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

Anonymous authors

Abstract

Data confidentiality is essential for blockchain applications handling sensitive data. 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 cannot support arbitrary arithmetic operations over ciphertext because they use operation-restricted cryptographic primitives such as partial homomorphic encryption.

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-organization trusted execution mechanism of execute-order-validate blockchains such as Hyperledger Fabric, GECO generates contract logic-agnostic NIZKPs, which prove only that quorum organizations produce consistent results, thereby ensuring the contract execution correctness while addressing potential result non-determinism. For better performance, GECO merges multiple conflicting transactions into a single one, minimizing the occurrence of conflicting aborts and invocations of cryptographic primitives. Theoretical analysis and extensive evaluation on notable contracts demonstrate that GECO achieves strong confidentiality guarantee among existing systems and supports general smart 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 widely favored in both industry and academia. The main spur is their support of smart contracts that enable trusted execution over a tamper-resistant ledger shared among mutually untrusted participants. However, notable blockchains like Ethereum [48] process and store client data in plaintext, raising deep concerns over data confidentiality. Such concerns are particularly problematic for applications involving highly sensitive information like financial data [5], as they are subject to stringent privacy laws, such as the General Data Protection Regulation of the European Union [47]. For example, a typical non-deterministic contract for token exchange (see Listing 1 and Table 1) executes over plaintexts, leaving significant vulnerability for data breach.

```
Exchange(id A, id B, tt X, tt Y, num V) {
  if (!HasID(A) or !HasID(B)) {Abort();}
  num W = V × GetLER(X, Y);
  num AX = GetState(A|X); num BY = GetState(B|Y);
  if (AX < V or BY < W) {Abort();}
  num AY = GetState(A|Y); num BX = GetState(B|X);
  PutState(A|X, AX − V); PutState(B|X, BX + V);
  PutState(A|Y, AY + W); PutState(B|Y, BY − W);
}
```

Listing 1: An example contract that client A exchanges V units of token X with client B for token Y, denoted as $A|X \overset{V}{\leftrightarrow} B|Y$. The code highlighted in red and brown is non-confidential and non-deterministic, respectively.

Existing work attempts to achieve data confidentiality in smart contracts and falls into two approaches. The first is the architecture approach, which designs dedicated blockchain architectures for data confidentiality. However, this approach imposes *strong trust assumptions* on either specific hardware or third-party contract executors. For example, Ekiden [13] uses trusted execution environments (TEE), while Arbitrum [29], Caper [2], and Qanaat [3] are vulnerable to malicious executors from different organizations, as these executors execute transactions over plaintexts and may disclose client data.

The second is the cryptography approach. It employs cryptographic primitives like homomorphic encryption (HE) to encrypt client data *without extra trust assumptions* and to mandate smart contracts to execute directly on ciphertexts [43–45], preventing sensitive data exposure to malicious participants.

Data type	Description
`id`	Represent a client's unique identifier.
`tt`	Represent a token type.
`cnum`	Represent a plaintext number, storing different ciphertexts encrypted for different organizations.
Function	**Description**
`HasID(id)`	Check the existence of the given client `id`
`GetLER(`x,y`)`	Get the live exchange rate of token type x and y.
`GetState(key)`	Read a key-value pair from the state database.
`PutState(key,val)`	Write a key-value pair to the state database.
`HAdd`$(\mathtt{ct}_1, \mathtt{ct}_2)$	Perform FHE addition $\mathtt{ct}_1 + \mathtt{ct}_2$.
`HSub`$(\mathtt{ct}_1, \mathtt{ct}_2)$	Perform FHE subtraction $\mathtt{ct}_1 - \mathtt{ct}_2$.
`HMul`$(\mathtt{ct}_1, \mathtt{ct}_2)$	Perform FHE multiplication $\mathtt{ct}_1 \times \mathtt{ct}_2$.

Table 1: Data types and functions used in Listing 1 and 2.

This approach uses non-interactive zero-knowledge proofs (NIZKPs) to ensure the correctness of results (in ciphertext), without leaking any plaintext involved. Specifically, existing work generates NIZKPs by a *re-execution method*, in which NIZKPs decrypt transaction inputs and results, then re-execute transactions using the decrypted inputs, and verify the consistency between the decrypted and re-executed results.

However, because of the re-execution method, the existing cryptography approach still face two intrinsic limitations. Firstly, existing work is incompatible with non-deterministic contracts, which adopts non-deterministic language features like multithreading [38]. However, these contracts may produce inconsistent results under the same input and initial state in different executions, leading to incompatibility with NIZKPs generated by the re-execution method, which implies that identical input will always produce the same result.

Secondly, the existing cryptography approach suffers from poor performance due to the re-execution method. This method is inefficient in NIZKP generation, especially when integrated with general cryptographic primitives such as fully homomorphic encryption (FHE), which allows arbitrary arithmetic computations over ciphertexts. To illustrate, executing the BFV scheme [11, 21] (a popular FHE scheme) inside the elliptic curve-based NIZKP [16] takes tens of seconds to generate a NIZKP [42]. Even when employing lightweight cryptographic primitives such as partial homomorphic encryption (PHE), which restricts arithmetic operations, the NIZKP operations remain cumbersome. For instance, generating a Groth16 [35] NIZKP using 2048-bit keys for the Paillier PHE scheme [36], consumes more than 256 GB of memory [43], which is impractical for commodity desktops available today.

Overall, we believe that the re-execution method of the existing cryptography approach for generating NIZKPs is the root cause of its inability to support 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-assign the responsibility for ensuring the correct execution of contracts between blockchains 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

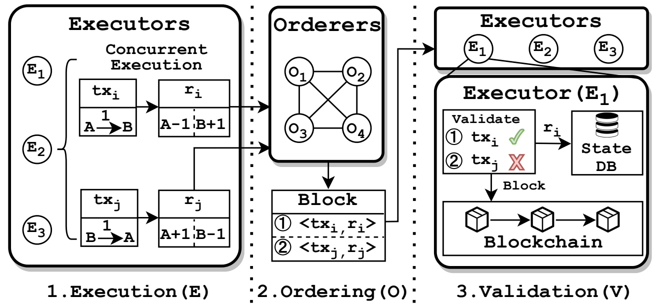


Figure 1: For high performance, EOV executors concurrently executes transactions tx_i and tx_j, causing conflicting aborts.

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, thereby 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 contracts. A general smart contract can be non-deterministic (e.g., multi-threaded) and/or involve general cryptographic primitives such as FHE to support arbitrary arithmetic computations. 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 a trusted manager, 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. The lead executor belongs to the client's organization, and consequently, GECO imposes no additional trust assumptions on the original trust model of EOV. 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, rather than re-executing the transaction inside NIZKPs as in existing re-execution method. By leveraging the CEOV workflow, GECO effectively preserves data confidentiality and ensures the execution correctness 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 Figure 1 shows, when multiple transactions con-

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

currently write to the same key-value pair in the blockchain state database, only one transaction can be committed (e.g., tx_i); other conflicting transactions are all aborted (e.g., tx_j). This challenge is exacerbated in GECO due to the costly cryptographic primitives, which prolong the execution latency and thus increase the likelihood of conflicting aborts. For example, Microsoft SEAL [40], a highly optimized FHE library, takes 2822 milliseconds to execute a single FHE multiplication [34]. Thus, conflicting aborts significantly degrade performance on executor nodes, wasting valuable computational resources.

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 to minimize conflicting aborts. Specifically, conflicting transactions invoking the same contract generally write to the same keys. By merging conflicting transactions into a single one, we can commit all transactions without abortion and preserve their original semantics. Consider a scenario where, for key X, two transactions attempt to perform $X+1$ and $X+2$ respectively. CTM merges and commits the transactions into a single transaction that performs $X+3$.

CTM consists of *offline analysis* and *online scheduling*. 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-write sets of transactions within each block to identify and merge the mergeable transactions. CTM is especially suitable for contracts involving ciphertexts, as it effectively reduces the waste of computing resources caused by conflicting aborts and the invocation of costly cryptographic primitives.

We implemented GECO on HLF [4], a prominent EOV permissioned blockchain framework. We integrated GECO with Lattigo [34], a performant FHE library, and Gnark [15], a popular NIZKP library. We evaluated GECO using both the SmallBank workload [28] under different conflict ratios, and ten blockchain applications, including both deterministic and non-deterministic contracts. We compared GECO with HLF and ZeeStar, a notable confidentiality-preserving blockchain of the cryptography approach [43]. Our evaluation shows that:

- GECO is efficient (§6.1). GECO achieved up to 7.14× higher effective throughput and 14% lower end-to-end latency than ZeeStar. While GECO had 32% lower 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 attack 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 employing general cryptographic primitives (e.g., FHE), rendering GECO more secure than existing cryptography-based confidentiality-preserving blockchains. CTM boosts the end-to-end performance by minimizing both the conflicting aborts of EOV blockchains and the high overhead of cryptographic primitives. Overall, GECO can benefit blockchain applications that desire both data confidentiality and high performance, such as supply chain [19] and healthcare [32]. GECO can also attract traditional non-deterministic applications developed in general programming languages like Go to be deployed upon. GECO's code is available at https://github.com/osdi25geco/osdi25geco.

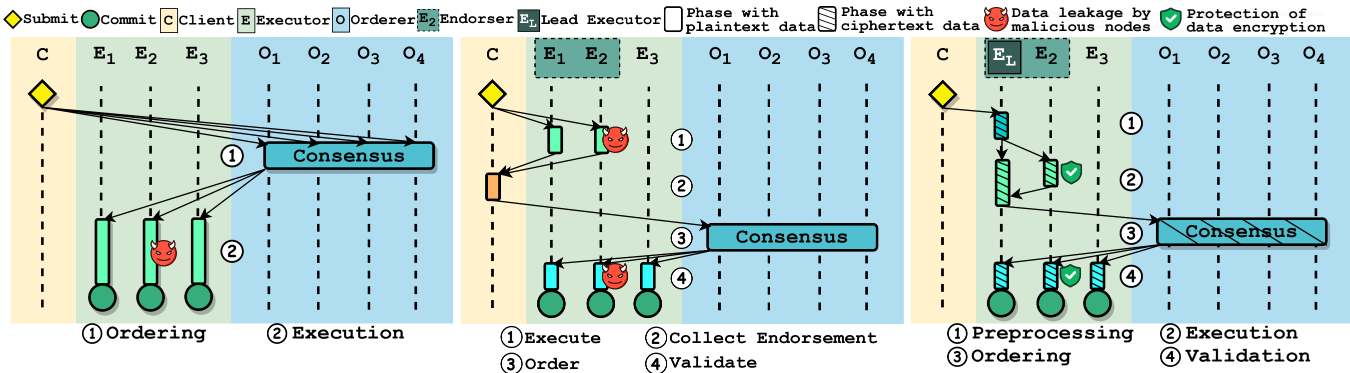
2 Background

2.1 EOV Permissioned Blockchain

A blockchain is a distributed ledger that records transactions across multiple nodes. It is either *permissionless* or *permissioned*. Permissionless blockchains (e.g., Ethereum [48]) 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 authenticated participants that are trusted within their organizations.

Permissioned blockchains have two main workflows: *order→execute* (OE) and *execute→order→validate* (EOV). As Figure 2a shows, in the OE workflow, the orderers establish a global transaction order (**O**), and then all executors execute the transactions in the predetermined order (**E**). As the OE workflow does not ensure the consistency of execution results, this workflow cannot tolerate non-deterministic smart contracts. However, non-deterministic smart contracts are prevalent and favorable as they can achieve higher performance and realize non-deterministic semantics via language features like multithreading [38] and random number generation [23].

The EOV workflow (Figure 2b) incorporates a **quorum-based trusted execution mechanism** to achieve high performance and tolerate non-deterministic transactions. 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 produces consistent execution results. The EOV workflow has three phases. **Execution (E)**: Executors execute transactions concurrently, generating a read set (keys read and their versions) and a write set (intended new values). The client verifies the result consistency. If consistent, the client forwards the results to the ordering service. **Ordering (O)**: The ordering service uses a consensus protocol (e.g., PBFT [12]) to decide the sequence of transactions in each block. **Validation (V)**: Executors validate transactions using multi-version concurrency control (MVCC), ensuring the read set matches the database state. Valid transactions update the database, while mismatches lead to transaction aborts.



(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.

This conflict issue, common in write-intensive smart contracts (e.g., supply chains [38]), impacts performance.

GECO leverages EOV's quorum-based trusted execution to avoid the computationally intensive NIZKP generation that is based on re-execution method, enabling support for general smart contracts while maintaining high performance. Moreover, GECO proposes CTM to minimize the conflicting aborts via merging conflicting transactions, thereby further enhancing the performance.

2.2 Homomorphic Encryption

GECO encrypts sensitive client data using homomorphic encryption (HE), which allows direct computation over ciphertexts without decryption. An HE scheme has four algorithms: *KeyGen* generates key pairs used by 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 $Enc(m_1, pk, r_1) \oplus Enc(m_2, pk, r_2) = Enc(m_1 + m_2, pk, r_3)$ for some randomness r_3. An HE scheme is either partial homomorphic encryption (PHE) or fully homomorphic encryption (FHE). Although PHE schemes have relatively low computation overhead, they only support 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. This flexibility enables the support for smart contract as any computation can be expressed as a combination of these two operations. Furthermore, substantial improvements have been made to improve the performance of FHE schemes since the proposal of the first plausible FHE scheme [24], making FHE a promising and practical solution for supporting general smart contracts.

In this study, we use FHE to support general computations. Specifically, GECO encrypt clients' data with FHE schemes, thereby allowing GECO executors to perform both addition and multiplication directly over ciphertexts without disclosing their corresponding plaintexts during contract execution.

2.3 Non-Interactive Zero-Knowledge Proofs

GECO co-designs the non-interactive zero-knowledge proof (NIZKP) [22, 26] 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 a 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 (zkSNARK) [9, 27], which allows any arithmetic circuit ϕ and guarantees constant-cost proof verification in the size of ϕ, making it suitable for blockchain applications [6, 17, 44]. GECO uses the Gnark [15] implementation of the Groth16 [35] 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 2.

Architecture approach. Existing work attempts to ensure data confidentiality by adopting dedicated blockchain architectures. Ekiden [13] demands TEE hardware to process private data. Arbitrum [29] executes smart contracts in plaintext and imposes strong trust assumptions on smart contract executors that might be incentivized to disclose clients' sensitive data. Caper [2] and Qanaat [3] adopt dedicated data models that limits data access to authenticated organizations but still exposes one organization's data to other organizations.

Cryptography approach. Existing work of the cryptography approach uses HE to protect data confidentiality and NIZKP to

System	Data Encryption	Multiple Organizations	General Contract	High Performance
✧ Ekiden [13]	✓	✓	✗	✓
✧ Arbitrum [29]	✗	✓	✗	✓
✧ Qanaat [3]	✗	✓	✓	✓
✧ Caper [2]	✗	✓	✓	✓
♦ ZeeStar [43]	✓	✤	✗	✗
♦ Zapper [45]	✓	✗	✗	✗
♦ Hawk [31]	✓	✓	✗	✓
♦ SmartFHE [42]	✓	✓	✗	✗
♦ FabZK [30]	✓	✗	✗	✗
♦ **GECO**	✓	✓	✓	✓

Table 2: 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 organizations.

enforce the execution correctness of smart contracts. However, existing work does not support general smart contracts.

ZeeStar [43] uses an additive PHE scheme [33] and supports ciphertext multiplication by repeating additions. However, such multiplication is inefficient and cannot accept foreign values (i.e., ciphertexts encrypted by different organizations). Zapper [45] uses an oblivious Merkle tree construction and a NIZK processor for data confidentiality. Neither ZeeStar nor Zapper tolerate non-deterministic contracts due to their reliance on the OE workflow, as discussed in §2.1.

Hawk [31] uses symmetric encryption for data confidentiality and NIZKP for execution correctness of smart contracts. However, Hawk is dedicated only to money transfer applications and does not support general smart contracts.

SmartFHE [42] 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.

FabZK [30] adopts a specialized tabular ledger data structure to obfuscate the involved organizations. However, this data structure severely 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*, *orderer*, and *executor*. The latter two are referred to as *nodes*. GECO groups participants into *organizations*. Each organization runs multiple executors and orderers, and possesses a set of clients. We describe the functionalities of each participant type as follows:

Clients. Only clients can submit transactions to nodes for execution and commitment.

Orderers. GECO orderers determine the order of transactions in blocks via a consensus protocol (e.g., PBFT [12]).

Executors. Each executor is responsible for three tasks: executing transactions, validating results, and maintaining a local blockchain state database. As Figure 3 shows, each organization has a designated *lead executor*, which is built upon the existing Fabric Gateway [18] with the following extra 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.

No.	API	Description
	LEAD EXECUTOR APIS	
1	`Preprocess(tx)`	Preprocess the given transaction. (§4.2)
2	`Prove(rs)`	Generate NIZKPs (§4.2) to prove that quorum results are consistent.
3	`Verify(rs,nizkps)`	Verify NIZKPs (§4.2) 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 transaction if the predicate is false. (§4.2)

Table 3: libGECO APIs.

Threat model. GECO adopts the Byzantine failure model [7, 12], where orderers run a BFT consensus protocol that tolerates up to $\lfloor \frac{N-1}{3} \rfloor$ malicious orderers out of N orderers. Participants are grouped into organizations, and each organization forms one trust domain, where participants trust each other within the same organization and distrust any participants in other organizations. Note that GECO inherits the same threat model of existing EOV permissioned blockchains [4, 37, 38] and does not require extra trust assumptions. We make standard assumptions on cryptographic primitives, including FHE and NIZKP.

GECO's guarantees. GECO's *confidentiality* guarantee ensures that all client data are stored and processed in ciphertext, and the corresponding plaintext can be only decrypted by participants from the client's organization. GECO's *generality* guarantee ensures that GECO supports the provably correct execution of smart contracts that are non-deterministic and/or involve any cryptographic primitives such as FHE to support general computations.

3.2 libGECO

GECO provides a library named libGECO with five APIs (see Table 3). The lead executor APIs assist lead executors in preprocessing transactions and ensuring the execution correctness. The smart contract APIs enable contract developers to analyze the mergeability (Definition 4.3) 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 [43]. We will further discuss how to integrate libGECO into GECO's protocol in §3.4 and §4.2.

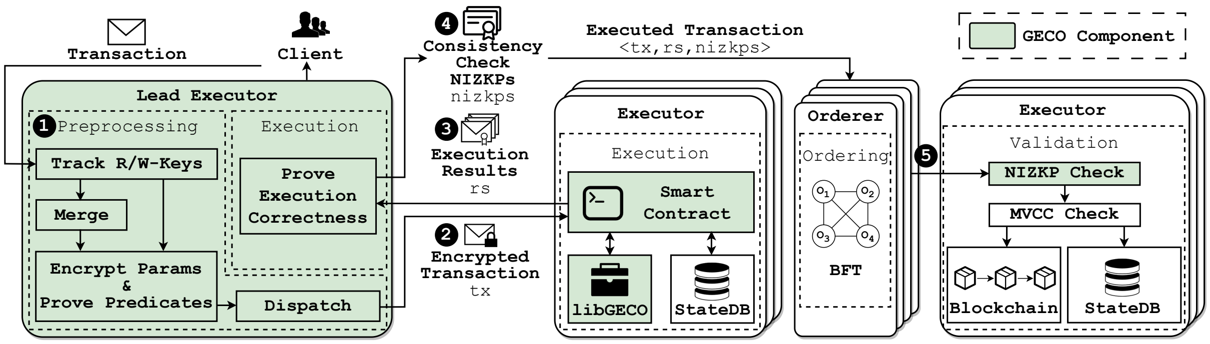


Figure 3: GECO's runtime workflow. GECO components are highlighted in green.

3.3 Example Smart Contract

Listing 2 and Table 1 illustrates a GECO smart contract for token exchange transfer. In contrast to Listing 1, Listing 2 achieves data confidentiality via executing smart contracts over ciphertexts, and tolerates non-determinism caused by external APIs (i.e., `GetLER`). Note that function `Add`, `Sub`, and `Mul` can be implemented with any FHE scheme, such as the CKKS scheme [14]. In this contract, the `Require()` API (Table 3) verifies a NIZKP that decrypts the ciphertexts involved (i.e., `AX`, `V`, `BY`, and `W`) 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 to other organizations, while the number of tokens being exchanged is known solely to the clients involved.

```
Exchange(id A, id B, tt X, tt Y, cnum V) {
  if (!HasID(A) or !HasID(B)) {Abort();}
  num W = Mul(V, GetLER(X, Y));
  cnum AX=GetState(A|X);cnum BY=GetState(B|Y);
  Require(AX>=V and BY>=W);            // Invoke API 5
  cnum AY=GetState(A|Y);cnum BX=GetState(B|X);
  PutState(A|X,Sub(AX,V));PutState(A|Y,Add(AY,W));
  PutState(B|X,Add(BX,V));PutState(B|Y,Sub(BY,W));
}
```

Listing 2: A GECO contract for token exchange that achieves data confidentiality and tolerates non-determinism. The code highlighted in green executes over ciphertexts.

3.4 GECO's Protocol Overview

GECO has two sub-protocols: (1) *offline contract analysis* and (2) *runtime transaction scheduling*.

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 4.3). For example, the contract in Listing 2 is *additively mergeable*: multiple conflicting transactions invoking this contract with identical `A`, `B`, `X`, and `Y` ($tx_1 : A|X \xrightarrow{1} B|Y$ and $tx_2 : A|X \xrightarrow{1} B|Y$) can be merged into one transaction ($tx_{1,2} : A|X \xrightarrow{2} B|Y$) by summing their `V`, while keeping others unchanged.

Runtime transaction scheduling (§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): it caches transactions submitted by clients from the same organization (§3.1), merges mergeable conflicting transactions, and leaves others unchanged. Next, the lead executor encrypts all transaction's plaintext inputs of data type `cnum` using the encryption keys of respective 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), an 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 `Prove()` API to generate NIZKPs, proving that quorum executors (i.e., a majority of executors) have produced consistent results, thereby ensuring execution correctness.

Phase 3: Ordering. The lead executors deliver the transactions with the NIZKPs to the 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.

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 `Verify()` 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.

The highlights of GECO stem from achieving data confidentiality and high performance for general smart contracts by integrating the EOV workflow with cryptography primitives, combining the best of both worlds while addressing their respective limitations. We illustrate this in two aspects.

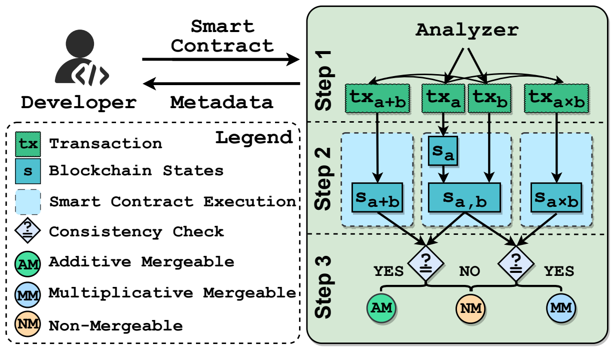


Figure 4: GECO's offline smart contract analysis protocol. tx_{a+b} and $tx_{a\times b}$ are formed by merging tx_a and tx_b via adding or multiplying their input parameters (§4.1). s_{a+b} and $s_{a\times b}$ are the states produced by executing tx_{a+b} and $tx_{a\times b}$. $s_{a,b}$ is the state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

Firstly, GECO co-designs cryptographic primitives and the EOV workflow to address the fundamental limitations faced by existing cryptography-based confidentiality-preserving blockchains in supporting general contracts. Specifically, GECO's CEOV workflow leverages 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 non-deterministic contracts. This also enables GECO to encrypt client data using cryptographic primitives that can perform arbitrary arithmetic computations over ciphertexts, such as FHE.

Secondly, GECO's CTM protocol simultaneously minimizes the invocations of costly cryptographic primitives and EOV's conflicting aborts by merging conflicting transactions without changing the original semantics.

4 Protocol Description

4.1 Offline Smart Contract Analysis

Definition 4.1 (Smart contract parameter types)**.** 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 derive the respective values.

Definition 4.2 (Equivalent smart contracts)**.** Two smart contracts c_1 and c_2 are *Equivalent* iff, for any transaction tx that invokes these two contracts , the transactions produce identical resulting states respectively, .

Definition 4.3 (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 *Equivalent* 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.

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.
$\mathtt{s}$, $\mathtt{s_a}$, $\mathtt{s_{a,b}}$, $\mathtt{s_{a+b}}$	States in blockchain database.
$\mathtt{tx_a}$, $\mathtt{tx_b}$, $\mathtt{tx_{a+b}}$	Transactions.
`paths`	The set of all control flow paths.
$\mathtt{p}$, $\mathtt{p_a}$, $\mathtt{p_b}$, $\mathtt{p_{a+b}}$	Control flow paths.
$\mathtt{n_{a+b}}$	The counter for tracking $\mathtt{s_{a,b}} = \mathtt{s_{a+b}}$.
`k`	Key parameters for the smart contract.
$\mathtt{v_a}$, $\mathtt{v_b}$	Value parameters for the smart contract.

Table 4: Functions and variables for Algorithm 1.

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. A *multiplicative mergeable* smart contract can be defined similarly. Otherwise, the contract is *non-mergeable*.

The *offline analysis* determines whether a contract is additive mergeable, multiplicative mergeable, or non-mergeable. The offline analysis outputs a *metadata*, 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.

The `Analyze()` API implements the offline analysis protocol for detecting *additive mergeable* contracts in three steps using symbolic execution. (see Figure 4, Algorithm 1, and Table 4). *Multiplicative mergeable* smart contract can be detected similarly. For a better performance, random interpretation [1] can also be used.

Before the iteration, the analyzer gets all control flow paths of the contract C by using symbolic execution. Meanwhile, for a contract C, the GECO analyzer first checks whether each parameter is a key or value parameter. If a parameter is used as a key in EOV's state access APIs, such as `GetState()` and `PutState()` (see Table 1), the analyzer records it as a key parameter; otherwise, it is recorded as a value parameter.

Analysis step 1: Preparation. In each loop, S, k, v_a, and v_b are generated to make sure that tx_a, tx_b, and tx_{a+b} can be executed on p_a, p_b, and p_{a+b}, respectively. Meanwhile, the analyzer generates a "*merged*" transactions, namely tx_{a+b}, based on tx_a and tx_b. For tx_{a+b}, its value parameters are the sums of the respective value parameters of tx_a and tx_b, while leaving the key parameters unchanged.

Analysis step 2: Random interpretation. GECO performs two 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

Algorithm 1: Offline analysis protocol for detecting additive mergeable contracts (§4.1; API 4)

1 $n_{a+b} \leftarrow 0$; paths $\leftarrow$ `GetAllCFPs(C)`;
2 **foreach** p_a **in** paths **do**
3   **foreach** p_b **in** paths **do**
4   **foreach** p_{a+b} **in** paths **do**
      // Step 1: Preparation
5       $p = \{p_a, p_b, p_{a+b}\}$;
6       $s \leftarrow$ `GenRandStateDB(C, p)`;
7       $k, v_a, v_b \leftarrow$ `GenRandParams(C, p)`;
8       $tx_a \leftarrow$ `GenTX`(k, v_a); $tx_b \leftarrow$ `GenTX`(k, v_b); $tx_{a+b} \leftarrow$ `GenTX`$(k, v_a + v_b)$;
      // Step 2: Execution
9       $s_a \leftarrow$ `ExecTX`(tx_a, s); $s_{a,b} \leftarrow$ `ExecTX`(tx_b, s_a); $s_{a+b} \leftarrow$ `ExecTX`(tx_{a+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 **if** $n_{a+b} = ($`GetSize`$(\text{paths}))^3$ **then**
13   **return** `AdditiveMergeable`
14 **else**
15   **return** `NonMergeable`

state $s_{a,b}$. In the second run, GECO separately executes tx_{a+b}, producing s_{a+b}.

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

Assumptions. An analyzable contract must satisfy two requirements. For analysis step 1, the read-write set must be identified in EOV's state access APIs before execution. For analysis step 2, the contract should not contain recursive contract calls. GECO can easily integrate more advanced analysis techniques to relax requirements on contracts. For non-analyzable contracts that do not satisfy these requirements, GECO conservatively regards these contracts as *non-mergeable*. Therefore, the non-mergeable contracts can at most degrade GECO's performance without compromising confidentiality and correctness.

Guarantees. Algorithm 1 will never determine non-mergeable smart contracts as mergeable. Supposing a smart contract C' is non-mergeable, then there exist parameters k', v'_a, and v'_b (line 7 of Algorithm 1) which can be used to generate transactions tx'_a, tx'_b, and tx'_{a+b} (line 8 of Algorithm 1) resulting states after execution $s'_{a,b}$ and s'_{a+b} (line 8 of Algorithm 1) are not equal. So analysis step 3 (line 9~10 of Algorithm 1)in this loop will fail. As a result, the total number of successful consistency checks will be less than the total number of enumerations, and the contract C' will be identified as non-mergeable in line 12~13 of Algorithm 1.

4.2 Runtime Transaction Scheduling

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.

Function	Description
`GetRWKeys(tx)`	Get all read and write keys of a transaction.
`DetectConflict(K,Q)`	Return mergeable and 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 block to the blockchain.
Variable	**Description**
`Q`	The transaction buffering queue.
`K`	All read and write keys for transactions in `Q`.
`T`	The threshold number of transactions in `Q`.
txs_m, txs_n	Mergeable and non-mergeable transactions.
txs_e	Encrypted transactions.

Table 5: Functions and variables for Algorithm 2.

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 $(tx : A \xrightarrow{val} B)$ that invokes the contract in Listing 2, 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 1). 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 2), 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.

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

```
// Phase 1: Preprocessing (Invoke API 1)
K ← ∅; Q ← ∅;
Upon reception of Transaction tx from client; do
    K ← K ∪ GetRWKeys(tx);
    Q ← Q ∪ tx;
Upon timeout or |Q| ≥ T
    txs_m, txs_n ← DetectConflict(K, Q);
    txs_m ← Merge(txs_m);
    txs_e ← Encrypt(txs_m, K) ∪ Encrypt(txs_n, K);
    ProvePredicates(txs_e);
    Dispatch(txs_e);
// Phase 2: Execution
Upon reception of Transaction tx from lead executor; do
    r ← Execute(tx);
    ReplyResult(tx, r);
// Phase 3: Ordering
Upon reception of all Results rs for Transaction tx; do
    nizkps ← Prove(rs);                    // Invoke API 2
    Order(tx, rs, nizkps);
// Phase 4: Validation
Upon reception of Block blk; do
    For tx, rs, nizkps in blk; do
        if Verify(rs, nizkps) ∧             // Invoke API 3
          CheckMVCC(rs) then
            Commit(tx, rs, nizkps);
        AppendBlock(blk);
```

The lead executor also generates NIZKPs for contracts that invoke the `Require()` API to validate specific predicates. For example, the contract in Listing 2 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 all endorsers, the lead executor ensures correct executions by proving that quorum endorsers produced consistent read-write sets. However, the read-write sets contain ciphertexts that cannot be directly compared as they involve random noise for security reasons [46]. To address this issue, the lead executor invokes the `Prove()` 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 to protect the confidentiality of the decryption key. In the case where 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 results for key *A*. Similarly, org_2 provides a NIZKP that different endorsers produce consistent results 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 endorsers 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 [12]) 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 `Verify()` 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 organization are mutually trusted (§3.1). For clients from other organizations, the participant can only access their ciphertext data but not the corresponding plaintexts.

While malicious participants cannot compromise GECO's confidentiality, they can still significantly degrade its 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 attack, 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 executor. 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

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). Thus, GECO inherits the mature consistency (safety) guarantee of existing BFT consensus protocols [8]. This ensures that all benign executors process and append the same blocks in the same order determined by the orderers.
- *State consistency*. Every benign executor follows the same deterministic protocol to validate and commit transactions, as demonstrated in Phase 4 of the runtime protocol (§4.2). A transaction's result is committed to the state database *iff* quorum results are consistent and does 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 [26]. 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 CKKS scheme [14]) achieves *indistinguishability under chosen-plaintext attack* (IND-CPA [10]), 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 corresponding to the ciphertext belonging to *org*'s clients. This is because org_1 keeps its decryption key confidential, and IND-CPA makes it computationally infeasible for the attacker to gain any information about the ciphertext without the decryption key. □

6 Evaluation

GECO implementation. We built GECO on HLF v2.5 [20]. GECO uses the CKKS scheme [14] implemented by Lattigo [34] and uses the Groth16 NIZKP [26] on the BN254 elliptic curve implemented by Gnark [15]. Same as existing systems [6,43] that use Groth16, GECO requires a circuit-specific setup to derive the verification key for each NIZKP circuit.

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	exchange ♠	✓	✓	Exchange two different types of tokens (see Listing 2).
3	rebate ♠	✓	✓	Calculate rebate for qualifying invoices based on ratio from non-deterministic external APIs.
4	dot-product	✓	✓	Compute the dot-product of two vectors of different clients.
5	taxing	✓	✓	Calculate and pay tax.
6	timed-pay ♠	✓		Pay the bill within the specified time, otherwise the bill is cancelled.
7	appraisal	✓		Appraise collectibles with confidential value estimates.
8	election	✓		Determine the winner in a two-candidate election.
9	med-chain	✓		Medicine inventory management.
10	p2pay	✓		Peer-to-peer payment.

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.

This setup is a one-time procedure with negligible overhead.

Baselines. We compared GECO with four baselines: HLF [4], ZeeStar [43], GECO (w/o. CTM), and GECO (w/o. defense). HLF is one of the most popular permissioned blockchain frameworks [25, 38, 39]. ZeeStar is a notable confidentiality-preserving blockchain that encrypts sensitive data using an additive PHE scheme [33] and generates Groth16 NIZKPs to ensure the execution correctness. GECO (w/o. CTM) implements only the CEOV workflow without integrating our CTM protocol (§4). GECO (w/o. defense) implements both the CEOV workflow and the CTM protocol, but lacks the defense mechanism against the NIZKP omission attack as specified in §4.3.

Workloads. We used two workloads for evaluation. The first one (§6.1 and §6.2) is *SmallBank* [28], a widely used benchmark for evaluating blockchain performance [38, 41]. SmallBank simulates the banking scenario and provides diverse contracts like token transfer. We evaluated the encrypted version of SmallBank, where each contract is re-written with libGECO APIs (see Table 3) and Gnark. Our second workload (§6.3) consists of ten contracts from existing work [42, 43, 45] that involve FHE arithmetic operations and non-determinism, as shown in Table 6. In §6.3, we evaluated GECO, GECO (w/o. CTM), and ZeeStar with these contracts to measure the performance gain of the CTM protocol in these ten applications such as gaming (`casino`) and finance (`exchange`).

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 HLF, GECO, GECO (w/o. CTM), and GECO (w/o. defense), we created four orderers running the PBFT [12] protocol. In §6.1 and §6.2, 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.

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.

Our evaluation focused on three primary questions.

§6.1 How efficient is GECO compared to baselines?

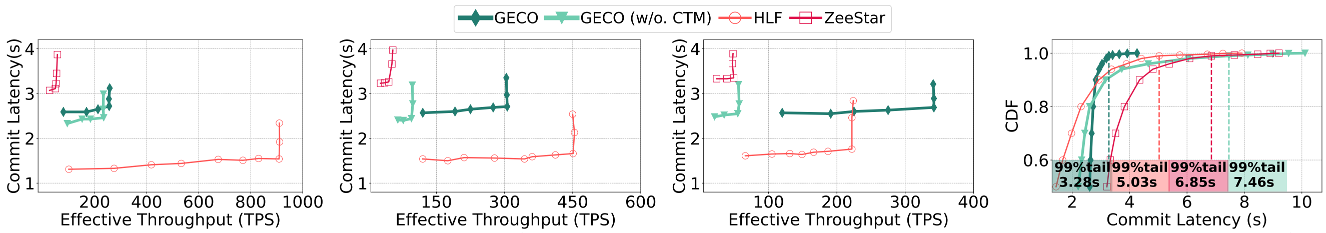
§6.2 How robust is GECO to malicious participants?

§6.3 How efficient is GECO under diverse applications?

6.1 End-to-End Performance

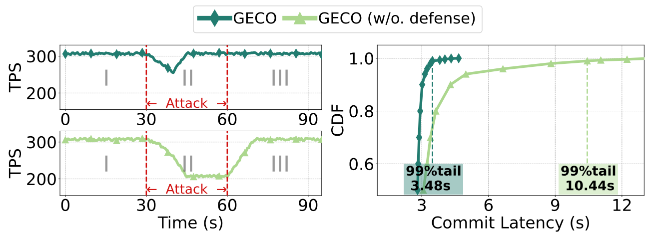
We first evaluated the end-to-end performance in benign environments on four systems: GECO, GECO (w/o. CTM), ZeeStar, and HLF. In the benign environment, the network was stable, and all participants were correct. We conducted three experiments using SmallBank with conflict ratios of 10%, 50%, and 90%, respectively. For each experiment, we generated a series of *Transfer* transactions (Listing 2). Each transaction transferred tokens from the sender client to the receiver client. Note that both clients were randomly selected and may belong to different organizations. For aborted transactions, we repeatedly submitted them until they were successfully committed.

GECO achieved up to 7.14× higher effective throughput and the shortest 99% tail latency, outperforming both ZeeStar and GECO (w/o. CTM). As Figure 5 shows, 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. This gap became more evident as the conflict ratio increased. ZeeStar performed even worse,



(a) CR: 10% (b) CR: 50% (c) CR: 90% (d) Commit latency dist. of CR: 50%

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



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

Figure 6: Performance under malicious participants.

achieving merely 54, 52, and 48 TPS, respectively.

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 offers no data confidentiality, GECO incurred an average performance overhead of 32%. This overhead is due to three aspects. Firstly, GECO has an extra preprocessing phase, which merges correlated transactions and encrypts inputs. Secondly, GECO involves costly FHE computation and NIZKP generation in the execution phase. Lastly, in the validation phase, GECO verifies the consistency check NIZKPs. We believe that these overheads are necessary and practical 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 abrupt throughput decrease. However, GECO exhibited a growth in throughput, and even surpassed HLF with a 1.54x higher throughput under 90% conflict ratio, despite the overhead of confidentiality guarantee. This was attributed to the CTM protocol (§4), which merges correlated transactions, thereby minimizing conflicting aborts and 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.

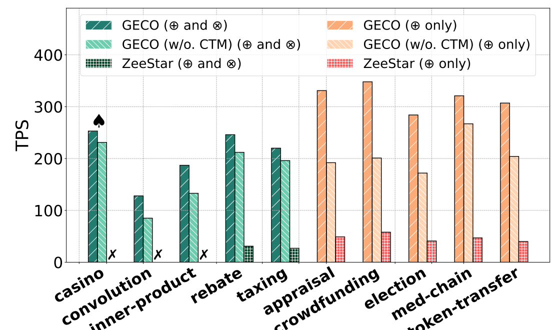


Figure 7: Throughputs of contracts in Table 6. ⊕ and ⊗ denote FHE addition and multiplication. ♠ is non-deterministic contracts. ✗ is contracts not supported by ZeeStar.

6.2 Robustness to Malicious Participants

We conducted NIZKP omission attack (§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 attack 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 [38] and bidding [43].

6.3 Performance under Diverse Applications

We evaluated GECO, GECO (w/o. CTM), and ZeeStar using the second workload (i.e., ten diverse smart contracts in Table 1). As Figure 7 shows, GECO exhibited high performance in all evaluated contracts, achieving throughput ranging from 120 TPS (`exchange`) to 348 TPS (`timed-pay`). The CTM protocol delivered notable throughput gains, especially in contracts where conflicts are prevalent, such as `election` and `p2pay`. Furthermore, GECO can support general applications with contracts that are non-deterministic (e.g., `casino`) and involve foreign values (e.g., `dot-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. Thanks to GECO's contract logic-agnostic trusted execution (§4.2), we can easily integrate various cryptographic primitives like BFV [11, 21] 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 boosts GECO's performance in contending applications with 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 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] Elli Androulaki, Jan Camenisch, Angelo De Caro, Maria Dubovitskaya, Kaoutar Elkhiyaoui, and Björn Tackmann. Privacy-preserving auditable token payments in a permissioned blockchain system. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 255–267, 2020.

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

[15] consensys. Gnark, 2023.

[16] 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.

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

[18] Hyperledger Fabric. Fabric gateway.

[19] 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.

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

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

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

[23] James E Gentle. *Random number generation and Monte Carlo methods*, volume 381. Springer, 2003.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[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] 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.

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

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

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

[35] 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] Zeshun Peng, Yanfeng Zhang, Qian Xu, Haixu Liu, Yuxiao Gao, Xiaohua Li, and Ge Yu. Neuchain: a fast per-

missioned blockchain system with deterministic ordering. *Proc. VLDB Endow.*, 15(11):2585–2598, July 2022.

[38] 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.

[39] 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.

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

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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