke this contract with identical `src` and `dst` ($tx_1 : A \xrightarrow{1} B$ and $tx_2 : A \xrightarrow{1} B$) can be merged into a single transaction ($tx_{1,2} : A \xrightarrow{2} B$) by adding their corresponding *val* parameters, 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 libGeco `Preprocess` API (Table 2) to preprocess the transactions submitted by clients. Firstly, the lead executor invokes the Ctm protocol to merge transactions (❶ in Figure 3 [3]): the lead executor collects transactions submitted by clients within the same organization (§3.2), merges conflicting transactions whose parameters can be merged without changing their semantics, and leaves other transactions unchanged. Subsequently, the lead executor encrypts all transaction's plaintext inputs of the data type `euint` using the public keys (i.e., encryption keys) of corresponding organizations. Additionally, for smart contracts using libGeco `Require` API introduced in Table 2, the lead executor generates NIZKPs to prove the given predicates. Lastly, the lead executor selects a group of executors, known as endorsers in EOV [5], and dispatches the encrypted transactions to the endorsers for execution.

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

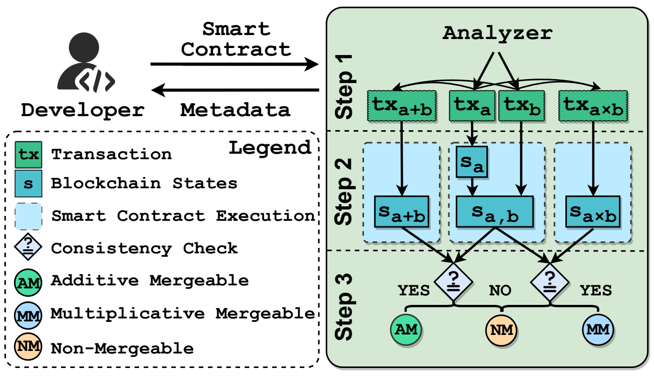


Figure 4: Geco's offline smart contract analysis protocol. tx_{a+b} and $tx_{a\times b}$ are formed by merging tx_a and tx_b through adding or multiplying their input parameters (§4.1). s_{a+b} and $s_{a\times b}$ are the states produced by executing tx_{a+b} and $tx_{a\times b}$. $s_{a,b}$ is the state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

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

After receiving execution results from all endorsers(❸), the lead executors invoke the `ProveConsistency` API (Table 2) to generate NIZKPs for proving that quorum executors (i.e., a majority of executors) have produced consistent results for keys belonging to the same organization, thereby ensuring the correctness of execution.

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 order of transactions in blocks and disseminate the block to all executors for validation and commitment.

Phase 4: Validation and commit. When receiving a block from the orderers(❺), the executor sequentially validates each transaction within the block in the predetermined order. During the validation process, the executor invokes libGeco `VerifyConsistency` API (Table 2) 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 and updates the blockchain state database accordingly. Unlike existing EOV permissioned blockchains that abort conflicting transactions [5], Geco's Ctm protocol commits both conflicting transactions tx_1 and tx_2 as a whole in $tx_{1,2}$.

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 like FHE (§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 specific logic of smart contracts 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 user data using cryptographic primitives capable of performing arbitrary arithmetic computations over

Function	Description
`GetAllCFPs`	Get all control flow paths of the given contract.
`GenRandStateDB`	Randomly generate a state database for the given contract and control flow path.
`GenRandParams`	Randomly generate transaction parameters for the given contract and control flow path.
`GenTX`	Generate a transaction based on the parameters provided by `GenRandParams`.
`ExecTX`	Execute a transaction over a state database and return the updated state database.
`GetSize`	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.
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.

ciphertexts, such as FHE.

Secondly, in contrast to existing EOV permissioned blockchains which abort conflicting transactions [7, 42], 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 number 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 *value*. The key parameters specify the contract's accessed keys while the value parameters are used to derive the corresponding 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 corresponding 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 corresponding value parameters of the conflicting transactions. A smart contract is *multiplicative mergeable* iff it is mergeable and generates new value parameters by multiplying the corresponding value parameters of the conflicting transactions. Otherwise, the contract is *non-mergeable*.

The *offline smart contract analysis*, also known as *offline analysis*, determines whether a smart contract is additive mergeable, multiplicative mergeable, or non-mergeable. libGECO `Analyze` API implements the offline analysis protocol. As shown in Figure 4, GECO analyzes the smart contract in three steps using random interpretation [2], 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

Algorithm 1: Offline analysis protocol (§4.1; API $\boxed{4}$)

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

the mergeability of the smart contract. During runtime, lead executors inspect the metadata and only carry out transaction merging on additive mergeable and multiplicative mergeable smart contracts.

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

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 corresponding 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 corresponding 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 modification APIs (e.g., `PutState()`). For analysis

Function	Description
`GetRWKeys`	Get all read and write keys of a transaction.
`DetectConflict`	Detect and return the set of mergeable transactions and the set of non-mergeable transactions.
`Merge`	Merge the given set of mergeable transactions.
`Encrypt`	Encrypt the input parameters of the transactions.
`Dispatch`	Dispatch the given transactions to endorsers.
`Execute`	Execute the transaction and return the result.
`ReplyResult`	Send the execution results to the lead executor of the given transaction.
`ProvePredicates`	Generate NIZKPs to prove predicates if the smart contract invokes the `Require` API.
`SendForOrdering`	Send the transaction for ordering.
`CheckMVCC`	Perform a multi-version concurrency control check on the execution results.
`Commit`	Commit the given transaction.
`AppendBlock`	Append a block to the local copy of blockchain.
Variable	**Description**
K	All read-write keys of transactions within a block.
Q	The queue that buffers transactions during Phase 1.
$QSize$	The maximum number of transactions in Q.
txs_m	The set of mergeable transactions.
txs_n	The set of non-mergeable transactions.
txs_e	The set of encrypted transactions.

Table 5: Functions and variables for Algorithm 2.

step 2, the contract should not contain recursive contract calls [2]. 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.

4.2 Runtime Transaction Schedule

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

Phase 1: Preprocessing. Following our CTM protocol, the lead executor invokes libGECO `Preprocess` API (Table 2) to preprocess and encrypt the client transactions.

Phase 1.1: Tracking read-write keys. For each block, the lead executor buffers the transactions in a queue and keeps track of each transaction's read-write keys. For example, for a transaction ($tx : A \xrightarrow{1} B$) that invokes the smart 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 the number of queuing transactions exceeds a specified size or a timeout occurs, the lead executor detects conflicting transactions according to the transactions' read-write keys. Based on the detection, the lead executor performs 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

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

```
   // Phase 1: Preprocessing (Invoke API [1])
1  $K \leftarrow \emptyset$; $Q \leftarrow \emptyset$;
2  Upon reception of Transaction $tx$ from client; do
3      $K \leftarrow K \cup$ GetRWKeys($tx$);
4      $Q \leftarrow Q \cup tx$;
5  Upon $|Q| \geq QSize$ or timeout
6      $txs_m, txs_n \leftarrow$ DetectConflict($K, Q$);
7      $txs_m \leftarrow$ Merge($txs_m$);
8      $txs_e \leftarrow$ Encrypt($txs_m, K$) $\cup$ 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     $r \leftarrow$ Execute($tx$);
13     ReplyResult($tx, r$);
   // Phase 3: Ordering
14 Upon reception of all Results $rs$ for Transaction $tx$; do
15     $nizkps \leftarrow$ ProveConsistency($rs$);        // Invoke API [2]
16     SendForOrdering($tx, rs, nizkps$);
   // Phase 4: Validation and commit
17 Upon reception of Block $blk$; do
18     For $tx, rs, nizkps$ in $blk$; do
19         if VerifyConsistency($rs, nizkps$) $\wedge$       // Invoke API [3]
            CheckMVCC($rs$) then
20             Commit($tx, rs, nizkps$);
21         AppendBlock($blk$);
```

either an additive mergeable or multiplicative mergeable smart contract, respectively. For example, consider two transactions that invoke the additive mergeable smart contract in Listing 1: $tx_1 : A \xrightarrow{1} B$ and $tx_2 : A \xrightarrow{1} B$. The lead executor merges these transactions by preserving the key parameters $\{src = A, dst = B\}$, and adding the original value parameters to generate the new value parameter $\{val = 2\}$, resulting in the merged transaction $tx_{1,2} : A \xrightarrow{2} B$.

Phase 1.3: Encrypting transaction parameters and proving predicates. The lead executor encrypts the transaction input parameters that are of the data type `euint`, namely encrypted unsigned integers (Table 3). The lead executor analyzes the read-write keys associated with the `euint` parameters, then encrypts the parameter plaintexts using the encryption keys of the clients specified by the read-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 read-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 smart contracts that invoke the `Require` API (Table 2) to validate specific predicates. For example, the smart contract in Listing 1 requires that the encrypted `srcBalance` must be greater than `val` for a successful token transfer. Thus, the lead executor generates a NIZKP which first decrypts the encrypted data (`srcBalance`) and then proves whether the predicate holds (`srcBalance` > `val`). The NIZKPs are attached to the transactions for verifications by other participants.

Phase 1.4: Dispatch. The lead executor disseminates transactions to the involved organizations' executors (we name these executors "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 smart contract specified by the transaction. The execution produces a read-write set as the execution result, which records the blockchain states that the transaction reads from and writes to the blockchain state database. Next, the endorser signs the result and replies it to the lead executor for checking the correctness of executions.

Phase 2.2: Checking the correctness of executions. After collecting a transaction's results from endorsers of all involved organizations, the lead executor ensures correct executions by proving that quorum executors have produced consistent read-write sets. However, the read-write sets contain HE ciphertext data that cannot be directly compared without decryption as they involve random noise for security reasons [45]. To address this issue, the lead executor invokes libGeco `ProveConsistency` API (Table 2) to generate a *consistency check NIZKP* that first decrypts the ciphertext data belonging to clients within the same organization, and then checks the consistency of the decrypted data. 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 ciphertext data 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 execution 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. The results of tx are deemed consistent only when the same quorum of endorsers (e.g., org_1 and org_2) produce consistent results for all keys A, B and C, in tx's read-write set results.

Phase 3: Ordering. Geco orderers run a BFT consensus protocol (e.g., BFT-SMaRT [9]) 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 and commit.

Phase 4: Validation and commit. The executor sequentially validates each transaction upon receiving a block from the orderers. This validation process involves two steps: (1) invoking libGeco `VerifyConsistency` API (Table 2) to verify the consistency NIZKPs associated with the transactions, and (2) performing a multi-version concurrency check on the execution 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). Once the executor has applied the above procedure to 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 access the cleartext data of only clients within the same organization, as participants in the same organizations are mutually trusted (§3.2). For clients in other organizations, the participant can only access the ciphertext data.

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 (e.g., network failures) or maliciously omit the submissions of the consistency check NIZKPs for specific transactions submitted by other organizations, 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 above NIZKP omission attacks caused by malicious lead executors, Geco mandates the lead executors to submit the *consistency check NIZKPs* for all involved transactions, even if the transaction's execution results from different executors are inconsistent. To reduce false positives caused by network failures, Geco executors track the number of NIZKP omissions for different lead executors. A lead executor is deemed malicious only when the number of omissions caused by the lead executor exceeds a configurable upper bound (by default, ten omissions). Geco executors proactively reject transactions that involve organizations of the malicious lead executors in Phase 1 for a long duration (by default, 1000 blocks, after which the omission number for a lead executor is reset to zero), preventing the potential wastage 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 resulting state is equivalent 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 [5] 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 [9] and their implementations. This ensures that all benign executors process and append 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.

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

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

- *Atomicity.* GECO commits or aborts each transaction as an indivisible unit.
- *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 GECO transaction operates as if it is the only transaction executing, without interference or conflicts with other transactions.
- *Durability.* Once a GECO transaction is committed, its changes to the blockchain state become permanent and cannot be reversed. □

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 guarantee from existing EOV permissioned blockchains [5]. 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 benign 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 [25]. 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 [12, 20]) achieves *indistinguishability under chosen-plaintext attack* (IND-CPA [11]), 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 top of Hyperledger Fabric v2.5 [19]. GECO uses the implementation of the BFV scheme [12, 20] provided by the Lattigo library [33]. With the pre-configured cryptographic parameters, GECO supports FHE arithmetic computation over 58-bit unsigned integers, which satisfy the requirements of diverse smart contracts listed in Table 6. GECO uses the implementation of the Groth16 NIZKP system [25] on the BN254 elliptic curve provided by the gnark library [14]. Same as existing systems [7, 42] that also use Groth16, our implementation requires a circuit-specific trusted setup to generate the verification key for each NIZKP circuit. This trusted setup is a one-time procedure with negligible overhead, and hence it is not discussed in our evaluation.

Baselines. Our evaluation compared GECO with four baselines: HLF [5], ZeeStar [42], GECO (w/o. CTM), and GECO (w/o. defense). HLF adopts the access-control approach to ensure data confidentiality and is one of the most popular blockchain frameworks in both industry and academia [24, 37, 38]. ZeeStar is a notable cryptography-based blockchain that encrypts sensitive data using exponential ElGamal encryption [32] (i.e., a PHE scheme) and generates Groth16 NIZKPs to ensure the correctness of transaction execution. GECO (w/o. CTM) solely implements 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 was *SmallBank* [27], a widely used benchmark for evaluating

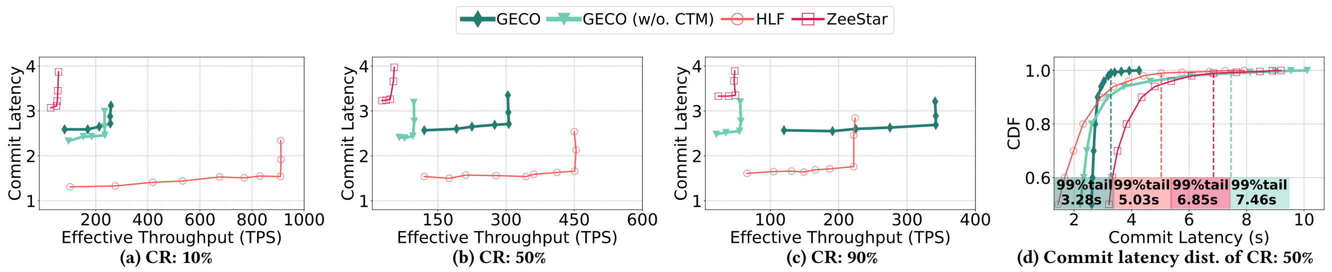


Figure 5: End-to-end performance of GECO and baselines in benign environments with different conflict ratios.

blockchain performance [37, 40]. SmallBank simulates a banking scenario and provides various functionalities like token transfer. In §6.1, we used SmallBank to evaluate the performance in benign environments, where all participant nodes and clients adhered to their system protocols. In §6.2, we used SmallBank to evaluate the performance in malicious environments, where a malicious lead executor conducted the NIZKP omission attacks as discussed in §4.3.

Our second workload consisted of ten distinct smart contracts that involve FHE addition and multiplication, as exemplified in Table 6. In §6.3, we evaluated GECO and GECO (w/o. CTM) with these smart contracts in order 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*, which represents the average number of client transactions per second (TPS) that are commited to the blockchain, excluding any aborted transactions; and (2) *commit latency* (also known as *end-to-end latency*), which measures the time duration from client submission to transaction commitment. In addition, we reported the 99th percentile commit latency and presented the cumulative distribution function (CDF) of the commit latency.

Testbed. 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. The average node-to-node RTT was about 0.2 ms.

Settings. For each system, we created five organizations, each consisting of 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 both HLF and GECO-based systems, we created four orderers running the BFT-SMaRT [9] consensus protocol.

We developed a distributed benchmark tool based on Tape [29] and eth-tester [17]. Tape is an efficient benchmark toolkit for EOV blockchains, while eth-tester is used by ZeeStar to simulate transactions according to the authors [42]. We ran each experiment ten times and reported the average values of the metrics. Note that our tool distributed the clients across multiple servers to avoid the bottleneck of transaction submission from a single server.

To simulate real-world scenarios 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?

System	Average Latency (s)					99% tail latency (s)
	P	E	O	V	E2E	
GECO	0.75	0.85	0.89	0.23	2.72	3.28
GECO (w/o. CTM)	0.47	0.87	0.88	0.24	2.46	7.46
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

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.

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 these benign environments, the network remained stable, and all participants behaved correctly. 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 *SendPayment* transactions. In each transaction, a sender client transferred a certain number of tokens to a receiver client. Note that both clients were randomly selected by the benchmark tool, and these clients may belong to different organizations. For aborted transactions, the benchmark tool repeatedly submitted them until they were successfully commited to the blockchain.

GECO achieved high effective throughput, outperforming both ZeeStar and GECO (w/o. CTM) with increases of 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. We explain these results in two aspects. Firstly, GECO performed correlated merging on conflicting transactions as specified by the CTM protocol (§4.2). Although GECO incurred an 11% extra latency compared to GECO (w/o. CTM), the CTM protocol empowered GECO with noticeable throughput gains of up to 6.2×, and also significantly reduced the conflicting aborts, thereby lowering the 99% tail latency. Secondly, GECO proved the correctness of transaction execution by using the consistency check NIZKPs (§4.2) that were independent of the smart contract logic, greatly reducing the latency of NIZKP generation. In contrast, ZeeStar relied on the inefficient re-execute approach (§1) for NIZKP generation. This resulted in a shorter latency of GECO compared to ZeeStar, as confirmed in Table 7.

GECO was less performant than HLF due to the additional overhead required to satisfy GECO's confidentiality guarantee. The overhead can be attributed to three aspects. Firstly, GECO introduced the preprocessing phase, which involved merging correlated transactions and encrypting transaction parameters. Secondly, during the execution phase, GECO performed compute-intensive FHE arithmetic computation and NIZKP generation. Lastly, in the validation phase, GECO was required to verify the validity of the consistency check NIZKPs. We believe that these overheads are necessary and practical in order to fulfill GECO's confidentiality guarantee.

GECO is well-suited for scenarios with conflicting transactions. Figure 5 shows that as the conflict ratio increased, GECO (w/o. CTM), ZeeStar, and HLF suffered from a noticeable decrease in throughput. However, GECO demonstrated an increase in throughput, performing even better than in scenarios with fewer conflicts. Furthermore, GECO outperformed HLF with 1.54x higher throughput in the scenario with 90% conflict ratio, despite the additional overhead for satisfying GECO's confidentiality guarantee. This was attributed to the CTM protocol, which effectively minimized transaction conflicts, leading to fewer conflicting aborts and reducing invocations of expensive cryptographic primitives.

Overall, GECO offers a compelling combination of both high performance and confidentiality guarantee. These qualities collectively make GECO particularly suitable for applications that prioritize data confidentiality and involve conflicting transactions.

6.2 Robustness to Malicious Participants

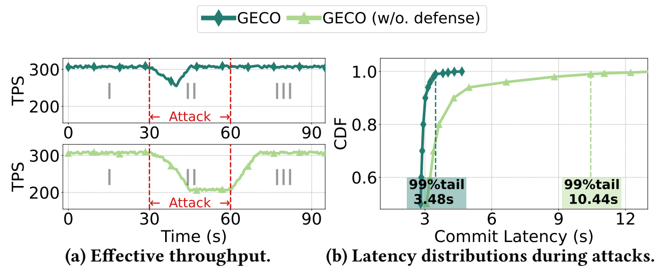


Figure 6: Performance under malicious participants.

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

The defense mechanism against the NIZKP omission attack is effective. Figure 6a shows that both GECO and GECO (w/o. defense) experienced performance degradation at the onset of the attack. However, GECO quickly resumed to its pre-attack throughput level, whereas GECO (w/o. defense)'s throughput remained at a low level until we terminated the attack. Figure 6b shows that the GECO's tail latency remained largely unaffected during the attack. In contrast, GECO (w/o. defense) experienced significant degradation in tail latency because the transactions involving O_M were repeatedly re-submitted and aborted. These differences were attributed to GECO's defense mechanism (§4.3), which successfully detected the malicious lead executor of O_M and rejected the transactions involving O_M for a long duration. Overall, the results in Figure 6 demonstrate that GECO's defense mechanism against the NIZKP omission attack is both necessary and effective.

6.3 Performance under Diverse Applications

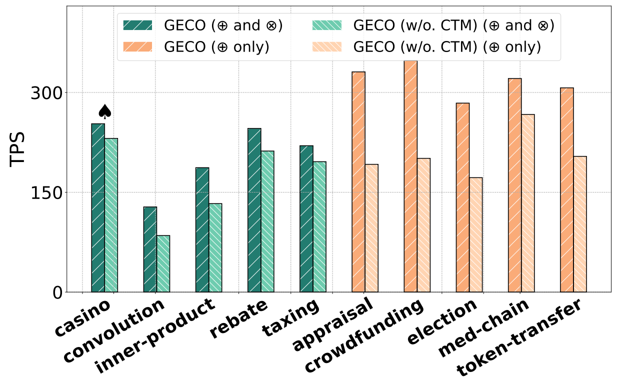


Figure 7: Throughputs of diverse smart contracts in Table 6. ⊕ and ⊗ denote FHE addition and multiplication, respectively. ♠ denotes non-deterministic smart contracts.

We evaluated GECO and GECO (w/o. CTM) using the second workload, which consisted of ten distinct smart contracts (Table 3) representing typical use cases of GECO. Our objective was to measure the end-to-end performance of GECO and the performance improvement provided by the CTM across various applications.

As shown in Figure 7, GECO exhibited high performance in all evaluated smart contracts, achieving throughput ranging from 120 TPS (`convolution`) to 348 TPS (`crowdfunding`). Notably, the CTM protocol delivered noticeable improvements in throughput, particularly in smart contracts where transaction conflicts were prevalent, such as `election` and `token-transfer`. Furthermore, we observed that the smart contracts exclusively performing FHE addition generally achieved higher throughput compared to those involving both FHE addition and multiplication. This difference can be attributed to the higher computational overhead associated with FHE multiplication compared to addition.

Overall, GECO achieves high performance for general smart contracts while also providing data confidentiality. We believe that GECO can bring benefits to various industries, including but not limited to finance, gaming, and healthcare.

7 CONCLUSION

We present GECO, a high-performance confidential permissioned blockchain framework for general smart contracts. GECO tolerates non-deterministic contracts and supports general cryptographic primitives. GECO ensures the correctness of transaction execution by efficiently generating non-interactive zero-knowledge proofs that are agnostic to contract logic. Moreover, GECO enhances performance by proposing the CTM protocol to effectively tackle the conflicting abort issue. Extensive evaluation demonstrates that GECO achieves superior performance compared to existing baselines while providing confidentiality guarantee, making GECO an ideal choice for a wide range of blockchain applications that desire both high performance and data confidentiality.

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[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] Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I 23*, pages 111–131. Springer, 2018.

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

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

[12] 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.

[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] consensys. Gnark, 2023.

[15] 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.

[16] 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.

[17] Ethereum. Ethereum/eth-tester: Tool suite for testing ethereum applications., 2023.

[18] 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.

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

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

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

[22] The Hyperledger Foundation. Private data collections: A high-level overview – hyperledger foundation, Oct 2018.

[23] 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.

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

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

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

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

[33] 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.

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

[35] NCCGroup, 2016.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

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