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

Anonymous authors

Abstract

Data confidentiality is essential for blockchain applications that handle sensitive data. A promising approach to achieving data confidentiality is executing smart contracts on ciphertexts and then verifying correctness through non-interactive zero-knowledge proofs (NIZKPs). However, existing solutions cannot tolerate non-deterministic (ND) contracts whose correct execution cannot be proven by NIZKPs, and suffer from poor performance for deterministic contracts because of using homomorphic encryption.

We present Geco, a confidentiality-preserving and high-performance permissioned blockchain framework for general smart contracts. Geco re-assigns the responsibility for ensuring execution correctness of contracts from cryptographic primitives to blockchain, thereby tolerating ND contracts. To support fully homomorphic encryption (FHE) without incurring significant performance overhead, we present Ctm protocol, which merges multiple conflicting transactions into a single one, minimizing conflicting aborts and invocations of cryptographic primitives. Theoretical analysis and extensive evaluation of various contracts demonstrate that Geco achieves a 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.

ACM Reference Format:
Anonymous authors. 2025. Geco: A Confidentiality-Preserving and High-Performance Permissioned Blockchain Framework for General Smart Contracts. In . ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 such as Ethereum [53] process and store client data in plaintext, raising deep concerns about data confidentiality. Such concerns are particularly problematic for applications involving highly sensitive information such as financial data [4], as they are subject to stringent privacy laws, such as the General Data Protection Regulation of the European Union [52]. For example, a typical token transfer contract (see Listing 1) executes in plaintexts, causing a data breach.

```
func Transfer(client A, client B, num V) {
  num AV = GetState(A); num BV = GetState(B);
  if (AV < V) {Abort();}
  RunThread({PutState(A, AV - V)}); // thread 1
  RunThread({PutState(B, BV + V)}); // thread 2
}
```

Listing 1. A token transfer contract where client A transfers V units of token to client B, denoted as $A \xrightarrow{V} B$. The data in red is non-confidential. The code in brown uses multithreading, a non-deterministic programming language feature.

Existing work attempting to achieve data confidentiality falls into two approaches. The first is the architecture approach, which designs dedicated blockchain architectures for data confidentiality and attains high performance as it does not involve costly cryptography primitives. However, this approach imposes *strong trust assumptions* on either hardware or third-party contract executors. For example, Ekiden [12] uses trusted execution environments (TEE), while Arbitrum [29], Caper [1], and Qanaat [2] are vulnerable to malicious executors from various organizations, as these executors execute transactions in plaintext 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 [47–49], preventing sensitive data exposure to malicious participants. This approach uses non-interactive zero-knowledge proofs (NIZKPs) to ensure the correctness of results (in ciphertext), without leaking any plaintext. Specifically, existing work generates NIZKPs using 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, due to the re-execution method, the existing cryptography approach still faces two intrinsic limitations. Firstly, existing work is incompatible with non-deterministic contracts written in general programming languages (e.g., Go) supporting non-deterministic language features like multithreading [41]. These contracts may produce inconsistent results under the same input and initial state in different executions, making them incompatible 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 [9, 22] (a popular FHE scheme) inside the elliptic curve-based NIZKP [16] takes tens of seconds to generate a NIZKP [46]. 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 [37] NIZKP using 2048-bit keys for the Paillier PHE scheme [38], consumes more than 256 GB of memory [47], 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 its 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 the blockchain and cryptographic primitives*. Instead of relying solely on NIZKPs, we can prove the correct execution of contracts by leveraging the quorum-based trusted execution of execute→order→validate (EOV) permissioned blockchains, such as Hyperledger Fabric (HLF) [3]. This mechanism first executes a transaction on multiple executor nodes (Figure 1). If a quorum of executors produces consistent results, the execution is considered correct; if the quorum fails to produce consistent results due to non-determinism, the transaction is aborted. By decoupling the correctness-proving logic from the contract logic, EOV can reduce the correctness-proving overhead for contracts involving expensive cryptographic primitives while tolerating non-deterministic contracts.

This insight leads to GECO[1], a confidentiality-preserving and high-performance permissioned blockchain framework

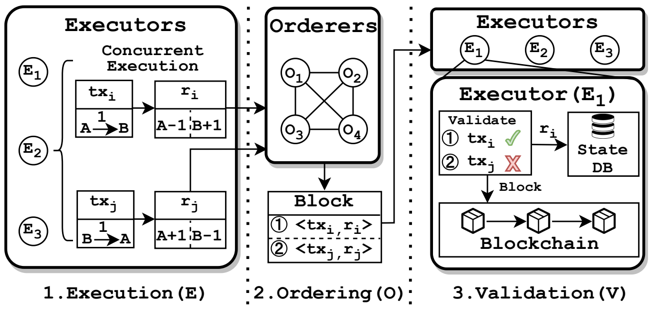


Figure 1. For high performance, EOV executors concurrently executes transactions tx_i and tx_j, causing conflicting aborts.

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 EOV (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., FHE), 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 concurrently execute the transaction over ciphertexts and return results to the lead executor. Thirdly, the lead executor decrypts the results and generates NIZKPs to prove that quorum results are consistent, rather than re-executing the transaction in NIZKPs as the existing re-execution method. With the CEOV workflow, GECO effectively preserves data confidentiality and ensures the correct execution of general smart contracts.

However, GECO still faces a performance challenge due to the transaction conflicting aborts of EOV. In EOV, executors concurrently execute transactions for high performance and check conflicts before committing transactions for serializability (i.e., optimistic concurrency control [3]). In Figure 1, when multiple transactions concurrently write to the same key-value pair in the blockchain state database, only one transaction can commit (e.g., tx_i); other conflicting transactions are all aborted (e.g., tx_j), causing performance degradation and wasted computational resources. The challenge is exacerbated in GECO due to the costly cryptographic primitives, which not only waste more computational resources on aborted transactions, but also prolong execution latency, increasing the likelihood of conflicting aborts. For example, Microsoft SEAL [44], a highly optimized FHE library, takes 2822 milliseconds to execute a single FHE multiplication [35].

To tackle the above challenge, we propose Correlated

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

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 abort 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 them into a single transaction that performs $X + 3$ without aborts.

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 [3], a prominent EOV permissioned blockchain framework. We integrated Geco with Lattigo [35], a performant FHE library, and Gnark [14], a popular NIZKP library. We evaluated Geco using both the SmallBank workload [28] under different conflict ratios, and ten blockchain applications, including deterministic and non-deterministic contracts. We compared Geco with HLF and ZeeStar, a notable confidentiality-preserving blockchain of the cryptography approach [47]. 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 a 32% lower throughput than HLF, HLF does not preserve confidentiality.
- Geco is general (§6.3). Geco tolerates non-deterministic contracts and supports cryptographic primitives like FHE which allows arbitrary computations over ciphertexts.
- Geco is robust (§6.2). Geco can maintain high effective throughput and low end-to-end latency under attack from malicious participants.

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), making Geco more secure than existing cryptography-based confidentiality-preserving blockchains. Ctm enhances 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 [20] and healthcare [33]. Geco can also attract non-deterministic applications developed in general programming languages like Go to be deployed on. Geco's code is available at https://github.com/eurosys26geco/eurosys26geco.

2 Background

2.1 EOV Permissioned Blockchain

A blockchain can be either *permissionless* or *permissioned*. Permissionless blockchains (e.g., Ethereum [53]) are open to anyone, and their participants are mutually untrusted. In contrast, permissioned blockchains (e.g., HLF [3]) are run by a group of explicitly identified organizations, and only grant access to participants that are trusted within their organizations.

Blockchains have two main workflows: *order→execute* (OE) and *execute→order→validate* (EOV). As Figure 2 shows, in the OE workflow, the orderers establish a global transaction order (**O**), and then all executors individually execute the transactions in this order (**E**). OE does not mandate the explicit identification of participants, rendering OE particularly suitable for permissionless blockchains [36, 53]. However, this leads to OE's inability to check the consistency of execution results on executors for tolerating non-deterministic contracts. Nonetheless, **non-deterministic contracts are prevalent and favorable** as they can achieve higher performance and realize non-deterministic semantics via language features like multithreading [41].

The EOV workflow (Figure 2) is designed for permissioned blockchains and 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. EOV 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 [11]) 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. This conflict issue, common in write-intensive smart contracts (e.g., supply chains [41]), harms performance.

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

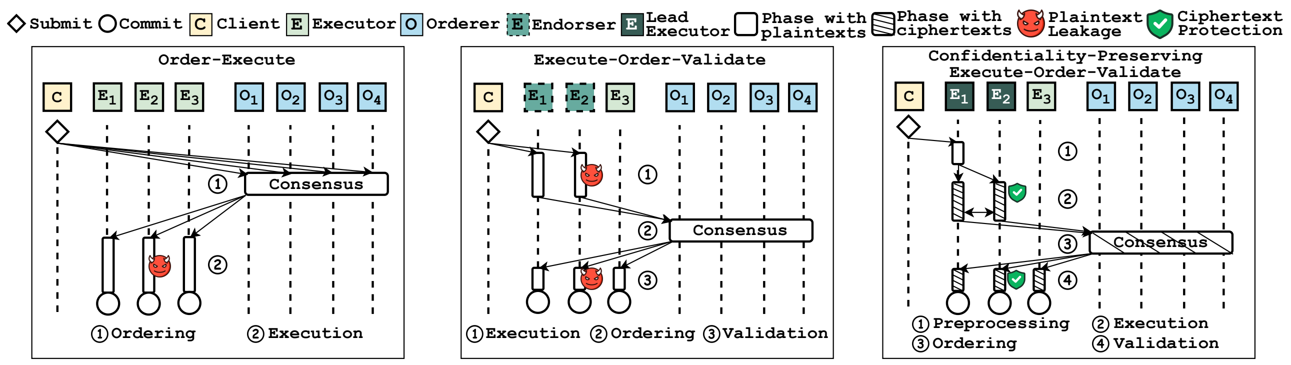


Figure 2. Our Ceov workflow and two existing typical permissioned blockchain workflows.

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)* uses the public key *pk* to encrypt a plaintext *m* into a ciphertext *c*; *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 producing a result ciphertext. For instance, in an additive HE scheme, the $\oplus$ operator satisfies $Enc(m_1, pk) \oplus Enc(m_2, pk) = Enc(m_1 + m_2, pk)$. 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 harms 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 general contracts as any computation can be expressed as a combination of these two operations. Moreover, 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 solutions for supporting general contracts.

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

2.3 Non-Interactive Zero-Knowledge Proofs

Geco co-designs the non-interactive zero-knowledge proof (NIZKP) [23, 26] with EOV workflow to ensure the correctness of transaction execution. NIZKP 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

System	Data Encryption	Multiple Organizations	General Contract	High Performance
✧ Ekiden [12]	✓	✓	✗	✓
✧ Arbitrum [29]	✗	✓	✗	✓
✧ Qanaat [2]	✗	✓	✓	✓
✧ Caper [1]	✗	✓	✓	✓
♦ ZeeStar [47]	✓	✤	✗	✗
♦ Zapper [49]	✓	✗	✗	✗
♦ Hawk [32]	✓	✓	✗	✓
♦ SmartFHE [46]	✓	✓	✗	✗
♦ FabZK [30]	✓	✗	✗	✗
♦ **Geco**	✓	✓	✓	✓

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

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) [8, 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, 18, 48]. Geco uses the Gnark [14] implementation of the Groth16 [37] system (a zkSNARK construction) with advantages of constant proof size (a few kilobytes) and short verification time (tens of milliseconds).

2.4 Related Work

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

Architecture approach. Existing work attempts to ensure data confidentiality by adopting dedicated blockchain architectures. Ekiden [12] 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 disclose clients' sensitive data. Caper [1] and Qanaat [2] adopt dedicated data models that limit data access to authenticated organizations but still expose one organization's data to other organizations.

Cryptography approach. Existing work of the cryptography approach uses HE to protect data confidentiality and NIZKP to enforce the execution correctness of smart contracts. However, existing work does not support general smart contracts.

ZeeStar [47] uses an additive PHE scheme [34] 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 [49] uses an oblivious Merkle tree construction and a NIZK processor for data confidentiality. Neither ZeeStar nor Zapper tolerates non-deterministic contracts due to their reliance on the OE workflow, as discussed in §2.1.

Hawk [32] uses symmetric encryption for data confidentiality and NIZKP for the correct execution of smart contracts. However, Hawk is dedicated only to money transfer applications and does not support general smart contracts.

SmartFHE [46] is OE-based and uses FHE schemes. It cannot support non-deterministic contracts and 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

As with 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 [11]).

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 [19] 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.
4	`Analyze(C)`	Run the offline analysis (§4.1) on the contract.
		Smart contract APIs
5	`Require(predicate)`	Abort transaction if the predicate is false. (§4.2)

Table 2. libGeco APIs.

Threat model. Geco adopts the Byzantine failure model [7, 11], 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. Each organization forms a 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 [3, 40, 41] and does not require extra trust assumptions. We make standard assumptions on cryptographic primitives, including FHE and NIZKP.

Geco's guarantees. (1) Geco's *confidentiality* guarantee ensures that the value of client data is stored and processed in ciphertext, while the key of client data and the contract code remain in plaintext, the same as existing work [32, 46, 47]. (2) Geco's *generality* guarantee ensures that Geco supports provably correct executions of smart contracts that are non-deterministic and/or involve any cryptographic primitives such as FHE to support arbitrary arithmetic computations. Geco employs lightweight NIZKPs (§3.4) to tolerate the random noise [9, 13, 22] in FHE ciphertexts, while using Geco's Ceov workflow to handle the non-determinism caused by language features like multithreading [41]. (3) Geco's *high performance* is defined as producing lightweight consistency-check NIZKPs for merged transactions, instead of producing re-execution-based NIZKPs for individual transactions (§1).

3.2 libGeco

Geco offers a library named libGeco with five APIs (see Table 2). The lead executor APIs enable lead executors to preprocess transactions, ensure execution correctness, and analyze contracts' mergeability (Definition 4.3). The smart contract API (i.e., `Require()`) ensures the validity of predicates with ciphertexts and follows the idea of the *required* statement of ZeeStar [47]. For example, the `Require()` API in Listing 2 enforces that client `A` have enough tokens (i.e., the predicate $AV \geq V$ is true). We will further discuss how to integrate libGeco into Geco's protocol in §3.4 and §4.2.

3.3 Example Smart Contract

```
func Transfer(client A, client B, cnum V) {
  cnum AV = GetState(A); cnum BV = GetState(B);
  Require(AV >= V);
  RunThread({PutState(A,HSub(AV,V))}); // thread 1
```

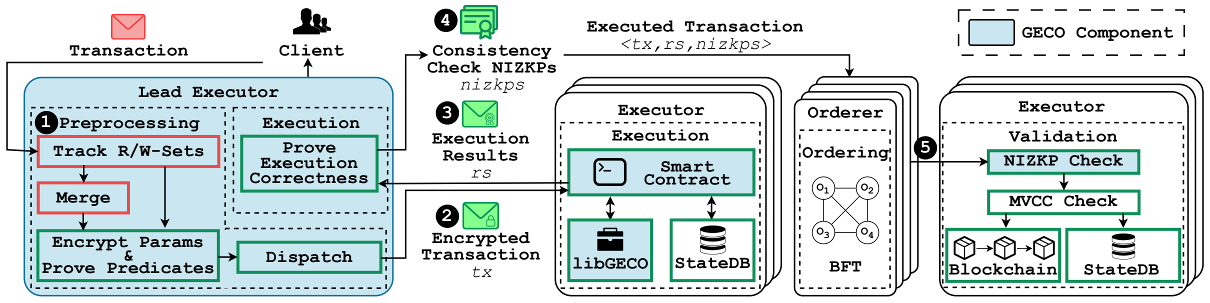


Figure 3. GECO's runtime workflow. Plaintext and ciphertext are highlighted in red and green, respectively.

Data type	Description
`client`	Represent a client's identity.
`cnum`	Represent a plaintext number, storing different ciphertexts encrypted for different organizations.
Function	**Description**
`GetState(key)`	Read a key-value pair from the state database.
`PutState(key,val)`	Write a key-value pair to the state database.
`HAdd(ct_1,ct_2)`	Perform FHE addition $ct_1 + ct_2$.
`HSub(ct_1,ct_2)`	Perform FHE subtraction $ct_1 - ct_2$.
`RunThread(func)`	Run the given function in a separate thread.

Table 3. Data types and functions used in Listing 2. Note that the FHE functions like HAdd are implemented by smart contract developers and can adopt various FHE schemes.

```
  RunThread({PutState(B,HAdd(BV,V))}); // thread 2
}
```

Listing 2. A GECO contract for token transfer. It achieves data confidentiality via ciphertexts (i.e., the data in green) and uses multithreading for better performance.

Listing 2 and Table 3 illustrate a GECO smart contract for token transfer. In contrast to Listing 1, Listing 2 achieves data confidentiality by executing contracts over ciphertexts. Listing 2 also tolerates non-determinism caused by multithreading. For instance, multiple threads concurrently updating a value might produce different results. Note that functions HAdd and HSub can be implemented with any FHE scheme, such as the BFV scheme [9, 22]. In this contract, client data AV and V are ciphertexts. They cannot be compared due to random noises and cannot be directly decrypted during execution to avoid exposing plaintexts. Therefore, to ensure the predicate $AV \geq V$, executors invoke the `Require()` API (Table 2) to verify the NIZKP that confidentially decrypts AV and V 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 secret from other organizations, while the number of tokens being transferred is solely known by the clients involved.

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). During contract deployment, the lead executor invokes the `Analyze()` API on the contract to perform 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 and B ($tx_1 : A \xrightarrow{1} B$ and $tx_2 : A \xrightarrow{2} B$) can be merged into one transaction ($tx_{1,2} : A \xrightarrow{3} B$) 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 of the client's organization invokes the `Preprocess()` API to preprocess transactions. 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 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 [3], 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

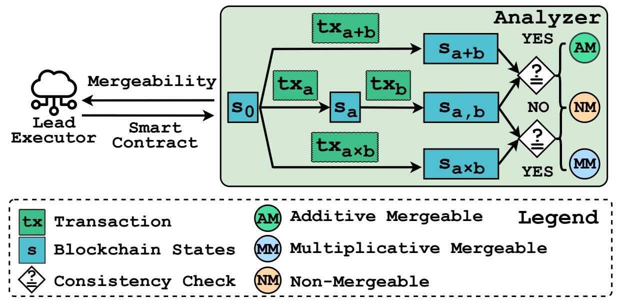


Figure 4. GECO's offline smart contract analysis protocol. tx_{a+b} and $tx_{a\times b}$ are formed by merging tx_a and tx_b via adding or multiplying their input parameters (§4.1). s_{a+b} and $s_{a\times b}$ are the states produced by executing tx_{a+b} and $tx_{a\times b}$. $s_{a,b}$ is the state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

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 fusing EOV workflow with cryptography primitives, combining the best of both worlds while addressing their limitations. We illustrate this in two aspects. Firstly, GECO co-designs cryptographic primitives and EOV workflow to address the fundamental limitations faced by existing cryptography-based confidentiality-preserving blockchains in supporting general contracts. Specifically, GECO's CEOV workflow uses the quorum-based trusted execution of EOV blockchains to generate lightweight NIZKPs, which are agnostic to contract logic and prove only 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 minimizes both the invocations of costly cryptographic primitives and EOV's conflicting aborts by merging conflicting transactions without altering 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 transactions). Two transactions are *equivalent* iff they invoke the same contract and produce identical resulting states irrespective of the initial state.

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.

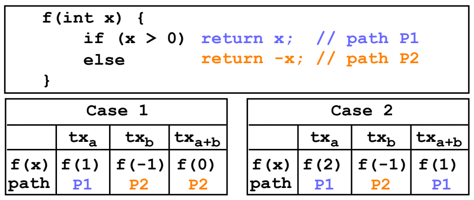
A smart contract is *additive mergeable* iff it is mergeable and generates new value parameters by adding the respective value parameters of the conflicting transactions. A *multiplicative mergeable* smart contract can be defined similarly. Otherwise, the contract is *non-mergeable*. The *offline analysis* determines whether a contract is additive mergeable, multiplicative mergeable, or non-mergeable. During runtime, lead executors carry out transaction merging solely on additive mergeable and multiplicative mergeable contracts. The `Analyze()` API implements the offline analysis protocol, as shown in Figure 4. Without loss of generality, we illustrate the `Analyze()` API via the process of detecting additive mergeable contracts.

For each contract C, the analyzer first labels the key and value parameters. If a parameter is used as a key in EOV's state access APIs like `GetState()` (see Table 3), the analyzer labels it as a key parameter; otherwise, it is labeled as a value parameter. Then, the analyzer uses symbolic execution [5, 15, 31] tool KLEE [10] with the theorem prover Z3 [54] to get all execution paths of C. As Algorithm 1 and Table 4 show, for the correctness of offline analysis, the analyzer traverses all combinations of execution paths p_a, p_b, and p_{a+b}. Specifically, each loop runs the following three steps.

Analysis step 1: Preparation. For each combination of execution path p_a, p_b, and p_{a+b}, the analyzer tries to generate an initial state s, key parameters k, and value parameters v_a and v_b using KLEE. These state and parameters make transactions tx_a, tx_b, and tx_{a+b} execute on p_a, p_b, and p_{a+b}, respectively. Specifically, tx_{a+b} takes the sums of the value parameters of tx_a and tx_b, while taking the same key parameters. Note that the prover (Z3) of KLEE may indicate the combination as *unsatisfied* or *unknown* [50]. An *unsatisfied* combination matches no qualified initial state and parameters and hence can be safely skipped. An *unknown* combination cannot be analyzed by the prover due to non-determinism or excessive path depth. Therefore, we conservatively regard the contract with *unknown* combination of execution paths as *non-mergeable*.

Analysis step 2: Execution. GECO performs two independent runs of C on the same initial state. In the first run, GECO executes tx_a and produces s_a, then executes tx_b on s_a to produce $s_{a,b}$. In the second run, GECO executes tx_{a+b}, producing s_{a+b}.

Analysis step 3: Consistency check. Lastly, GECO checks the consistency of the two resulting states. If the equality



```
f(int x) {
    if (x > 0) return x;  // path P1
    else       return -x; // path P2
}
```

Case 1	tx_a	tx_b	tx_{a+b}
f(x)	f(1)	f(-1)	f(0)
path	P1	P2	P2

Case 2	tx_a	tx_b	tx_{a+b}
f(x)	f(2)	f(-1)	f(1)
path	P1	P2	P1

Figure 5. An example showing the necessity of traversing all combinations of path p_a, p_b, and p_{a+b} in Algorithm 1.

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

```
paths ← GetAllExecutionPaths(C);
foreach $p_a$ in paths do
    foreach $p_b$ in paths do
        foreach $p_{a+b}$ in paths do
            // Step 1: Preparation
            p ={$p_a$, $p_b$, $p_{a+b}$ };
            s ← GenRandStateDB(C, p);
            k, $v_a$, $v_b$ ← GenRandParams(C, p);
            $tx_a$ ← GenTX(k, $v_a$); $tx_b$ ← GenTX(k, $v_b$);
              $tx_{a+b}$ ← GenTX(k, $v_a + v_b$);
            // Step 2: Execution
            $s_a$ ← ExecTX($tx_a$, s); $s_{a,b}$ ← ExecTX($tx_b$, $s_a$);
              $s_{a+b}$ ← ExecTX($tx_{a+b}$, s);
            // Step 3: Consistency check
            if $s_{a,b} \neq s_{a+b}$ then
                return NonMergeable
return AdditiveMergeable
```

$s_{a,b} = s_{a+b}$ does not hold, C is non-mergeable. After traversing all combinations, the contract C is additive mergeable.

Evaluating all combinations of execution path p_a and p_b without considering p_{a+b} is insufficient for correctly detecting mergeable contracts. As shown in Figure 5, case 1 and case 2 both runs tx_a (executed on path P1) and tx_b (executed on path P2). However, tx_{a+b} of case 1 follows path P2, while tx_{a+b} of case 2 follows path P1. This discrepancy highlights that even if the execution paths of tx_a and tx_b are identical, tx_{a+b} might follow different execution paths. Thus, for the correctness of offline analysis, it is necessary to traverse all combinations of execution paths p_a, p_b, and p_{a+b} in Algorithm 1.

Assumptions. An offline-analyzable contract must satisfy three requirements. Firstly, the read-write set must be identified in EOV's state access APIs before execution. Secondly, the execution path depth of the contract cannot exceed the predefined limit, because symbolic execution cannot handle state-space explosions with unlimited depths, which includes recursive contract calls and infinite loops. Thirdly, there must be no non-determinism during execution, because there are many sources of non-determinism difficult to analyze offline. We resolve them during runtime just as HLF does. For contracts not satisfying these requirements, GECO conservatively regards these contracts as *non-mergeable*. Therefore, non-mergeable contracts can at most degrade GECO's performance without compromising confidentiality and correctness.

Variable	Description
s, s_a, $s_{a,b}$, s_{a+b}	States in blockchain database.
tx_a, tx_b, tx_{a+b}	Transactions.
paths, p, p_a, p_b, p_{a+b}	Execution paths.
k	Key parameters of the smart contract.
v_a, v_b	Value parameters of the smart contract.

Table 4. Variables for Algorithm 1.

Guarantee. Algorithm 1 never determines an offline-analyzable non-mergeable smart contracts as mergeable.

Discussion. Our CTM protocol, which merges transactions with identical key parameters, is highly suitable for diverse blockchain applications. For instance, contracts that transfer funds between corporate accounts [43] often execute many transactions between identical pairs of accounts. Besides, contracts that log IoT sensor data [39] often execute multiple data-logging transactions within one block for each sensor. Our CTM protocol can effectively resolve the conflicting aborts prevailing in these applications, ensuring high performance.

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.

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

Phase 1.1: Tracking read-write sets. For each block, the lead executor buffers the transactions in a queue and keeps track of read-write set of each transaction . For example, for a transaction ($A \xrightarrow{V} B$) 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 queued transactions exceeds a threshold, the lead executor detects conflicting transactions according to their read-write sets. Next, the lead executor runs our CTM protocol (Phase 1.2) to merge mergeable transactions 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 a multiplicative mergeable contract, respectively.

Phase 1.3: Encrypting transaction inputs and proving predicates. The lead executor encrypts the transaction inputs of data type `cnum` (see Table 3). The lead executor analyzes the read and write keys associated with the `cnum` inputs, then encrypts the input plaintexts using the encryption keys of the clients specified by the read and write keys. For example,

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

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

for transaction $A \xrightarrow{V} B$, the cnum input V has two write keys A and B that belong to org_1 and org_2, respectively. Thus, the lead executor generates two ciphertexts for V by encrypting V using org_1's and org_2's public keys, respectively.

The lead executor also generates NIZKPs for contracts invoking the Require() API to validate predicates. For example, the contract in Listing 2 requires $AV \geq V$. Thus, the lead executor of org_1 generates a NIZKP that decrypts the ciphertexts (AV and V) and proves the predicates. The NIZKP, along with the keys and versions of its ciphertexts, is attached to the transaction 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, the endorser invokes the contract specified by the transaction. For contracts using the Require() API, the executor verifies the NIZKPs generated in Phase 1.3 and checks if the latest ciphertext versions match the version attached to the NIZKPs. If any checking fails, the executor aborts the transaction execution. 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 returns it to the lead executor.

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, read-write sets contain ciphertexts that cannot be directly compared as they involve random noise for security reasons [51]. To address this issue, the lead executor invokes the Prove() API to generate a *consistency check NIZKP* to compare whether two ciphertexts correspond to the same plaintext. For example, when comparing ciphertexts c_1 and c_2, the lead executor generates a NIZK circuit $C(x, y) = (dec(x, sk) == dec(y, sk))$, where sk is the secret key and dec is decryption function, and checks if $C(c_1, c_2) == 1$. In this part, plaintexts($dec(c_1, sk)$ and $dec(c_2, sk)$) are not revealed.

In the case where read-write sets contain ciphertexts of multiple organizations, the lead executors of those organizations all follow the same procedure to prove the consistency of the results. For example, 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 and org_3 provide NIZKPs for key B and C. All of these NIZKPs are submitted for ordering, collectively forming the correctness proof of tx.

Note that a result of a transaction 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 confirm the consistency of the execution results of tx.

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

Phase 4: Validation. Upon receiving a block from orderers, the executor sequentially validates each transaction. This validation process involves two steps: (1) invoking the 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 or-

Variable	Description
`Q`	The transaction buffering queue.
`K`	All read-write sets for transactions in `Q`.
`T`	The threshold number of transactions in `Q`.
$\texttt{txs}_\texttt{m}$, $\texttt{txs}_\texttt{n}$	Mergeable transactions and non-mergeable transactions.
$\texttt{txs}_\texttt{e}$	Encrypted transactions.

Table 5. Variables for Algorithm 2.

ganization 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 jeopardize 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 aborted in Phase 4. The attack is detrimental in GECO as it substantially wastes the computation resources of executors, especially when 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

Theorem 5.1 (Correctness)**.** Let σ_0 be the initial blockchain state, and let $\mathcal{B}_0, \ldots, \mathcal{B}s$ be the sequence of agreed blocks ordered by serial number. Let $\mathcal{T}_{\leq s}$ denote the sequence of all valid transactions in these blocks.

Assuming all benign executors start from σ_0 and execute transactions per protocol GECO, they will converge to the same state σ_s such that: $\sigma_s = \mathsf{ApplySeq}(\sigma_0, \mathcal{T}_{\leq s})$. This ensures both BFT safety (state agreement) and ACID correctness (sequential equivalence).

Proof. GECO preserves the underlying BFT consensus protocol [3], ensuring that all benign executors receive and process the same sequence of blocks $\mathcal{B}_0, \ldots, \mathcal{B}_s$ in identical order (block consistency).

Each executor applies only valid transactions in each block, as determined by deterministic validation rules. Transactions are executed sequentially in the order assigned by the orderers, ensuring:

- Atomicity: Transactions (or merged transaction groups by CTM) are either fully committed or fully aborted.
- Consistency: Only valid transactions are committed, preserving invariants.
- Isolation: Executors do not observe or depend on intermediate states of concurrent transactions.
- Durability: Committed changes are persistent and irreversible.

Since all benign executors start from the same state σ_0 and apply the same valid transaction sequence $\mathcal{T} \leq s$ in the same order, they reach an identical final state $\sigma_s = \mathsf{ApplySeq}(\sigma_0, \mathcal{T} \leq s)$. □

5.2 Liveness

Theorem 5.2 (Liveness)**.** Let tx be a valid transaction submitted by a benign client during the ordering phase. Then, under eventual synchrony and assuming a bounded number of Byzantine orderers, GECO guarantees that tx will eventually be included in a committed block $\mathcal{B}_s$ for some $s \in \mathbb{N}$.

Proof. GECO preserves the BFT liveness property of the underlying EOV permissioned blockchain [3], where orderers execute a BFT consensus protocol (e.g., PBFT).

Given eventual synchrony and that fewer than one-third of orderers are Byzantine, the consensus protocol guarantees that any valid transaction tx submitted by a benign client will eventually be: 1. included in some committed block $\mathcal{B}_s$, and 2. delivered to all benign executors for execution.

Hence, tx will not be indefinitely delayed or lost, satisfying liveness. □

5.3 Confidentiality

Theorem 5.3 (Confidentiality)**.** Let pt be a plaintext belonging to a client of organization Org, and let $ct = \mathsf{Enc}_{\mathrm{Org}}(pt)$ be its ciphertext under Org's public key. Then, for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ from a different organization $\mathrm{Org}^* \neq \mathrm{Org}$, the probability that $\mathcal{A}$ can recover pt from ct is negligible in the security parameter λ.

Proof. GECO enforces confidentiality through a combination of access control and IND-CPA secure encryption throughout all phases of the CEOV workflow(Figure 2).

- In Phase 1 (Preprocessing), plaintext data pt are only accessed by the lead executor of the client's organization Org. According to the system model (§3.1), the lead executor and its clients are mutually trusted, and no other entity—including adversaries from Org^*—has access to pt at this stage.
- In Phases 2–4, GECO operates solely on encrypted data. All client inputs are encrypted as $ct = \mathsf{Enc}_{\mathrm{Org}}(pt)$ using an encryption scheme (e.g., BFV [9, 22]) that satisfies IND-CPA security.

Formally, IND-CPA implies that for any PPT adversary $\mathcal{A}$, the advantage in distinguishing encryptions of two chosen

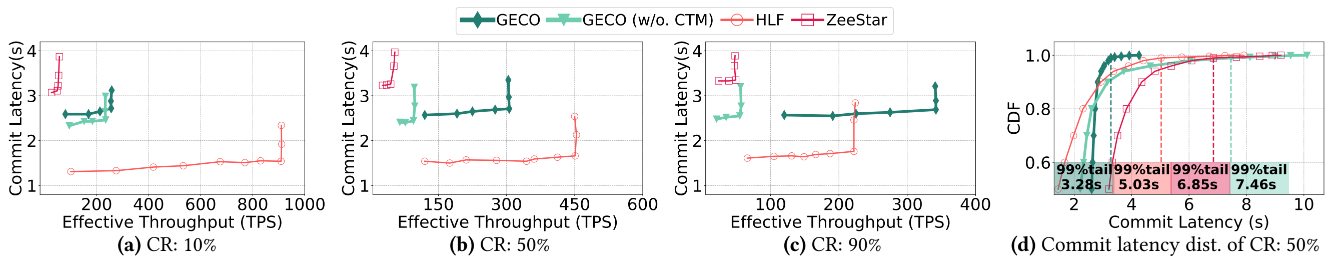


Figure 6. End-to-end performance of Geco and baselines in benign environments with different conflict ratios.

plaintexts pt_0, pt_1 is negligible:

$$\left| \Pr\left[\mathcal{A}(\mathsf{Enc}_{\mathrm{Org}}(pt_b)) = b\right] - \frac{1}{2} \right| \le \mathrm{negl}(\lambda),$$

where $b \leftarrow 0, 1$ is chosen uniformly at random, and λ is the security parameter.

Because the decryption key is held solely by Org, an adversary $\mathcal{A}$ from Org* cannot distinguish $ct = \mathsf{Enc}_{\mathrm{Org}}(pt)$ from an encryption of any other plaintext, nor decrypt ct to recover meaningful information. This holds even when Geco generates NIZK proofs for encrypted state consistency. Therefore, the adversary's probability of learning pt from ct is negligible:

$$\forall \mathcal{A} \in \mathrm{PPT}, \quad \Pr[\mathcal{A}(ct) \mapsto pt] \le \mathrm{negl}(\lambda).$$

Hence, Geco guarantees data confidentiality across all phases for clients of Org against adversaries from Org*. □

6 Evaluation

Geco implementation. We built Geco on HLF v2.5 [21]. Geco uses the BFV scheme [9, 22] implemented by Lattigo [35] and uses the Groth16 NIZKP [26] on the BN254 elliptic curve implemented by Gnark [14]. Same as existing systems [6, 47] that use Groth16, Geco requires a circuit-specific setup to generate the verification key for each NIZKP circuit. This setup is a one-time procedure with negligible overhead.
Baselines. We compared Geco with four baselines: HLF [3], ZeeStar [47], Geco (w/o. Ctm), and Geco (w/o. defense). HLF is one of the most popular permissioned blockchain frameworks [25, 41, 42]. ZeeStar is a notable confidentiality-preserving blockchain that encrypts sensitive data using an additive PHE scheme [34] 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 [41, 45]. SmallBank simulates the banking scenario and provides diverse contracts like token transfers. We evaluated the encrypted version of SmallBank, where each contract is rewritten with libGeco APIs (see Table 2) and Gnark. Our second workload (§6.3) consists of ten smart contracts from existing work [46, 47, 49] that involves HE arithmetic operations and non-determinism, as shown in Table 7.

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

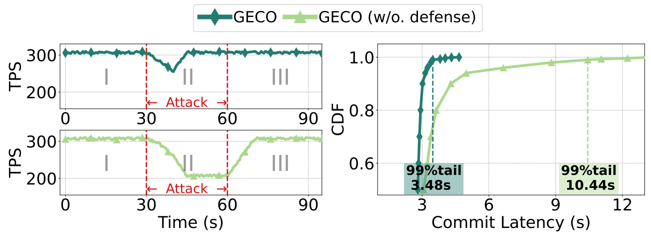
Table 6. Each phase's latency and the 99% tail latency of Figure 6b. The letters "P", "E", "O", and "V" denote the preprocessing, execution, ordering, and validation phases, respectively. The term "E2E" denotes the end-to-end latency.

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. Same as HLF [3], 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 [11] protocol. Same with existing work [45], 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.

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?

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 types of token based on the exchange rate obtained from a non-deterministic external API.
3	rebate ♠	✓	✓	Calculate invoice rebate based on ratio from non-deterministic external APIs.
4	dotprod	✓	✓	Compute the dot-product of two vectors of different clients.
5	taxing	✓	✓	Calculate and pay tax.
6	timedpay ♠	✓		Pay the time-limited bill based on the non-deterministic local clock of each executor.
7	appraisal	✓		Appraise collectibles with confidential value estimates.
8	election	✓		Determine the winner in a two-candidate election.
9	medchain	✓		Medicine inventory management.
10	transfer	✓		Token transfer (see Listing 2).

Table 7. Example smart contracts used in the second workload. ⊕ and ⊗ indicate whether the contract uses HE addition or multiplication, respectively. ♠ indicates that the contract is non-deterministic.



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

Figure 7. Performance under malicious participants.

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 benign. 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 might 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 6 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, 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 6.

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 6 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 a 90% conflict ratio, despite the overhead of confidentiality guarantee. This is 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.

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 the 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 7a 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 7b shows

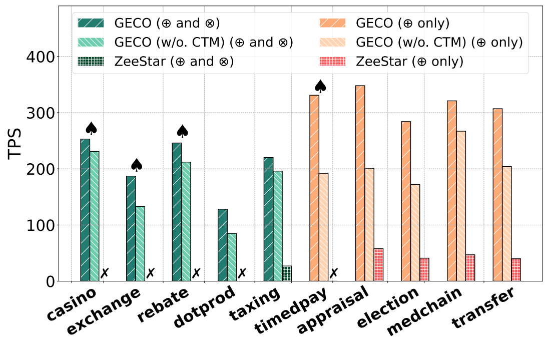


Figure 8. Throughputs of contracts in Table 7. "⊕ and ⊗" is contracts using both HE addition and multiplication. "⊕ only" is contracts using HE addition only. ✗ is contracts unsupported by ZeeStar. ♠ is non-deterministic contracts.

that GECO's tail latency barely changed during the attack (compared to Figure 6d). 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 such as supply chain [41] and bidding [47].

6.3 Performance under Diverse Applications

We evaluated GECO, GECO (w/o. CTM), and ZeeStar using the second workload (Table 3). As Figure 8 shows, GECO exhibited high performance in all evaluated contracts, even though GECO carries out FHE addition of the BFV scheme that is at least 14.8× slower than the PHE scheme adopted by ZeeStar, as confirmed in [17, 35]. GECO achieved throughput ranging from 120 TPS (`dotprod`) to 348 TPS (`appraisal`). The CTM protocol delivered notable throughput gains, especially in contracts with high conflict rates, such as `election` and `transfer`. Furthermore, GECO can support general applications with contracts that are non-deterministic (e.g., `casino`) and involve foreign values (e.g., `dotprod`). 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 employ various cryptographic primitives like the CKKS scheme [13] 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 correct transaction execution by efficiently generating contract-logic-agnostic NIZKPs. Our CTM protocol effectively enhances GECO's performance in contending applications by reducing conflicting aborts and invocations of costly cryptographic primitives. Extensive evaluation demonstrates that GECO achieves performance superior to that of baselines, making GECO an ideal choice for broad blockchain applications that desire data confidentiality and high performance.

References

[1] 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.
[2] 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.
[3] 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.
[4] 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.
[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.
[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] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 326–349, New York, NY, USA, 2012. ACM.
[9] 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.
[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
[11] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, New York, NY, USA, 1999. ACM.
[12] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.
[13] 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.

[14] consensys. Gnark, 2023.

[15] P David Coward. Symbolic execution systems—a review. *Software Engineering Journal*, 3(6):229–239, 1988.

[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] Thi Van Thao Doan, Mohamed-Lamine Messai, Gérald Gavin, and Jérôme Darmont. A survey on implementations of homomorphic encryption schemes. *The Journal of Supercomputing*, pages 1–42, 2023.

[18] 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.

[19] Hyperledger Fabric. Fabric gateway.

[20] 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.

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

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

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

[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] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[32] 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.

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

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

[35] 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.

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

[37] NCCGroup, 2016.

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

[39] Jianli Pan, Jianyu Wang, Austin Hester, Ismail Alqerm, Yuanni Liu, and Ying Zhao. Edgechain: An edge-iot framework and prototype based on blockchain and smart contracts. *IEEE Internet of Things Journal*, 6(3):4719–4732, 2019.

[40] Zeshun Peng, Yanfeng Zhang, Qian Xu, Haixu Liu, Yuxiao Gao, Xiaohua Li, and Ge Yu. Neuchain: a fast permissioned blockchain system with deterministic ordering. *Proc. VLDB Endow.*, 15(11):2585–2598, July 2022.

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

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

[43] Fabian Schär. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review*, 2021.

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

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] Pascal Strebel. Implementation of a cloud-based mobile application for time tracking. Bachelor's thesis, ETH Zurich, 2023. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Pascal_Strebel_BA_Report.pdf.

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

2021. IEEE.

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

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

[54] The z3 theorem prover (release 4.3.13). https://github.com/Z3Prover/z3, October 2024. Microsoft Research, Redmond, WA.