University of Hong Kong

MPhil Thesis

Achieving High-Performance and Confidentiality-Preserving Workflow for Permissioned Blockchains

Author:
Rongxin Guan

Supervisor:
Prof. Heming Cui

A thesis submitted in fulfillment of the requirements for the degree of Master of Philosophy

in the

Computer Science
Engineering

March 13, 2025

Abstract of thesis entitled

"Achieving High-Performance and Confidentiality-Preserving Workflow for Permissioned Blockchains"

Submitted by

Rongxin Guan

for the degree of Master of Philosophy

at The University of Hong Kong

in March, 2025

Permissioned blockchains gain significant traction in both industry and academia. This owes to the support for general smart contracts, which enable the trusted and auditable execution of general-purpose programs on a tamper-proof ledger that is collectively maintained by a group of explicitly identified yet mutually untrusted parties. This makes permissioned blockchains a compelling data storage solution for many real-world applications, such as peer-to-peer payment. However, typical permissioned blockchains store data in plaintext and impose weak or even no data access controls, which seriously undermines data confidentiality. As permissioned blockchains assume a central role in handling highly sensitive information, achieving efficient confidentiality preservation becomes an imperative for permissioned blockchains.

Existing work has primarily focused on two approaches to designing high-performance and confidentiality-preserving permissioned blockchains. The first approach involves designing new blockchain architectures that rely on extra trust assumptions, such as trusted hardware or third-party software managers. However, the first approach is limited by the fact that the required trusted hardware is not universally available, and there is a significant risk of potential data disclosure. In contrast, the second approach, based on cryptography, utilizes cryptographic primitives like homomorphic encryption (HE) to safeguard on-chain data. Notwithstanding, existing work of the second approach suffers from low performance caused by the high computation overhead of cryptographic primitives, hindering its practical adoption.

This thesis presents three pioneering permissioned blockchain systems, Geco, Gafe, and Gacm, which harmonize the two aforementioned approaches while addressing their existing limitations. To eliminate reliance on extra trust assumptions, Geco employs end-to-end encryption of on-chain data and generates non-interactive zero-knowledge proofs (NIZKPs) to validate the execution correctness of transactions involving ciphertexts. Geco minimizes computation overhead by validating the equality of quorum execution results computed by independent nodes, rather than re-executing the transaction logic within NIZKPs, which are significantly slower than normal execution. Furthermore, Geco proposes a novel concurrency control protocol that identifies and merges

correlated transactions, resulting in a reduction in transaction conflicting aborts and thereby an improvement in overall end-to-end performance.

In contrast to GECO's software-only method, GAFE and GACM leverage the superior parallel processing capability of GPU hardware to enhance the performance of confidentiality-preserving blockchains that employ fully homomorphic encryption (FHE) for data encryption. GAFE carries a GPU-accelerated transaction execution workflow that (1) parallelizes the internal arithmetic operations of individual FHE operators and (2) concurrently executes multiple transactions using multiple GPU threads. GACM proposes the FHE Transaction Merging (FTM) protocol to reduce the number of computationally intensive FHE transactions and the Three-Dimension GPU Acceleration (TDGA) protocol to accelerate FHE transaction execution from three distinct dimensions using GPU.

In conclusion, the three innovative systems tackle the fundamental challenges that have long baffled the development of high-performance confidentiality-preserving general permissioned blockchains. Through extensive testing on large-scale server cluster, the three systems have consistently demonstrated exceptional performance advantages and the effectiveness of their workflows in unlocking the practical adoption of confidentiality-preserving permissioned blockchains.

(479 words)

Achieving High-Performance and Confidentiality-Preserving Workflow for Permissioned Blockchains

by

Rongxin Guan

M.Phil. *HKU*

A Thesis Submitted in Partial Fulfilment of the Requirements for
the Degree of Master of Philosophy
at University of Hong Kong

March, 2025

For Family and Friends

Declaration

I, Rongxin Guan, declare that this thesis titled, "Achieving High-Performance and Confidentiality-Preserving Workflow for Permissioned Blockchains", which is submitted in fulfillment of the requirements for the Degree of Master of Philosophy, represents my own work except where due acknowledgement have been made. I further declare that it has not been previously included in a thesis, dissertation, or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Signed ..... Rongxin Guan .....

Acknowledgements

I extend my sincere gratitude to Prof. Heming Cui, Prof. Siu-Ming Yiu, Prof. Bruno Oliveira, and Prof. Reynold Cheng for their consistent support and invaluable counsel throughout my MPhil's journey. Their exceptional expertise and enlightening suggestions have been pivotal in developing my research skills and cultivating my tenacious attitude towards life. I am deeply appreciative of the significant impact they have made on my academic journey.

I am deeply indebted to Ji Qi, Qi Hu, Bocheng Xiao, Siyuan Wen, Junming Wang, Xian Wang, Wei Chen, Yu Tian, Songxiao Guo, Yuxing Pei, Dong Huang, Tianxiang Shen, Xusheng Chen, Yuhao Qing, Guichao Zhu, Zekai Sun, Xiuxian Guan, Haoze Song, and my fellow researchers for their precious dedication and extensive expertise to the enrichment of my research. Moreover, I appreciate all the friends I have made from diverse backgrounds around the world during my time in HKU and Chi Sun College. All the happiness we have shared, the challenges we have overcome, and the lasting friendship we have nurtured have made this journey even more rewarding and fulfilling.

With all my heart, I express my sincerest gratitude to all my beloved family members - my father Yibiao Guan, my mother Aiqing Situ, and my love Miss Li. Their unwavering support has been the inexhaustible soul power in guiding me through the challenges and triumphs of this scholarly endeavor. I will always love them.

Finally, I would like to commend myself for the enduring determination and optimistic attitude throughout this challenging process. This achievement is a testament to the faith I held in my abilities and the perseverance I demonstrated along this arduous yet rewarding journey. I take great pride in the better person I have become.

Contents

Declaration i

Acknowledgements ii

Table of Contents iii

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Motivation and Background . . . 1
- 1.2 Solutions . . . 2
- 1.3 Thesis Overview . . . 2
- 1.4 Related Publications . . . 3

2 Related Work 5

3 GECO: A Confidentiality-Preserving and High-Performance Permissioned Blockchain Framework for General Smart Contracts 7
- 3.1 Introduction . . . 7
- 3.2 Background . . . 11
  - 3.2.1 EOV Permissioned Blockchain . . . 11
  - 3.2.2 Homomorphic Encryption . . . 12
  - 3.2.3 Non-Interactive Zero-Knowledge Proofs . . . 12
  - 3.2.4 Related Work . . . 13
- 3.3 Overview . . . 14
  - 3.3.1 System Model . . . 14
  - 3.3.2 libGeco . . . 15
  - 3.3.3 Example Smart Contract . . . 15
  - 3.3.4 Geco's Protocol Overview . . . 16
- 3.4 Protocol Description . . . 17
  - 3.4.1 Offline Smart Contract Analysis . . . 17
  - 3.4.2 Runtime Transaction Scheduling . . . 20
  - 3.4.3 Defend against Malicious Participants . . . 23
- 3.5 Analysis . . . 23
  - 3.5.1 Correctness . . . 23

3.5.2 Liveness . . . 24
3.5.3 Confidentiality . . . 24
3.6 Evaluation . . . 25
3.6.1 End-to-End Performance . . . 27
3.6.2 Robustness to Malicious Participants . . . 28
3.6.3 Performance under Diverse Applications . . . 28
3.7 Conclusion . . . 29

4 High-Performance Confidentiality-Preserving Blockchain via GPU-Accelerated Fully Homomorphic Encryption 31
4.1 Introduction . . . 31
4.2 Related work . . . 33
4.2.1 Confidentiality-Preserving Blockchains . . . 33
4.2.2 GPU-Accelerated Fully Homomorphic Encryption . . . 33
4.3 Overview . . . 34
4.3.1 System Model . . . 34
4.3.2 Threat Model . . . 34
4.3.3 GAFE's Workflow Overview . . . 34
4.4 Workflow Description . . . 35
4.4.1 Client Key Generation . . . 35
4.4.2 GPU-Accelerated Transaction Execution . . . 36
4.5 Evaluation . . . 38
4.5.1 Settings . . . 38
4.5.2 End-to-End Performance . . . 39
4.6 Conclusion . . . 40

5 GACM: A High-Performance and Confidentiality-Preserving Middleware for Permissioned Blockchains 41
5.1 Introduction . . . 41
5.2 Related work . . . 43
5.2.1 Confidentiality-Preserving Permissioned Blockchains . . . 43
5.2.2 GPU Acceleration for Blockchains and FHE . . . 44
5.3 Overview . . . 44
5.3.1 System Model . . . 44
5.3.2 Threat Model . . . 45
5.3.3 Key Management . . . 46
5.4 Workflow . . . 46
5.4.1 Phase 1: Preparation . . . 46
5.4.2 Phase 2: Execution . . . 46
5.4.3 Phase 3: Commit . . . 49
5.5 Evaluation . . . 49
5.5.1 Settings . . . 49
5.5.2 End-to-End Performance . . . 50

5.6 Conclusion 51

6 Conclusion and Future Work **53**

Bibliography **55**

List of Figures

3.1 For high performance, EOV executors concurrently executes transactions tx_i and tx_j, causing conflicting aborts. 9
3.2 Our Ceov workflow and two existing typical permissioned blockchain workflows. 11
3.3 Geco's runtime workflow. Plaintext and ciphertext are highlighted in red and green, respectively. 14
3.4 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 (§3.4.1). 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. 18
3.5 An example showing the necessity of traversing all combinations of path p_a, p_b, and p_{a+b} in Algorithm 1. 19
3.6 End-to-end performance of Geco and baselines in benign environments with different conflict ratios. 25
3.7 Performance under malicious participants. 27
3.8 Throughputs of contracts in Table 3.7. "$\oplus$ and $\otimes$" is contracts using both HE addition and multiplication. "$\oplus$ only" is contracts using HE addition only. ✗ is contracts unsupported by ZeeStar. ♠ is non-deterministic contracts. 29

4.1 Gafe's GPU-accelerated transaction execution workflow. Each executor utilizes multiple Streaming Multiprocessors (SM) of a GPU to concurrently execute FHE computations for multiple transactions. 35
4.2 End-to-end performance of Gafe and the baseline. 39

5.1 The workflow of a Gacm-enabled blockchain network. 45
5.2 Comparison of effective throughput and commit latency among Gacm, Gacm (w/o. Ftm), Gacm (w/o. Tdga), and Gacm (null). 50

List of Tables

3.1 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. 13

3.2 libGECO APIs. 15

3.3 Data types and functions used in Listing 3.2. Note that the FHE functions like `HAdd` are implemented by smart contract developers and can adopt various FHE schemes. 16

3.4 Variables for Algorithm 1. 20

3.5 Variables for Algorithm 2. 23

3.6 Each phase's latency and the 99% tail latency of Figure 3.6(b). 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. 26

3.7 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. 26

4.1 Comparison of GAFE and related confidentiality-preserving blockchains. "❖/◆" represent general-purpose and specific-purpose blockchains, respectively. 33

5.1 Comparison of GACM and related confidentiality-preserving permissioned blockchains. "❖/◆" represent the architecture approach and the cryptography approach, respectively. 43

5.2 The end-to-end performance of GACM and the three baseline systems under HLF and the SmallBank workload. "tput" denotes the effective throughput and is measured in transactions per second (TPS). "lat" refers to the transaction commit latency and is measured in milliseconds. "ar" stands for the transaction abort rate. 49

Chapter 1

Introduction

1.1 Motivation and Background

Since the emergence of Bitcoin [58], blockchains have made profound impact across industries and academia. At its core, blockchain is a distributed ledger that is collectively maintained among multiple nodes and employs decentralized consensus protocol [88] such as Proof-of-Work [34, 58] and Proof-of-Stake [71, 48] to agree on the ledger data. The traceability, immutability, and decentralization nature makes blockchain an appealing solution for various industries such as cryptocurrency [57] and supply chain [27]. However, the public data accessibility of notable blockchains like Bitcoin [58] and Ethereum [86] enable anyone to be able to peek on any data stored on-chain, resulting in a deep concern about data confidentiality. The situation is even exacerbated when blockchains' end-to-end performance is dragged by expensive decentralized consensus protocol, weak node computation power, and unstable inter-node connection. Compromised data confidentiality and low end-to-end performance have become the two prominent and unsolved challenges, seriously hindering the further adoption of blockchains.

Permissioned blockchains have emerged as an promising alternative in response to these challenges. Unlike permissionless blockchains, permissioned blockchains restrict network participation through identity-based authentication and role-based access control [5, 21, 40], thereby enhancing data confidentiality. By replacing computationally intensive consensus protocols with Byzantine Fault Tolerance (BFT) variants like PBFT [17] or Raft [61], permissioned blockchains achieve higher transaction throughput [35] while maintaining decentralized trust. A notable example is Hyperledger Fabric [5] (HLF), an enterprise-grade permissioned blockchain framework that implements channel-based data isolation and modular consensus architecture. HLF restricts access to transaction data exclusively to blockchain participants while supporting parallel execution of smart contracts. These advantages make permissioned blockchains suitable for regulated industries like finance [92] and healthcare [52].

However, the reliance on identity-based authentication and channel-based isolation mechanisms alone fails to provide comprehensive data confidentiality guarantees. This is because malicious participants within authorized organizations—such as corrupted executor nodes or compromised ordering nodes—can exploit systemic vulnerabilities

to access sensitive transactional data. Consequently, current permissioned blockchain implementations still exhibit critical vulnerabilities in data confidentiality, urgently requiring solutions to address this prominent systemic deficiency.

1.2 Solutions

This thesis presents three high-performance and confidentiality-preserving systems designed for permissioned blockchains. Each system proposes dedicated protocols to accelerate the execution of transactions involving ciphertext data, thereby reconciling the traditionally conflicting objectives of high performance and strong data confidentiality.

In our first work, we present Geco (§3), 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 HLF, Geco generates contract logic-agnostic NIZKPs, which prove only that quorum organizations produce consistent results, thereby ensuring the correctness of contract execution while addressing potential non-determinism. For better performance, Geco merges multiple conflicting transactions into a single one, minimizing conflicting aborts and invocations of cryptographic primitives.

In our second work, we introduce Gafe, a high-performance confidentiality-preserving blockchain that carries a GPU-accelerated transaction execution workflow. Gafe encrypts transaction data with FHE, allowing both addition and multiplication on ciphertexts. For high performance, Gafe leverages parallel execution on GPUs to accelerate FHE computations. For result correctness, Gafe generates lightweight NIZKPs that incur low overhead.

In our third work, we propose Gacm, a high-performance and confidentiality-preserving middleware for permissioned blockchains. Gacm effectively integrates FHE with permissioned blockchains' explicit membership identification mechanism to ensure end-to-end data confidentiality, eliminating the need for resource-intensive cryptographic proofs (e.g., NIZKPs) for result validation. Moreover, Gacm improves the performance of FHE transaction execution through two novel protocols: (1) the Ftm protocol minimizes the number of FHE transactions that need to be executed via merging FHE transactions, and (2) the Tdga protocol enables multi-dimensional GPU acceleration (transaction-, statement-, and FHE-operator-level parallelism) for FHE transactions.

1.3 Thesis Overview

The remainder of the thesis is organized as follows: Chapter 2 discusses the related work. Chapter 3 introduces GECO, a confidential-preserving and high-performance permissioned blockchain framework for general smart contracts involving ciphertext data. Chapter 4 introduces GAFE, a permissioned blockchain that employs GPU acceleration for transaction execution efficiency and applies FHE for on-chain data confidentiality.

Chapter 5 introduces GACM, a high-performance and confidentiality-preserving middleware for permissioned blockchains that performs FHE transaction merging and leverages GPU to accelerate FHE transaction execution from three distinct dimensions. Finally, Chapter 6 concludes and foresees future work.

1.4 Related Publications

- **Chapter 3 :** Rongxin Guan, Songxiao Guo, Ji Qi, Sen Wang, Nicholas Zhang, Yanjun Wu, and Heming Cui*, "GECO: A Confidentiality-Preserving and High-Performance Permissioned Blockchain Framework for General Smart Contracts", [Under Review].
- **Chapter 4 :** Rongxin Guan, Tianxiang Shen, Sen Wang, Gong Zhang, and Ji Qi*, "High-Performance Confidentiality-Preserving Blockchain via GPU-Accelerated Fully Homomorphic Encryption", Advanced Information Systems Engineering Workshops, International Conference on Advanced Information Systems Engineering (CAiSE), 2024.
- **Chapter 5 :** Rongxin Guan, Heming Cui, and Ji Qi*, "GACM: A High-Performance and Confidentiality-Preserving Middleware for Permissioned Blockchains", [Under Revision].

Chapter 2

Related Work

Related work addresses the data confidentiality issue through two distinct approaches: the architecture approach and the cryptography approach.

The architecture approach enhance data confidentiality via designing new blockchain architectures with trusted hardware or centralized executors. Ekiden [18] leverages trusted execution environments (TEEs) to isolate smart contract computations. Arbitrum [45] processes smart contracts in plaintext through trusted managers, risking client data exposure. While Caper [3] and Qanaat [4] implement authenticated data access models, they fail to prevent inter-organizational data leakage due to shared computation contexts. Although the solutions of the architecture approach achieve near-native performance by avoiding cryptographic overhead, they introduce critical trust assumptions: TEEs rely on hardware vendors' security guarantees, while manager-based models risk collusion among malicious software managers.

The cryptography approach employs cryptographic primitives like homomorphic encryption (HE) [1] and non-interactive zero-knowledge proofs (NIZKPs) [87] to enforce data confidentiality. By executing contracts directly on ciphertexts and generating proofs of correct computation (e.g., zk-SNARKs [11, 37]), the cryptography approach theoretically prevents data leakage to all participants and achieves end-to-end data confidentiality. However, existing implementations of the the cryptography approach suffers from extremely poor performance due to the expensive computation overhead caused by the cryptographic primitives, rendering them impractical for real-world deployment. ZeeStar [76] employs additive partial homomorphic encryption (PHE) [53] with multiplicative emulation via repeated additions, but suffers from inefficiency and multi-organizational ciphertext incompatibility. Zapper [75] combines oblivious Merkle trees with NIZK proofs, yet both it and ZeeStar cannot handle non-deterministic contracts due to their reliance on the order-execute (OE) workflow (§3.2.1). Hawk [49] enforces transaction privacy through symmetric encryption and NIZKPs, but restricts applications to token transfers rather than general computations. SmartFHE [74] implements FHE within the OE workflow, but its inability to process multi-key ciphertexts and non-deterministic transactions renders it impractical for multi-organizational deployments. FabZK [46] obscures client identities via tabular ledgers at the cost of application generality, fundamentally limiting its smart contract support. In summary, neither approach achieves the dual objectives of strong data confidentiality and high performance.

Chapter 3

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

3.1 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 [86] process and store client data in plaintext, raising deep concerns over data confidentiality. Such concerns are particularly problematic for applications involving highly sensitive information such as financial data [6], as they are under stringent privacy laws, such as the General Data Protection Regulation of the European Union [83]. For example, a typical token transfer contract (see Listing 3.1) executes in plaintexts, causing a data breach.

```
1 func Transfer(client A, client B, num V) {
2   num AV = GetState(A);
3   num BV = GetState(B);
4   if (AV < V) { Abort(); }
5   RunThread({PutState(A, AV - V)}); // thread 1
6   RunThread({PutState(B, BV + V)}); // thread 2
7 }
```

Listing 3.1: 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 [18] uses trusted execution environments (TEE), while Arbitrum [45], Caper [3], and Qanaat [4] 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 [76, 75, 77], 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 [67]. 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 [13, 30] (a popular FHE scheme) inside the elliptic curve-based NIZKP [23] takes tens of seconds to generate a NIZKP [74]. 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 [59] NIZKP using 2048-bit keys for the Paillier PHE scheme [63], consumes more than 256 GB of memory [76], 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) [5]. This mechanism first executes a transaction on multiple executor nodes (Figure 3.1). 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.

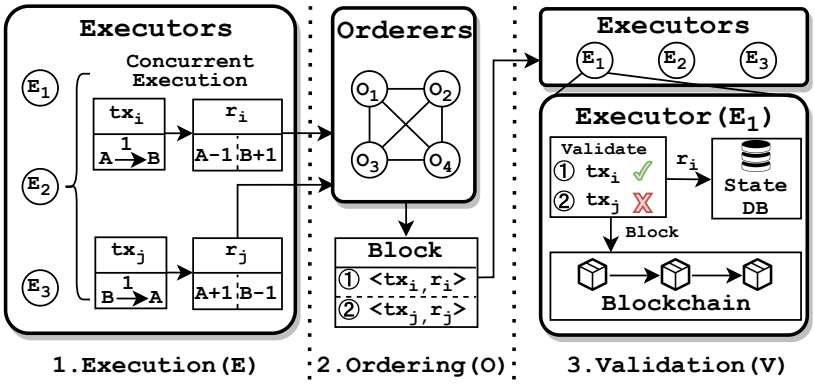


Figure 3.1: For high performance, EOV executors concurrently executes transactions tx_i and tx_j, causing conflicting aborts.

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 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 [5]). In Figure 3.1, 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 [54], a highly optimized FHE library, takes 2822 milliseconds to execute a single FHE multiplication [56].

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

To tackle the above 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 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 [5], a prominent EOV permissioned blockchain framework. We integrated GECO with Lattigo [56], a performant FHE library, and Gnark [20], a popular NIZKP library. We evaluated GECO using both the Small-Bank workload [38] 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 [76]. Our evaluation shows that:

- GECO is efficient (§3.6.1). 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 (§3.6.3). GECO tolerates non-deterministic contracts and supports cryptographic primitives like FHE which allows arbitrary computations over ciphertexts.
- GECO is robust (§3.6.2). 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 [27] and healthcare [52]. GECO can also attract non-deterministic applications developed in general programming languages like Go to be deployed on.

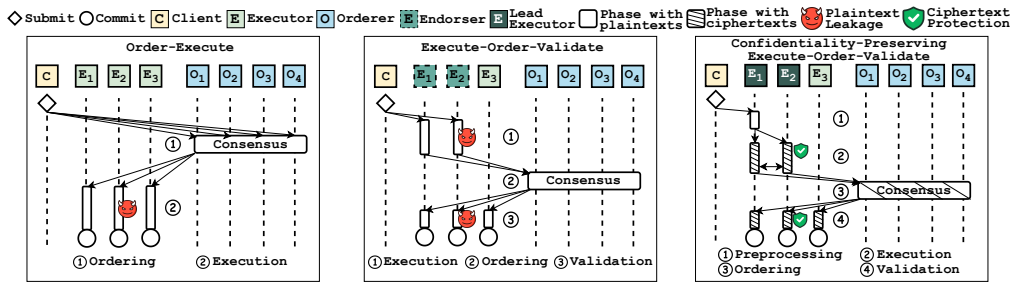


Figure 3.2: Our GECO workflow and two existing typical permissioned blockchain workflows.

3.2 Background

3.2.1 EOV Permissioned Blockchain

A blockchain can be either *permissionless* or *permissioned*. Permissionless blockchains (e.g., Ethereum [86]) are open to anyone, and their participants are mutually untrusted. In contrast, permissioned blockchains (e.g., HLF [5]) 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 3.2 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 [58, 86]. 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 [67].

The EOV workflow (Figure 3.2) 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 [17]) 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 [67]), 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.

3.2.2 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 §3.1. 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 [33], 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.

3.2.3 Non-Interactive Zero-Knowledge Proofs

GECO co-designs the non-interactive zero-knowledge proof (NIZKP) [31, 36] 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 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) [11, 37], which allows any arithmetic circuit ϕ and guarantees constant-cost proof verification in the size of ϕ, making it suitable for blockchain applications [77, 9, 26]. GECO uses the Gnark [20] implementation of the Groth16 [59] system (a zkSNARK construction) with advantages of constant proof size (a few kilobytes) and short verification time (tens of milliseconds).

System	Data Encryption	Multiple Organizations	General Contract	High Performance
✧ Ekiden [18]	√	√	×	√
✧ Arbitrum [45]	×	√	×	√
✧ Qanaat [4]	×	√	√	√
✧ Caper [3]	×	√	√	√
♦ ZeeStar [76]	√	✤	×	×
♦ Zapper [75]	√	×	×	×
♦ Hawk [49]	√	√	×	√
♦ SmartFHE [74]	√	√	×	×
♦ FabZK [46]	√	×	×	×
♦ GECO	√	√	√	√

Table 3.1: 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.

3.2.4 Related Work

We now discuss previous work on confidentiality-preserving blockchains, as illustrated in Table 3.1.

Architecture approach. Existing work attempts to ensure data confidentiality by adopting dedicated blockchain architectures. Ekiden [18] demands TEE hardware to process private data. Arbitrum [45] executes smart contracts in plaintext and imposes strong trust assumptions on smart contract executors that might disclose clients' sensitive data. Caper [3] and Qanaat [4] 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 [76] uses an additive PHE scheme [53] 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 [75] 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 §3.2.1.

Hawk [49] 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 [74] 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 [46] 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.

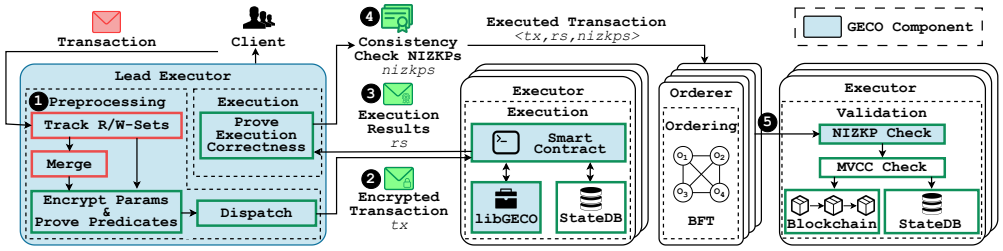


Figure 3.3: GECO's runtime workflow. Plaintext and ciphertext are highlighted in red and green, respectively.

3.3 Overview

3.3.1 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 [17]).

Executors. Each executor is responsible for three tasks: executing transactions, validating results, and maintaining a local blockchain state database. As Figure 3.3 shows, each organization has a designated *lead executor*, which is built upon the existing Fabric Gateway [28] 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.

Threat model. GECO adopts the Byzantine failure model [17, 10], 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 [5, 67, 65] 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

No.	API	Description
		Lead executor APIs
1	`Preprocess(tx)`	Preprocess the given transaction. (§3.4.2)
2	`Prove(rs)`	Generate NIZKPs (§3.4.2) to prove that quorum results are consistent.
3	`Verify(rs,nizkps)`	Verify NIZKPs (§3.4.2) to ensure that quorum results are consistent.
4	`Analyze(C)`	Run the offline analysis (§3.4.1) on the contract.
		Smart contract APIs
5	`Require(predicate)`	Abort transaction if the predicate is false. (§3.4.2)

Table 3.2: libGeco APIs.

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 [76, 49, 74]. (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 (§3.3.4) to tolerate the random noise [19, 13, 30] in FHE ciphertexts, while using Geco's Ceov workflow to handle the non-determinism caused by language features like multithreading [67]. (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 (§3.1).

3.3.2 libGeco

Geco offers a library named libGeco with five APIs (see Table 3.2). The lead executor APIs enable lead executors to preprocess transactions, ensure execution correctness, and analyze contracts' mergeability (Definition 3.4.3). The smart contract API (i.e., `Require()`) ensures the validity of predicates with ciphertexts and follows the idea of the *required* statement of ZeeStar [76]. For example, the `Require()` API in Listing 3.2 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 §3.3.4 and §3.4.2.

3.3.3 Example Smart Contract

```
1 func Transfer(client A, client B, cnum V) {
2   cnum AV = GetState(A);
3   cnum BV = GetState(B);
4   Require(AV >= V);
5   RunThread({PutState(A, HSub(AV, V))}); // thread 1
6   RunThread({PutState(B, HAdd(BV, V))}); // thread 2
7 }
```

Listing 3.2: 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 3.2 and Table 3.3 illustrate a Geco smart contract for token transfer. In contrast to Listing 3.1, Listing 3.2 achieves data confidentiality by executing contracts over ciphertexts. Listing 3.2 also tolerates non-determinism caused by multithreading. For instance, multiple threads concurrently updating a value might produce different

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(`$\texttt{ct}_1$`,`$\texttt{ct}_2$`)`	Perform FHE addition $\texttt{ct}_1 + \texttt{ct}_2$.
`HSub(`$\texttt{ct}_1$`,`$\texttt{ct}_2$`)`	Perform FHE subtraction $\texttt{ct}_1 - \texttt{ct}_2$.
`RunThread(func)`	Run the given function in a separate thread.

Table 3.3: Data types and functions used in Listing 3.2. Note that the FHE functions like `HAdd` are implemented by smart contract developers and can adopt various FHE schemes.

results. Note that functions `HAdd` and `HSub` can be implemented with any FHE scheme, such as the BFV scheme [13, 30]. 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 3.2) 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 (§3.4.2) 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.

3.3.4 GECO's Protocol Overview

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

Offline contract analysis (§3.4.1). 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 3.4.3). For example, the contract in Listing 3.2 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 (§3.4.2). GECO incorporates a new CEOV workflow and the CTM protocol to schedule transaction executions at runtime, as shown in Figure 3.3.

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 3.3): it caches transactions submitted by clients from the same organization (§3.3.1), 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 [5], 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 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.

3.4 Protocol Description

3.4.1 Offline Smart Contract Analysis

Definition 3.4.1 (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 3.4.2 (Equivalent transactions)**.** Two transactions are *equivalent* iff they invoke the same contract and produce identical resulting states irrespective of the initial state.

Definition 3.4.3 (Mergeable smart contract)**.** A smart contract is *mergeable* iff, for

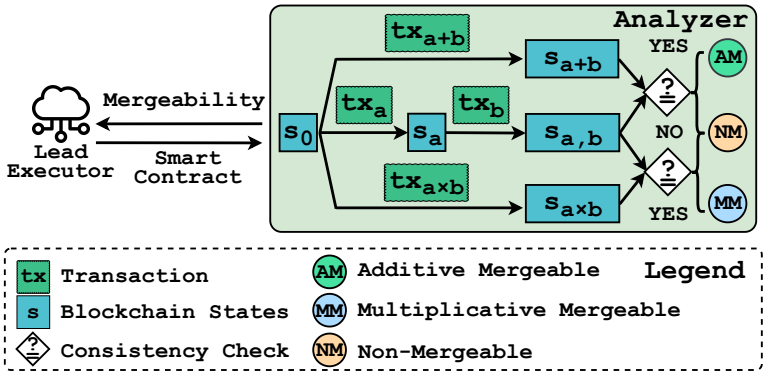


Figure 3.4: GECO's offline smart contract analysis protocol. tx_{a+b} and $tx_{a\times b}$ are formed by merging tx_a and tx_b via adding or multiplying their input parameters (§3.4.1). s_{a+b} and $s_{a\times b}$ are the states produced by executing tx_{a+b} and $tx_{a\times b}$. $s_{a,b}$ is the state produced by sequentially executing tx_a and tx_b.

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 3.4. 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 3.3), the analyzer labels it as a key parameter; otherwise, it is labeled as a value parameter. Then, the analyzer uses symbolic execution [8, 47, 22] tool KLEE [16] with the theorem prover Z3 [81] to get all execution paths of C. As Algorithm 1 and Table 3.4 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* [78]. 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

```
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 3.5: 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` (§3.4.1;API 4)

```
1  paths ← GetAllExecutionPaths(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 ← GenRandStateDB(C, p);
7              k, v_a, v_b ← GenRandParams(C, p);
8              tx_a ← GenTX(k, v_a); tx_b ← GenTX(k, v_b); tx_{a+b} ← GenTX(k, v_a + v_b);
               // Step 2: Execution
9              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
10             if s_{a,b} ≠ s_{a+b} then
11                 return NonMergeable
12         end
13     end
14 end
15 return AdditiveMergeable
```

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 $s_{a,b} = s_{a+b}$ 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 3.5, 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.

Variable	Description
$\mathtt{s}$, $\mathtt{s}_a$, $\mathtt{s}_{a,b}$, $\mathtt{s}_{a+b}$	States in blockchain database.
$\mathtt{tx}_a$, $\mathtt{tx}_b$, $\mathtt{tx}_{a+b}$	Transactions.
$\mathtt{paths}$, $\mathtt{p}$, $\mathtt{p}_a$, $\mathtt{p}_b$, $\mathtt{p}_{a+b}$	Execution paths.
$\mathtt{k}$	Key parameters of the smart contract.
$\mathtt{v}_a$, $\mathtt{v}_b$	Value parameters of the smart contract.

Table 3.4: Variables for 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.

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 [72] often execute many transactions between identical pairs of accounts. Besides, contracts that log IoT sensor data [64] 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.

3.4.2 Runtime Transaction Scheduling

GECO's *runtime protocol* (Algorithm 2 and Table 3.5) 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 each transaction's read-write set. For example, for a transaction ($A \xrightarrow{V} B$) that invokes the contract in Listing 3.2, 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 set. 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 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 3.3). 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, 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 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 3.2 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, the read-write sets contain ciphertexts that cannot be directly compared as they involve random noise for security reasons [82]. 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

Algorithm 2: Runtime protocol of the lead executor (§3.4.2)

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

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 and org_3 provides NIZKPs for key B and C. These NIZKPs are all 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 affirm the consistency of tx's execution results.

Phase 3: Ordering. GECO orderers run a BFT consensus protocol (e.g., PBFT [17]) 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.

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 3.5: Variables for Algorithm 2.

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.

3.4.3 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 (§3.3.1). 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 (§3.1).

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.

3.5 Analysis

3.5.1 Correctness

Theorem 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 the state obtained by sequentially

executing all valid transactions on the same initial state (ACID).

Proof. GECO inherits the BFT safety of existing EOV blockchains [5] as GECO preserves the consensus protocol without modifying it. Specifically, GECO attains *block consistency*, where all benign executors process and append identical blocks in the same order determined by the orderers, and *state consistency*, where all benign executors maintain a consistent state and a consistent local copy of the blockchain.

GECO ensures the ACID properties of transaction execution. (1) *Atomicity*: GECO either commits or aborts each transaction completely. GECO regards the transactions merged into a single one by the CTM protocol as a whole, meaning they are either all committed or all aborted. (2) *Consistency*: GECO solely commits valid transactions with consistent results in sequential order as dictated by the orderers. This leads to a consistent state among all benign executors, provided an identical initial state. (3) *Isolation*: Concurrent transactions have no access to the intermediate state of one another during the execution phase, and conflicting transactions are aborted in the validation phase. (4) *Durability*: Once a transaction is committed, its modifications to the blockchain state are irreversible. □

3.5.2 Liveness

Theorem 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 [5]. Specifically, the orderers run a BFT consensus protocol, which guarantees to finalize tx into a block B (§3.4.2) and deliver the block to all executors. □

3.5.3 Confidentiality

Theorem 3 (Confidentiality)**.** For any ciphertext ct along with its associated plaintext 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 ensures the confidentiality guarantee throughout all phases of the CEOV workflow. As Figure 3.2 shows, GECO only processes client data in plaintext in the preprocessing phase (Phase 1 in §3.4.2), while handling encrypted client data in other phases. During Phase 1, only the lead executor of the clients has access to the plaintext inputs of transactions. The lead executor will not expose the plaintext inputs as the lead executor and clients are mutually trusted (§3.3.1), thereby ensuring data confidentiality in Phase 1. In other phases, client data are protected by the encryption schemes used by GECO (e.g., the BFV scheme [13, 30]). When generating consistency check NIZKPs (§3.4.2), each lead executor can only decrypt the ciphertexts belonging to its organization and cannot decrypt those of other organizations. Moreover, these encryption schemes achieve *indistinguishability under chosen-plaintext attack* (IND-CPA [12]), implying that any attacker from a different organization $org*$ cannot decrypt the ciphertext belonging

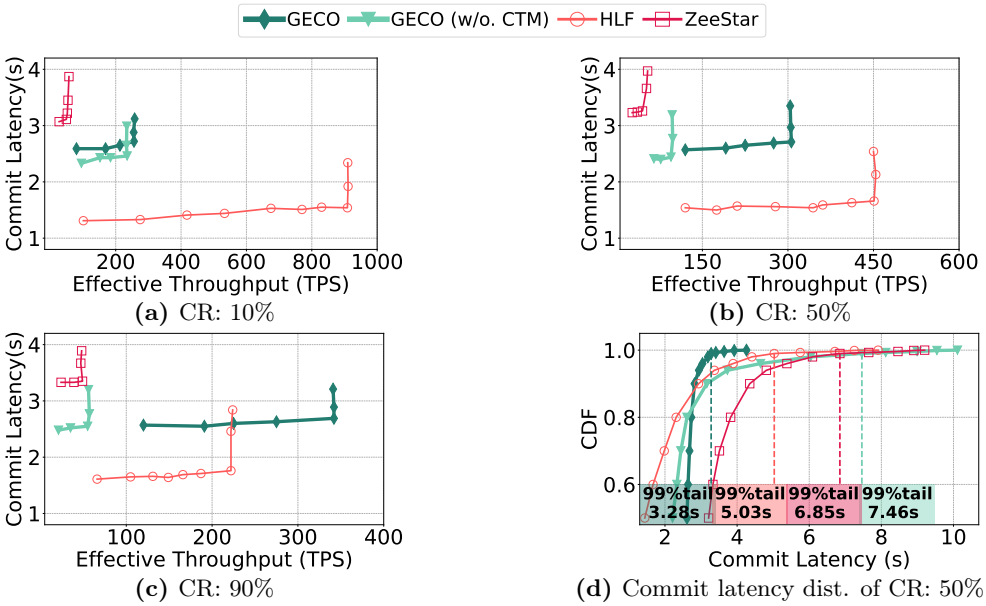


Figure 3.6: End-to-end performance of GECO and baselines in benign environments with different conflict ratios.

to *org*'s clients, since the *org* keeps its decryption key confidential. Thus, GECO achieves data confidentiality in all phases of the CEOV workflow. □

3.6 Evaluation

GECO implementation. We built GECO on HLF v2.5 [29]. GECO uses the BFV scheme [13, 30] implemented by Lattigo [56] and uses the Groth16 NIZKP [36] on the BN254 elliptic curve implemented by Gnark [20]. Same as existing systems [76, 9] 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 [5], ZeeStar [76], GECO (w/o. CTM), and GECO (w/o. defense). HLF is one of the most popular permissioned blockchain frameworks [67, 35, 69]. ZeeStar is a notable confidentiality-preserving blockchain that encrypts sensitive data using an additive PHE scheme [53] and generates Groth16 NIZKPs to ensure the execution correctness. GECO (w/o. CTM) implements only the CEOV workflow without integrating our CTM protocol (§3.4). 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 §3.4.3.

Workloads. We used two workloads for evaluation. The first one (§3.6.1 and §3.6.2) is *SmallBank* [38], a widely used benchmark for evaluating blockchain performance [67, 73]. SmallBank simulates the banking scenario and provides diverse contracts like token transfers. We evaluated the encrypted version of SmallBank, where each contract is re-written with libGECO APIs (see Table 3.2) and Gnark. Our second workload (§3.6.3) consists of ten smart contracts from existing work [76, 75, 74] that involves HE arithmetic

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 3.6: Each phase's latency and the 99% tail latency of Figure 3.6(b). 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.

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 3.2).

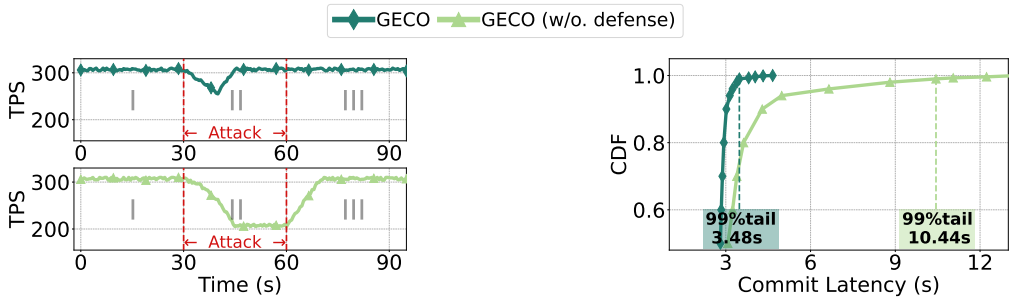
Table 3.7: Example smart contracts used in the second workload. ⊕ and ⊗ indicate whether the contract uses HE addition or multiplication, respectively. ♠ indicates that the contract is non-deterministic.

operations and non-determinism, as shown in Table 3.7.

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 [5], 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 [17] protocol. Same with existing work [73], in §3.6.1 and §3.6.2, 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.



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

Figure 3.7: Performance under malicious participants.

Our evaluation focused on three primary questions.

§3.6.1 How efficient is GECO compared to baselines?

§3.6.2 How robust is GECO to malicious participants?

§3.6.3 How efficient is GECO under diverse applications?

3.6.1 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 3.2). 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 3.6 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 (§3.4.2) 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 (§3.4.2), which were agnostic to the contract logic. In contrast, ZeeStar relied on the inefficient re-execution method (§3.1) for NIZKP generation. This enables GECO to achieve shorter commit latency than ZeeStar, as confirmed in Table 3.6.

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 3.6 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 (§3.4), 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.

3.6.2 Robustness to Malicious Participants

We conducted NIZKP omission attack (§3.4.3) on GECO and GECO (w/o. defense) with the same settings as §3.6.1. 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 (§3.4.3) is effective. Figure 3.7(a) 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 3.7(b) shows that GECO's tail latency barely changed during the attack (compared to Figure 3.6(d)). 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 [67] and bidding [76].

3.6.3 Performance under Diverse Applications

We evaluated GECO, GECO (w/o. CTM), and ZeeStar using the second workload (Table 3.3). As Figure 3.8 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 [25, 56]. 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

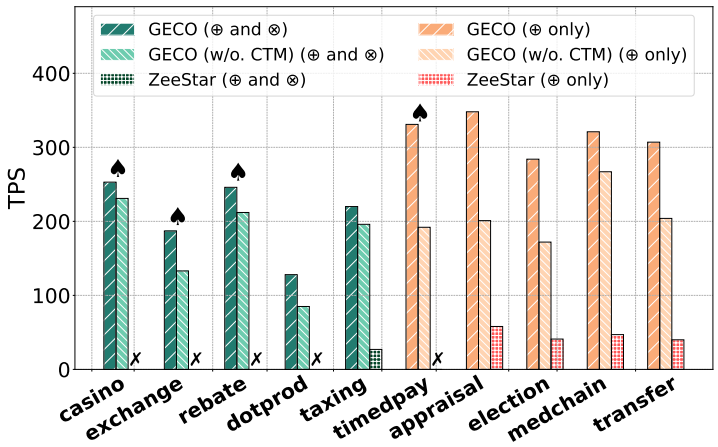


Figure 3.8: Throughputs of contracts in Table 3.7. "⊕ and ⊗" is contracts using both HE addition and multiplication. "⊕ only" is contracts using HE addition only. ✗ is contracts unsupported by ZeeStar. ♠ is non-deterministic contracts.

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 (§3.4.2), we can easily employ various cryptographic primitives like the CKKS scheme [19] without modifying our protocol.

3.7 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.

Chapter 4

High-Performance Confidentiality-Preserving Blockchain via GPU-Accelerated Fully Homomorphic Encryption

4.1 Introduction

Blockchain has been a transformative force for both industry and academia [7] due to its exceptional characteristics, such as immutability [50]. However, notable blockchains like Hyperledger Fabric [5] (HLF) face a serious confidentiality issue as they process and store transaction data in plaintext, exposing sensitive information to anyone with access to the blockchain. This confidentiality issue impedes the widespread adoption of blockchain, especially in safety-critical applications such as finance [92], where data confidentiality is crucial.

A promising approach to addressing the confidentiality issue is using homomorphic encryption [2] (HE) and non-interactive zero-knowledge proofs [79] (NIZKPs). HE enables arithmetic computation to be performed directly on ciphertexts. NIZKPs allow a prover to prove that a statement is true without revealing any information beyond the validity of the statement itself. In this approach, clients encrypt transaction input data using an HE scheme, and nodes execute transactions directly on ciphertexts. Then, clients generate NIZKPs to prove the execution correctness, without disclosing the plaintexts of the results.

However, prior work on this approach faces two prominent deficiencies. Firstly, they suffer from low performance due to the intensive computational costs of HE. For instance, the notable confidentiality-preserving blockchain ZeeStar [76] exhibits a long commit latency of tens of seconds. As a result, prior work fails to meet the high-performance requirements of many blockchain applications, such as digital payment [92]. Secondly, prior work has serious limitations in expressing complex business logic due to its reliance on partially homomorphic encryption [2] (PHE). PHE supports either addition or multiplication on ciphertexts, making prior work unsuitable for applications like finance [92] that involve both types of arithmetic operations. Although fully homomorphic encryption (FHE) offers a promising alternative that allows arbitrary computation on ciphertexts, its adoption is impeded by its even higher computation costs.

Our key insight to address the deficiencies is that, *we can efficiently integrate FHE and blockchains by introducing GPU acceleration for transaction execution and FHE computation.* Blockchain transactions that invoke the same smart contract are highly parallelizable, thus enabling parallel computations of FHE across multiple transactions. These parallel computations are well-suited for GPUs, which offer superior parallel processing capabilities and are widely available in modern commodity machines. Hence, employing GPU-accelerated FHE is both beneficial and feasible for blockchains, as it enables high performance, confidentiality preservation, and the implementation of complex business logic.

Nonetheless, the trivial combination of FHE and blockchains leads to the problem of inconsistent ciphertext results. Specifically, when given identical inputs, different nodes may generate inconsistent ciphertexts for the same plaintext result. This inconsistency occurs because FHE schemes intentionally introduce random noise to ciphertexts. This noise serves as a protection against attacks aimed at extracting information from the ciphertext [82].

To solve the problem of inconsistency, our second insight is *to integrate lightweight NIZKPs with the trusted execution mechanism of the execute-order-validate blockchain workflow* [5]. This mechanism first executes a transaction on multiple nodes and considers it correct iff a majority of nodes produce consistent results. Inspired by this mechanism, clients can generate NIZKPs that decrypt all ciphertext results and check the consistency of the majority of plaintexts. The generation of NIZKPs is lightweight as it does not perform costly HE arithmetic computations like existing studies [76, 89].

These two insights lead to GAFE[1], a high-performance confidentiality-preserving blockchain. GAFE carries a *GPU-accelerated transaction execution workflow* in four phases. First, the client encrypts transaction input data using FHE and sends the encrypted transaction to all executor nodes. Second, each executor uses a GPU to execute FHE computation for multiple transactions in parallel, then the client generates NIZKPs to prove the execution correctness. Third, the orderer nodes run a consensus protocol to determine the transaction order within each block. Finally, the executors validate and commit the transactions in the determined order. In short, this workflow achieves high performance while ensuring the provably correct execution of confidentiality-preserving transactions.

We built GAFE on top of HLF [5], a notable execute-order-validate blockchain framework. We compared GAFE with a baseline that performs FHE computation solely on the CPU. The results show that GAFE achieves high performance, with an effective throughput of 258 transactions per second and a commit latency of 1.61 seconds. Compared to the baseline, GAFE achieved a 3.1× increase in throughput and a 37% reduction in latency.

In summary, we make the following contributions: (1) We propose a GPU-accelerated transaction execution workflow that integrates GPU-accelerated FHE into blockchain and ensures execution correctness through lightweight NIZKPs. (2) We implement

[1] GAFE stands for GPU-Accelerated Fully Homomorphic Encryption Blockchain.

System	FHE Support	GPU Acceleration	High Performance
❖ ZeeStar [76]	×	×	×
❖ Ekiden [18]	×	×	✓
❖ Hawk [49]	×	×	✓
❖ Arbitrum [45]	×	×	✓
◆ Zcash [41]	×	×	×
◆ Monero [55]	×	×	×
◆ FabZK [46]	×	×	✓
◆ Zether [15]	×	×	×
◆ RFPB [89]	×	×	✓
◆ GAFE	✓	✓	✓

Table 4.1: Comparison of GAFE and related confidentiality-preserving blockchains. "❖/◆" represent general-purpose and specific-purpose blockchains, respectively.

GAFE, a high-performance confidentiality-preserving blockchain that incorporates the aforementioned workflow. (3) We conduct evaluations on GAFE, demonstrating its effectiveness and high performance.

4.2 Related work

4.2.1 Confidentiality-Preserving Blockchains

We categorize prior work on confidentiality-preserving blockchains into two groups, as shown in Table 4.1.

General-purpose blockchains. ZeeStar [76] uses ElGamal encryption [53], which only supports HE addition. While ZeeStar extends addition to emulate multiplication, this extension is inefficient and not applicable to ciphertexts encrypted by different keys. Ekiden [18], Hawk [49], and Arbitrum [45] present substantial vulnerabilities due to their reliance on trusted managers or hardware. Note that GAFE is a general-purpose blockchain.

Specific-purpose blockchains. To conceal sensitive digital payment information, Zcash [41] employs NIZKPs, while Monero [55] uses ring signatures and stealth addresses. However, their poor performance has impeded their broader adoption [68, 15]. FabZK [46] combines NIZKPs with a specialized tabular data structure to realize confidentiality, but this data structure restricts FabZK to performing only HE addition. Zether [15] and RFPB [89][2] also support only HE addition due to the use of PHE schemes.

4.2.2 GPU-Accelerated Fully Homomorphic Encryption

Much prior work leverages GPU acceleration for FHE computations to unlock the potential of FHE in real-world applications. HE-Booster [85] accelerates polynomial arithmetic computation by mapping five common phases of typical FHE schemes to the GPU parallel architecture. It also introduces thread-level and data-level parallelism to enable acceleration on single-GPU and multi-GPU setups, respectively. Ozcan et al. [62] presents a library that makes efficient use of the GPU memory hierarchy and minimizes

[2]We refer to the system proposed in [89] as RFPB for convenience.

the number of GPU kernel function calls. Yang et al. [90] provides GPU implementations of three notable FHE schemes (BGV [14], BFV [13, 30], and CKKS [19]) along with various optimizations, including a hybrid key-switching technique and several kernel fusing strategies. Note that GAFE is orthogonal to the implementation of GPU-accelerated FHE schemes, allowing GAFE to leverage the latest advancements in this field.

4.3 Overview

4.3.1 System Model

GAFE comprises three types of participants: *client*, *executor*, and *orderer*. Executors and orderers are referred to as *nodes*. GAFE uses permissioned settings, where all participants are organized into distinct *organizations* and explicitly identified. Each organization runs multiple executors and orderers, as well as possesses a set of clients. We built a prototype system of GAFE on top of HLF. Specifically, each type of participant is described as follows:

Client. Clients encrypt transaction input data, submit the encrypted transactions to the nodes for execution and commitment, and prove the execution correctness. Each client is identified by a unique string *id* and owns a public-private key set for performing FHE computations. In addition, each client is associated with FHE ciphertexts, such as the client's balance *bal*.

Executor. Each executor is responsible for three main tasks: (1) executing encrypted transactions, (2) validating transaction results, and (3) maintaining the latest local copy of blockchain and state database. Each executor is equipped with a GPU to accelerate FHE computations during transaction execution. We provide a detailed discussion of GAFE's transaction execution workflow in §4.4.2.

Orderer. Orderers determine the order of transactions within each block via a Byzantine-Fault-Tolerant consensus protocol (e.g., PBFT [17]).

4.3.2 Threat Model

GAFE adopts the Byzantine failure model [10, 17], which tolerates up to N malicious orderers out of $3N + 1$ orderers. Each client is limited to accessing only the plaintext of its own data and is unable to access the plaintext of other clients' data. Nodes are incapable of revealing the plaintext of any client's data. We make standard assumptions on FHE and NIZKP.

4.3.3 GAFE's Workflow Overview

GAFE carries a *GPU-accelerated transaction execution workflow* (§4.4.2) that consists of four phases, as shown in Figure 4.1. We take digital payment as an example to illustrate our workflow.

Phase 1: Construction. The client constructs a transaction tx by encrypting the transaction input data (e.g., payment amount val) and generates an NIZKP $\pi_{bal \geq val}$ to demonstrate that the client's balance bal is greater than or equal to val. Finally, the client submits tx and $\pi_{bal \geq val}$ to all executors for execution.

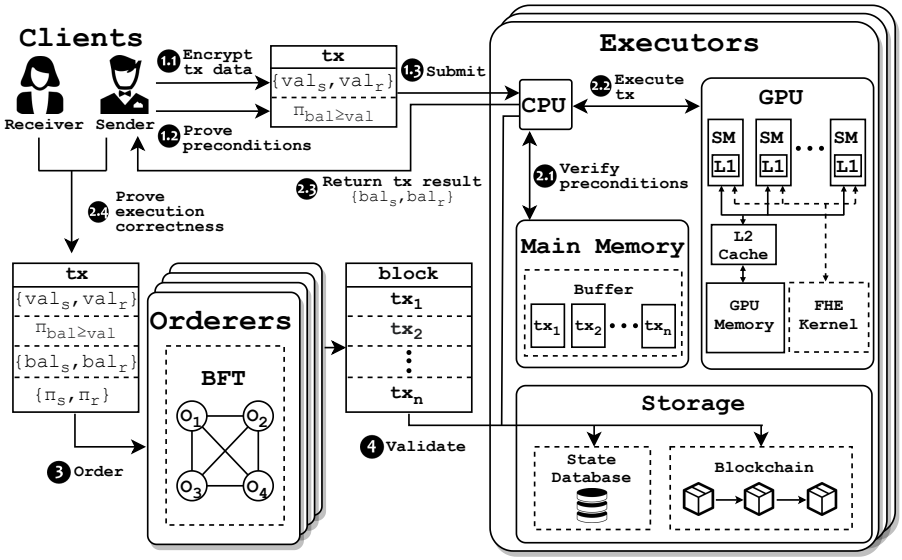


Figure 4.1: Gafe's GPU-accelerated transaction execution workflow. Each executor utilizes multiple Streaming Multiprocessors (SM) of a GPU to concurrently execute FHE computations for multiple transactions.

Phase 2: Execution. The executor verifies $\pi_{bal \geq val}$ and continues execution iff $\pi_{bal \geq val}$ is valid. Next, the executor buffers tx along with other transactions received within a specific time frame. These buffered transactions are moved to a GPU for concurrent FHE computations, and then the execution results are returned to the submitting clients. Upon receiving the results from all executors, the client generates NIZKPs to prove the execution correctness. Finally, the client sends both the execution result and the correctness NIZKPs to the orderers.

Phase 3: Ordering. All orderers run a BFT consensus protocol to determine the order of transactions within each block. Once the order is determined, the orderers disseminate the generated block to all executors for validation.

Phase 4: Validation. On receiving a block from the orderers, the executor sequentially validates each transaction within the block in the determined order. The executor commits a transaction iff the transaction has valid correctness NIZKPs and has no write conflict with previously committed transactions within the same block. Otherwise, the executor aborts the transaction.

4.4 Workflow Description

4.4.1 Client Key Generation

Each client must generate a unique public-private key set for conducting FHE operations. This key set consists of (1) an encryption key pk, (2) a decryption key sk, and (3) an evaluation key ek used for on-ciphertext arithmetic computation. While both pk and ek are open to all participants, sk must remain private to the client. Specifically, Gafe runs the key generation algorithm associated with the chosen FHE scheme (e.g., CKKS [19]), which initially generates sk, and then derives pk and ek from sk. In

Algorithm 3: Transaction execution workflow of the client.

input: Plaintext transaction input data `val`

// Phase 1: Construction

1 $\texttt{pk}_\texttt{s} \leftarrow \texttt{GetEncryptionKey}(\texttt{id}_\texttt{s})$; $\texttt{val}_\texttt{s} \leftarrow \texttt{Encrypt}(\texttt{val},\texttt{pk}_\texttt{s})$;

2 $\texttt{pk}_\texttt{r} \leftarrow \texttt{GetEncryptionKey}(\texttt{id}_\texttt{r})$; $\texttt{val}_\texttt{r} \leftarrow \texttt{Encrypt}(\texttt{val},\texttt{pk}_\texttt{r})$;

3 $\texttt{bal}_\texttt{s} \leftarrow \texttt{GetBalance}(\texttt{id}_\texttt{s})$;

4 $\pi_{\texttt{bal}\geq\texttt{val}} \leftarrow \texttt{GenerateNIZKP}(\texttt{bal}_\texttt{s} \geq \texttt{val}_\texttt{s}, \texttt{sk}_\texttt{s})$;

5 $\texttt{tx} \leftarrow \{\texttt{val}_\texttt{s}, \texttt{val}_\texttt{r}, \pi_{\texttt{bal}\geq\texttt{val}}\}$;

// Phase 2: Execution

6 $\texttt{results} \leftarrow \varnothing$;

7 **For every executor e; do in parallel**

8 $\quad \texttt{bal}'_\texttt{s}, \texttt{bal}'_\texttt{r} \leftarrow \texttt{SendForExecution}(\texttt{e},\texttt{tx})$;

9 $\quad \texttt{results} \leftarrow \texttt{results} \cup \{\texttt{bal}'_\texttt{s}, \texttt{bal}'_\texttt{r}\}$;

10 $\pi_\texttt{s} \leftarrow \texttt{GenerateNIZKP}(\texttt{a majority of } \texttt{bal}'_\texttt{s} \texttt{ in results are consistent})$;

11 $\pi_\texttt{r} \leftarrow \texttt{GenerateNIZKP}(\texttt{a majority of } \texttt{bal}'_\texttt{r} \texttt{ in results are consistent})$;

// Phase 3: Ordering & Phase 4: Validation

12 $\texttt{tx} \leftarrow \texttt{tx} \cup \{\texttt{bal}'_\texttt{s}, \texttt{bal}'_\texttt{r}, \pi_\texttt{s}, \pi_\texttt{r}\}$;

13 $\texttt{SendForOrderingAndValidation}(\texttt{tx})$;

addition, GAFE derives a unique fixed-length string *id* based on *sk* to serve as the client's unique identifier.

4.4.2 GPU-Accelerated Transaction Execution

GAFE's *GPU-accelerated transaction execution* workflow co-designs the execute-order-validate workflow [5] with GPU-accelerated FHE schemes. It consists of four phases, as outlined in Figure 4.1 and Algorithm 3.

Phase 1: Construction. For confidentiality preservation, GAFE processes and stores transaction data in the form of ciphertext. Thus, the client is required to construct an encrypted transaction before submitting it for execution.

Phase 1.1: Encrypting transaction data. The client encrypts transaction input data using the encryption keys. In digital payment, a transaction involves a sender client c_s and a receiver client c_r. Thus, c_s encrypts the payment amount *val* into two ciphertexts, val_s and val_r, using c_s's and c_r's encryption keys, respectively. The reason why two different ciphertexts are generated for *val* is that GAFE employs single-key FHE, which restricts computation to be performed on ciphertexts that are encrypted with the same encryption key. GAFE does not use multi-key FHE, which enables computation on ciphertexts encrypted with different encryption keys. This is because multi-key FHE incurs an extremely high computational overhead [91], impractical for real-world applications.

Phase 1.2: Proving preconditions. In certain applications, the client generates NIZKPs to prove the satisfaction of preconditions that are compulsory for the correct execution of transactions. For instance, digital payment demands that the sender client c_s's balance bal_s must be not less than *val*. To prove this precondition, c_s generates an NIZKP $\pi_{bal\geq val}$ that takes c_s's decryption key as private input, decrypts the two ciphertexts bal_s and val_s, and compares the resulting plaintexts. Thanks to the zero-knowledge

Algorithm 4: Execution phase of the executor.

```
1  buffer ← ∅;
   // Phase 2.1: Verifying preconditions
2  Upon reception of transaction tx from client; do
3  |  {val_s, val_r, π_{bal≥val}} ← tx;
4  |  bal_s ← GetBalance(id_s); ek_s ← GetEvaluationKey(id_s);
5  |  bal_r ← GetBalance(id_r); ek_r ← GetEvaluationKey(id_r);
6  |  if VerifyNIZKP(π_{bal≥val},bal_s,val_s) = true then
7  |  |  buffer ← buffer ∪ {val_s, val_r, bal_s, bal_r, ek_s, ek_r};
8  |  else
9  |  |  Terminate();
   // Phase 2.2: Executing transactions with GPU-accelerated FHE
10 Upon timeout or |buffer| ≥ threshold
11 |  MoveFromMainMeoryToGPUMemory(buffer);
12 |  For every transaction tx in buffer; do in parallel on GPU
13 |  |  bal'_s ← FHESub(bal_s,val_s,ek_s); bal'_r ← FHEAdd(bal_r,val_r,ek_r);
14 |  |  buffer ← buffer ∪ {tx, bal'_s, bal'_r};
15 |  MoveFromGPUMeoryToMainMemory(buffer);
16 |  buffer ← ∅;
   // Phase 2.3: Returning transaction results
17 For every transaction tx in buffer; do in parallel
18 |  ReturnResultToClient(tx,id_s)
```

property of NIZKPs, GAFE preserves the confidentiality for c_s's decryption key and the two resulting plaintexts.

Phase 1.3: Submitting transactions. Lastly, the client submits the transaction to all executors, including the encrypted input and precondition NIZKPs.

Phase 2: Execution. As shown in Figure 4.1, the execution phase comprises four steps. Algorithm 4 outlines Phase 2.1, 2.2, and 2.3 from the executor's viewpoint.

Phase 2.1: Verifying preconditions. Upon receiving a transaction, the executor first verifies the precondition NIZKPs associated with the transaction (e.g., $\pi_{bal \geq val}$). If $\pi_{bal \geq val}$ is invalid, the executor terminates the transaction execution.

Phase 2.2: Executing transactions with GPU-accelerated FHE. Each executor independently executes transactions. Internally, the executor maintains a buffer that stores the transactions received within a predefined time frame (e.g., 500 milliseconds). When a timeout occurs or the number of buffered transactions reaches a specific threshold, the executor (1) moves the encrypted data of all buffered transactions from main memory to GPU memory, (2) launches the corresponding GPU kernels of FHE computation, and (3) copies back the resulting ciphertexts back to main memory. Taking the example of digital payment, the executor first moves the following data to GPU memory: the ciphertexts of the payment amount $\{val_s, val_r\}$, and the sender's and receiver's balances $\{bal_s, bal_r\}$. Next, the executor launches the GPU kernels for FHE addition and subtraction to compute the updated balances: $bal'_s = bal_s - val_s$ for the sender, and $bal'_r = bal_r + val_r$ for the receiver. Inside each kernel, we concurrently perform polynomial arithmetic computations that are highly parallelizable and pervasive in FHE

schemes, such as Number-Theoretic Transform [85]. Note that GAFE is independent of FHE implementation and open to various GPU acceleration techniques [62, 85, 90]. Once the FHE computation is completed, the executor copies the updated balances $\{bal'_s, bal'_r\}$ back to main memory.

Phase 2.3: Returning transaction results. After execution, the executor returns the results to the client who submits the transaction. In the case of digital payment, the executor returns the updated balances $\{bal'_s, bal'_r\}$ to the sender.

Phase 2.4: Proving execution correctness. Upon receiving the results of a transaction from all executors, the clients of the transaction must prove the execution correctness. To achieve this, the clients must be online and generate NIZKPs to prove the consistency of the majority of the results. In digital payment, the sender c_s generates an NIZKP π_s that takes c_s's decryption key as private input, decrypts c_s's updated balance ciphertexts from all the results, and checks the consistency of the resulting plaintexts. The receiver c_r follows a similar procedure and generates an NIZKP π_r using c_r's decryption key to ensure the consistency of c_r's updated balance. Lastly, c_s sends the following materials to the orderers for ordering: the input data $\{val_s, val_r\}$, the precondition NIZKP $\pi_{bal \geq val}$, the consistent results $\{bal_s, bal_r\}$, and the correctness NIZKPs $\{\pi_s, \pi_r\}$.

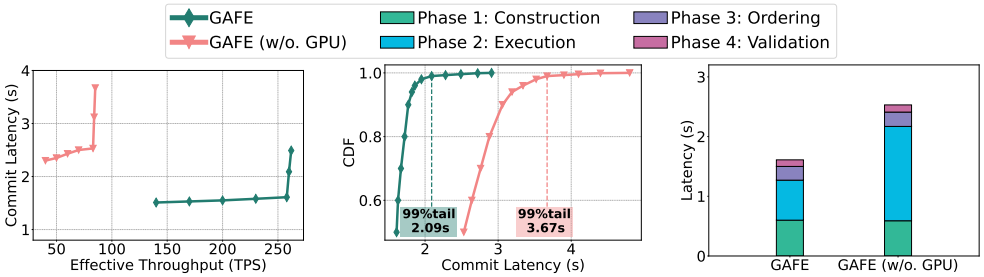
Phase 3: Ordering. GAFE orderers run a BFT consensus protocol (e.g., PBFT [17]) to collectively determine the transaction order within each block. Once a consensus is reached among all orderers, they proceed to generate the block and disseminate the block to all executors for validation.

Phase 4: Validation. When receiving a block from the orderers, the executor sequentially validates all transactions according to their order in the block. The executor will only commit transactions that satisfy two conditions. First, the transaction must not have any write conflict with previously committed transactions within the same block. Second, the transaction must be associated with valid correctness NIZKPs (e.g., $\{\pi_s, \pi_r\}$), which serve as proofs of the transaction's execution correctness. If a transaction fails to meet either of these conditions, the executor aborts the transaction and does not commit it to the state database. Once the executor has validated all transactions, the executor permanently appends the block to the local copy of the blockchain.

4.5 Evaluation

4.5.1 Settings

Implementation. We built a prototype system of GAFE based on HLF v2.5 [5] and simulated the business logic of digital payment. We implemented the CKKS scheme [19] for GAFE based on the state-of-the-art studies on GPU-accelerated FHE [62, 85, 90]. GAFE adopted the gnark [20] library's implementation for the Groth16 NIZKP system [36] and employed our Golang implementation of the PBFT [17] consensus protocol. We also developed a baseline system called GAFE (W/O. GPU) that follows a similar transaction execution workflow as GAFE, except that the baseline does not buffer transactions for concurrent execution and performs all FHE computations exclusively on the CPU.



(a) Effective throughput and commit latency. **(b)** CDF of commit latency. **(c)** Latency of each phase.

Figure 4.2: End-to-end performance of GAFE and the baseline.

Metrics. We evaluated two metrics: (1) *effective throughput*, which indicates the average number of transactions per second (TPS) committed to the blockchain; and (2) *commit latency*, which measures the time duration from transaction construction to commitment. Additionally, we also reported the cumulative distribution function (CDF) of commit latency for all committed transactions and the latency of each phase in the transaction execution workflow (§4.4.2).

Testbed. We ran all evaluations on a cluster of 4 machines, each with an Nvidia RTX 3090 GPU, a 3.1GHz AMD EPYC 9754 CPU, and 64GB of main memory.

4.5.2 End-to-End Performance

We evaluated the end-to-end performance of GAFE and GAFE (W/O. GPU). For each evaluation, we created three executors, four orderers, and one thousand clients. Next, we constructed and submitted 100,000 digital payment transactions. To prevent transaction aborts caused by write conflicts, we explicitly ensured that no two transactions in the same block shared identical clients. We ran the evaluation ten times and reported the average values of the metrics.

GAFE exhibited exceptional end-to-end performance, as shown in Figure 4.2(a). GAFE achieved a high throughput of 258 TPS and a low average latency of 1.61 seconds. In contrast, GAFE (W/O. GPU) displayed a significantly lower average throughput of 83 TPS and a notably longer average latency of 2.53 seconds. These results highlight the substantial performance advantage of GAFE over GAFE (W/O. GPU), with a 3.1× increase in effective throughput and a 37% reduction in commit latency. Additionally, Figure 4.2(b) illustrates that GAFE achieved a shorter 99% tail latency (2.09 seconds) compared to the baseline (3.67 seconds). This suggests that, even in worst-case scenarios, GAFE is capable of completing transaction execution in significantly less time.

GAFE's high performance is attributed to the concurrent FHE computations on GPU. Figure 4.2(c) compares the latency of each phase between GAFE and the baseline. Note that GAFE achieved notably lower latency in the execution phase. The reduced latency is enabled by GPUs' optimized parallel processing capability, facilitating concurrent execution of a significant portion of arithmetic computations in typical FHE schemes. As a result, GAFE avoids performing FHE computation on the CPU, which

has significantly fewer cores and is less efficient in executing a large number of compute-intensive computations in parallel.

4.6 Conclusion

We present Gafe, a confidentiality-preserving blockchain that achieves high performance via the novel GPU-accelerated transaction execution workflow. Gafe protects data confidentiality by encrypting transaction data using FHE, ensures execution correctness by generating lightweight NIZKPs, and achieves high performance by leveraging GPUs to execute transactions concurrently. We implemented Gafe on the codebase of HLF. Our evaluations demonstrated the superior performance of Gafe compared to the baseline, with a significant 3.1× increase in effective throughput (258 TPS) and a notable 37% decrease in commit latency (1.61 seconds).

Chapter 5

GACM: A High-Performance and Confidentiality-Preserving Middleware for Permissioned Blockchains

5.1 Introduction

Blockchains have a profound impact on many industries, such as finance [72] and healthcare [39], thanks to blockchains' outstanding properties like immutability [50]. Among various types of blockchains, permissioned blockchains are favored in enterprise scenarios involving multiple organizations. This is because permissioned blockchains employ an explicit membership identification mechanism, which enforces enhanced access control and facilitates highly efficient consensus protocol like PBFT [17].

However, mainstream permissioned blockchains such as Hyperledger Fabric [5] handle and store data in plaintext, leading to a serious data confidentiality issue. This issue is particularly prominent in multi-organization applications where organization data are highly sensitive and the breach of organization data may cause irreparable economic losses. Furthermore, sensitive organization data such as finance information are subject to strict privacy laws and regulations, such as the California Privacy Rights Act [80] of the United States.

Various work has been proposed in an attempt to address the data confidentiality issue of permissioned blockchains, and the existing work can be categorized into two approaches. The first is the *architecture approach*, which protects sensitive data by devising dedicated blockchain architectures without resorting to cryptographic primitives. This frees the architecture approach from the compute-intensive cryptography computations, thereby ensuring high performance. However, the existing work of the architecture approach does not achieve strong data confidentiality since it essentially stores on-chain data in plaintext, which leaves a huge attack vector for potential data breaches. For example, Caper [3] is susceptible to malicious transaction executors because the transactions are executed over plaintext data.

The second approach is the *cryptography approach*. It leverages various cryptographic primitives such as fully homomorphic encryption (FHE) to encrypt transaction data and non-interactive zero-knowledge proofs (NIZKPs) to ensure the correctness of

transaction execution. FHE allows arithmetic computation to be carried out directly on ciphertexts. NIZKPs empower a prover to prove the validity of a statement without revealing any information beyond the validity itself. Although the cryptography approach achieves strong data confidentiality, it is plagued by low performance, which stems from the substantial computational burden imposed by data encryption, ciphertext computation, and especially NIZKP generation.

Our key insight to address these limitations is that *we can integrate encryption schemes (e.g., FHE) with the explicit membership identification mechanism of permissioned blockchains without relying on expensive cryptographic proofs (e.g., NIZKPs) for proving the execution correctness.* Before submitting a transaction for execution, a client encrypts the transaction data using encryption schemes like FHE to protect data confidentiality. Once the permissioned blockchain receives a transaction from a client, it proceeds to execute the transaction over ciphertext, without having access to the corresponding plaintext. Prior to committing the transaction, the permissioned blockchain sends the transaction result back to the client for collecting the client's endorsement. Specifically, the client decrypts the result to verify the correctness of execution and signs on the result with client's private signing key if the result is correct. By employing the explicit membership identification mechanism, we are able to eliminate the computationally intensive process of generating cryptographic proofs such as NIZKPs, while still maintaining both the confidentiality and correctness of the transaction data.

However, the trivial combination of encryption schemes and permissioned blockchains still faces the performance degradation resulting from data encryption and ciphertext computation. Moreover, the extended transaction execution latency also increases the probability of transaction conflicting aborts within permissioned blockchains that adopt the execute-order-validate (EOV) workflow [5]. Specifically, within each block, EOV blockchains only commit one of all transactions that modify the same key-value pair in the blockchain state database, while aborting the remaining conflicting transactions. This impairs the overall performance of permissioned blockchains as the aborted transactions need to be re-executed. In addition, such a combination requires substantial modifications in the blockchain architectures, rendering this combination incompatible with existing permissioned blockchains such as HLF that have been widely deployed in production environments.

To solve these compatibility and performance issues, our second insight is to *assign the task of transaction execution to an individual middleware that is compatible with various permissioned blockchains and employs transaction merging and GPU acceleration to ensure high performance.* The middleware workflow is divided into three steps. Firstly, the middleware carries out the *FHE Transaction Merging* (FTM) protocol (§5.4), which caches the FHE transactions submitted by clients and merges multiple transactions into a single one without changing their original semantics. Secondly, the middleware runs the *Three-Dimension GPU Acceleration* (TDGA) protocol (§5.4), which uses GPUs to accelerate transaction execution in three dimensions: TDGA concurrently executes (1)

System	Data Encryption	GPU Acceleration	High Performance	Blockchain Compatibility
❖ PDC [32]	×	×	√	×
❖ Caper [3]	×	×	√	×
❖ Qanaat [4]	×	×	√	×
❖ LedgerView [70]	×	×	×	×
◆ PPChain [51]	√	×	×	×
◆ FabZK [46]	√	×	√	×
◆ RFPB [89]	√	×	√	×
◆ GACM	√	√	√	√

Table 5.1: Comparison of GACM and related confidentiality-preserving permissioned blockchains. "❖/◆" represent the architecture approach and the cryptography approach, respectively.

the arithmetic computations within each FHE operation, (2) the independent instructions within each transaction, and (3) the non-conflicting transactions within the cache. Finally, the middleware commits transactions to the underlying blockchain without modifying the blockchain architecture. We will further discuss the middleware workflow in §5.4.

These two insights lead to GACM, a high-performance and confidentiality-preserving middleware for permissioned blockchains. GACM encrypts transaction data using FHE on the client side for strong data confidentiality and uses the explicit membership identification mechanism to ensure the correctness of transaction execution. GACM employs the FTM and TDGA protocols to enhance the performance of FHE transaction execution. As a middleware, GACM is able to be integrated with any permissioned blockchain (particularly the existing ones) without modifying the blockchain architecture.

We built GACM and evaluated its end-to-end performance on top of HLF with three baselines using the SmallBank workload [38]. The results show that GACM attains high performance by employing both the FTM and TDGA protocols, presenting an effective throughput of 425 transactions per second (TPS) and a commit latency of 1180 milliseconds. Compared to the baselines, GACM achieved a maximum 7.61× increases in throughput and a maximum 39.0% reduction in latency.

In summary, we make the following contributions: (1) We effectively integrated encryption schemes into permissioned blockchains by leveraging the explicit membership identification mechanism. (2) We proposed the FTM and TDGA protocols, thereby significantly improving the performance of FHE transaction execution. (4) We implemented and evaluated GACM, demonstrating its effectiveness and high performance.

5.2 Related work

5.2.1 Confidentiality-Preserving Permissioned Blockchains

Existing work of confidentiality-preserving permissioned blockchain falls into two approaches, as shown in Table 5.1.

Architecture approach. Private Data Collection [32] (PDC) is a fine-grained access

control mechanism designed for HLF. Although PDC limits the sharing of private data only within a specific subset of organizations in an attempt to safeguard data confidentiality, PDC is highly vulnerable to the fake PDC result injection attack and the PDC leakage attack, as confirmed in [84]. Caper [3], Qanaat [4], and LedgerView [70] devise specialized data models and confine data access to certain authenticated organizations. However, none of them can prevent the leakage of one organization's data to other organizations.

Cryptography approach. RFPB [89] encrypts data using an additive partial homomorphic encryption (PHE) scheme and ensures the execution correctness via NIZKPs. The additive PHE scheme only permits ciphertext addition and does not support multiplication, which severely restricts the range of compatible applications for RFPB. FabZK [46] combines NIZKPs with a dedicated tabular ledger structure, which also significantly narrows down the types of blockchain applications to which FabZK can be applied. PPChain [51] uses group signature and broadcast encryption to achieve both data confidentiality and provable correctness. However, PPChain is tied to the order-execute (OE) workflow and not compatible with other permissioned blockchain workflow like EOV. Note that GACM is categorized into the cryptography approach.

5.2.2 GPU Acceleration for Blockchains and FHE

Much work has been proposed to separately accelerate blockchains and FHE using GPUs. This attributes to the superior parallel computing capability of GPUs, rendering GPUs particularly suitable for accelerating single-instruction multiple-data computations common in blockchains and FHE. Note that GACM is independent from specific GPU acceleration implementation, allowing GACM to integrate with any available implementations.

GPU acceleration for blockchains. Baldur [43] offloads the task of transaction signature verification to GPUs in an attempt to achieve a higher transaction throughput. Pungila et al. [66] re-designs the storage model of the Merkle Tree, which is a data structures prevalent in blockchains, and optimizes it for GPU computation. Deng et al. [24] utilizes GPUs to accelerate Merkle Patricia Trie (a variant of Merkle Tree) to enable more efficient data lookup and insertion operations.

GPU acceleration for FHE. HE-Booster [85] maps different phases of typical FHE schemes onto the GPU architecture and separately puts forward acceleration methods for both single-GPU and multi-GPU setups. Yang et al. [90] implements three mainstream FHE schemes (BGV [14], BFV [13, 30], and CKKS [19]) on GPU along with diverse engineering optimizations. Ozcan et al. [62] focuses on maximizing the use of the GPU memory hierarchy and minimizing the calls to GPU kernel functions.

5.3 Overview

5.3.1 System Model

A GACM-enabled blockchain network has three types of participants: *client*, *middleware*, and *blockchain node*. GACM adopts the permissioned setting and explicitly identifies the membership of all participants. We describe each type of participant as follows.

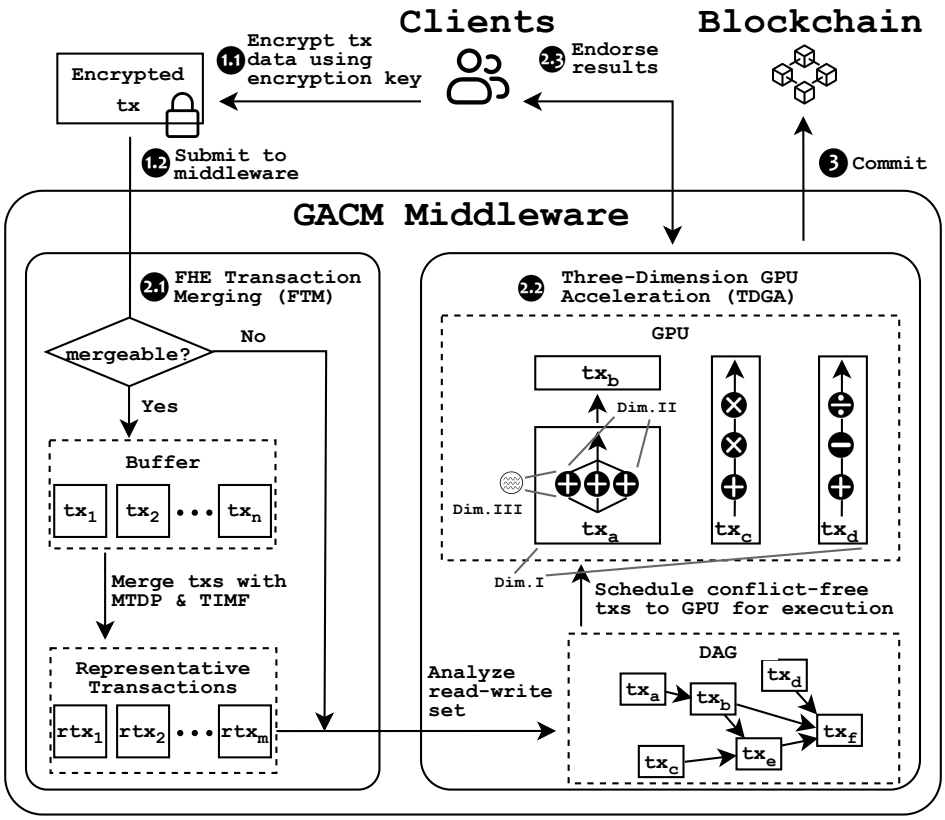


Figure 5.1: The workflow of a GACM-enabled blockchain network.

Client. Each client undertakes three main tasks: (1) Encrypting the transaction input data using the FHE encryption key of the client's organization; (2) Submitting the encrypted transactions to the middleware for execution; (3) Verifying the correctness of the result obtained from the middleware and, if correct, endorsing the result using the client's signing key.

Middleware. The middleware executes the encrypted transactions submitted by clients by carrying out the FTM and TDGA protocols (§5.4). Once a transaction has been executed, the middleware sends the execution result to all the involving organizations whose blockchain state data is modified by the result, in order to collect the endorsement of these organizations. After collecting endorsements from all the involving organizations, GACM forwards the transaction together with execution result to the blockchain nodes.

Blockchain node. Blockchain nodes order, verify, and append transactions received from the middleware. GACM as an middleware treats the underlying blockchain nodes as a blackbox without changing the blockchain architecture, thus making GACM compatible with any permissioned blockchain.

5.3.2 Threat Model

GACM inherits the threat model of the underlying blockchain because GACM treats the underlying blockchain as a blackbox. GACM groups all participants into *organizations*.

Participants within the same organization are mutually trusted and share the same set of FHE keys. However, participants belonging to different organizations does not trust each other and maintain different sets of FHE keys. We make standard assumptions on FHE.

5.3.3 Key Management

A GACM-enabled blockchain network requires two distinct types of cryptographic keys: one for performing FHE operations and the other for endorsing transaction results. Firstly, every organization possesses a unique FHE key set generated by the key-generation algorithm of the FHE scheme adopted by the network. The FHE key set comprises a decryption key dk, an encryption key ek, and an evaluation key ak for performing arithmetic computations directly on FHE ciphertexts. While dk is kept strictly confidential by the organization, both ek and ak are publicly accessible to all blockchain participants. Secondly, each client owns a unique signing key pair for endorsing transaction results produced by the GACM middleware. Prior to joining the blockchain network, the client generate a private signing key sk and a public verifying key vk according to the signature algorithm (e.g., ECDSA [44]) specified by the blockchain network. During the execution phase (§5.4.2), the client uses sk to sign the transaction result as an endorsement and thereafter the GACM middleware uses vk to verify the endorsement signature.

5.4 Workflow

The workflow of a GACM-enabled blockchain network consists of three phases: transaction preparation, execution, and commit, as shown in Figure 5.1 and Algorithm 5. We discuss the workflow in detail.

5.4.1 Phase 1: Preparation

During the preparation phase, the client is responsible for (1) encrypting the transaction data using the encryption key ek of the client's organization and (2) submitting the encrypted transactions to the GACM middleware for transaction execution. Transaction encryption is necessary as it restricts access to sensitive plaintext data exclusively to authorized participants within the client's organization, while preventing unauthorized disclosure of transaction details to other blockchain participants. Furthermore, by implementing client-side encryption, the computational workload is shifted to the client devices, thereby enhancing the performance and scalability of GACM-enabled blockchain networks while maintaining robust data confidentiality protection.

5.4.2 Phase 2: Execution

Definition 5.4.1 (Mergeable transactions and representative transactions)**.** Given any initial blockchain state, for multiple transactions tx_1, tx_2, ..., tx_n that invoke the same smart contract, if there exists a transaction $tx_=$, such that the blockchain state after committing $tx_=$ is equivalent to the blockchain state after committing tx_1, tx_2, ..., tx_n in any order, then tx_1, tx_2, ..., tx_n are *mergeable transactions*, and $tx_=$ is a *representative transaction*.

Algorithm 5: The execution phase of the GACM middleware

```
   // Phase 2.1: FHE Transaction Merging (FTM)
1  buffer ← ∅;
2  Upon reception of transaction tx from client; do
3  |  if IsInvokeMergeableContract(tx) then
4  |  |  buffer ← buffer ∪ tx;
5  |  else
6  |  |  InvokeTDGA(tx);
7  Upon |buffer| ≥ threshold or timeout
8  |  rtx ← MergeTransactions(buffer, MTDP, TIMF);
9  |  InvokeTDGA(rtx);
10 |  buffer ← ∅;
   // Phase 2.2: Three-dimension GPU acceleration (TDGA)
11 Upon invocation of InvokeTDGA(txs); do
12 |  DAG ← AnalyzeDependency(txs);
13 |  While DAG not empty; do
14 |  |  results ← ∅;
15 |  |  cftxs ← GetAllConflictFreeTransactions(DAG);
16 |  |  results ← GPUParallelExecute(cftxs);
17 |  |  RemoveExecutedTransactionsFromDAG(DAG, cftxs);
   |  |  // Phase 2.3: Endorsing results
18 |  |  For every transaction result res in results; do in parallel
19 |  |  |  endorsement ← Endorse(res);
```

Definition 5.4.2 (Mergeable smart contract)**.** A smart contract invoked by mergeable transactions is a *mergeable smart contract*. Otherwise, the smart contract is *non-mergeable*.

As shown in Figure 5.1, the execution phase consists of three sub-phases: First, the GACM middleware successively executes the FTM protocol (**Phase 2.1**) and the TDGA protocol (**Phase 2.2**) to accelerate transaction execution and produce transaction results. Subsequently, the GACM middleware sends these results back to clients for endorsement (**Phase 2.3**).

Phase 2.1: FHE transaction merging (FTM). The FTM protocol has two sub-protocols: (1) *offline declaration* and (2) *online merging*.

Phase 2.1.1: Offline declaration. During contract deployment, the smart contract developers declare whether the smart contract is a mergeable smart contract (Definition 5.4.2). For mergeable smart contracts, the developer needs to declare the *mergeable transactions detection predicate* (MTDP) and the *transaction inputs merging function* (TIMF). MTDP and TIMF are used by the online merging sub-protocol to detect mergeable transactions and merge the input of mergeable transactions, respectively.

Phase 2.1.2: Online merging. Algorithm 5 outlines the online merging sub-protocol. Upon receiving a FHE transaction from a client, the GACM middleware first determines whether the transaction invokes a mergeable smart contract. For a mergeable smart contract SC_M, the GACM middleware caches all transactions that invokes SC_M and are submitted within a predefined time frame, for instance, 500 milliseconds. These

transactions are cached into a transaction buffer that is dedicated for caching transactions invoking SC_M. Once the number of cached transactions surpasses a pre-set threshold or a timeout event occurs, the GACM middleware divides all cached transactions into separate groups of mergeable transactions (Definition 5.4.1) based on MTDP. Then, the GACM middleware uses TIMF to generate the corresponding representative transaction (Definition 5.4.1) for each group of mergeable transactions. Finally, the GACM middleware dispatches the representative transactions to the TDGA protocol (Phase 2.2) for transaction execution. In the case of a non-mergeable smart contract, the GACM middleware bypasses the aforementioned merging procedures. Instead, it directly forwards the transaction to Phase 2.2 for further processing.

Phase 2.2: Three-dimension GPU acceleration (TDGA). As shown in Algorithm 5, the GACM middleware carries out the TDGA protocol to execute FHE transactions with GPU acceleration and generate the corresponding execution result. Specifically, TDGA accelerates FHE transaction execution in three dimensions:

Dimension I: transaction parallelism. TDGA concurrently executes multiple conflict-free transactions. Note that transactions are conflict-free iff they have no read-write conflict and no write-write conflict. Before transaction execution, TDGA runs a read-write set analyzer to analyze the data dependency among all transactions. Then, TDGA schedules transactions into a directed acyclic graph (DAG) based on the data dependency and chronological order. Subsequently, TDGA launches multiple GPU streams [60] to execute multiple conflict-free transactions in parallel following the topological order determined by the DAG. It is important to note that in scenarios where, for instance, transaction tx_B has a dependency on transaction tx_A, the TDGA protocol will execute tx_B only after tx_A has either been successfully committed to the blockchain or has been aborted by it. This guarantees that all transaction results can be successfully committed to the blockchain, effectively preventing any conflicting updates to the same blockchain state data.

Dimension II: statement parallelism. TDGA performs parallel execution of multiple conflict-free statements within the same transaction. During smart contract deployment, developers are required to make static declarations of such conflict-free statements. At runtime, TDGA leverages these declarations and launches a series of GPU thread blocks [60] to concurrently execute conflict-free statements. For example, in the well-known SmallBank [42] benchmark, the TRANSFER smart contract contains two conflict-free statements. One statement is responsible for deducting funds from the sender's account, and the other is tasked with increasing the balance of the receiver's account. By utilizing statement parallelism, TDGA can execute these two statements in parallel, thereby reducing the latency of transaction execution.

Dimension III: FHE operator parallelism. TDGA runs multiple GPU threads [60] to execute in parallel the internal computations (e.g., Number-Theoretic Transformation [85]) within individual FHE operator, thanks to the highly parallelizable architecture of FHE operators [14, 13, 30, 19]. Note that TDGA is independent of specific FHE implementation and can be integrated with a diverse range of GPU acceleration

	GACM			GACM (w/o. FTM)			GACM (w/o. TDGA)			GACM (null)		
	tput	lat	ar	tput	lat	ar	tput	lat	ar	tput	lat	ar
Q1: Balance	334	1180	0%	310	1492	0%	73	1484	13%	61	1886	24%
Q2: DepositChecking	418	1320	0%	303	1771	0%	79	1599	17%	57	2075	26%
Q3: TransactSaving	425	1246	0%	294	1697	0%	81	1705	11%	58	2041	23%
Q4: Amalgamate	312	2041	0%	281	2246	0%	63	2312	10%	54	2501	23%
Q5: WriteCheck	411	2223	0%	298	2468	0%	76	2328	11%	54	2697	21%

Table 5.2: The end-to-end performance of GACM and the three baseline systems under HLF and the SmallBank workload. "tput" denotes the effective throughput and is measured in transactions per second (TPS). "lat" refers to the transaction commit latency and is measured in milliseconds. "ar" stands for the transaction abort rate.

methods [62, 85, 90].

Phase 2.3: Endorsing results. Upon completion of the transaction execution, GACM requests the client to verify and endorse the execution result. Specifically, the client re-executes the transaction using the plaintext transaction input and generates an execution result denoted as R_C. Subsequently, the client decrypts the execution result R_M produced by GACM. The client deems R_M to be correct iff R_C is equivalent to R_M. In the case where R_M is determined to be correct, the client uses his signing key (as described in §5.3.3) to signs R_M. The resulting signature is then sent to GACM as an endorsement.

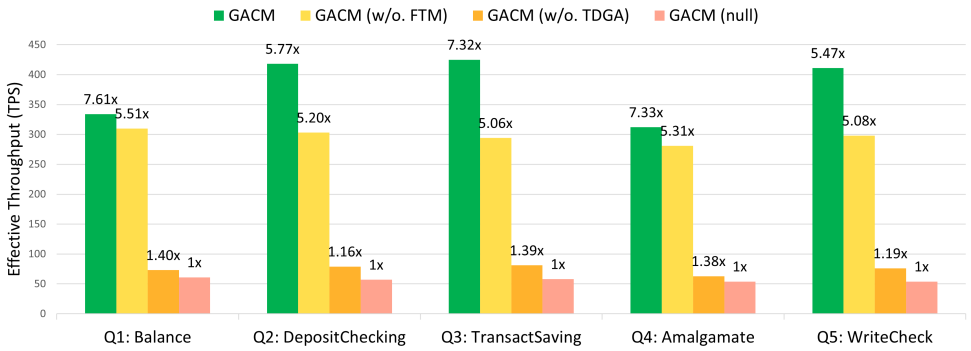
5.4.3 Phase 3: Commit

Upon receiving the endorsement, GACM commits R_M along with the associated transaction to the blockchain. Conversely, if R_M is found to be incorrect, or if GACM fails to receive an endorsement from the client, GACM will mark R_M as faulty and directly abort the associated transaction. Once the transactions have either been successfully committed to the blockchain or have been aborted, GACM is responsible to inform the clients who submitted those transactions.

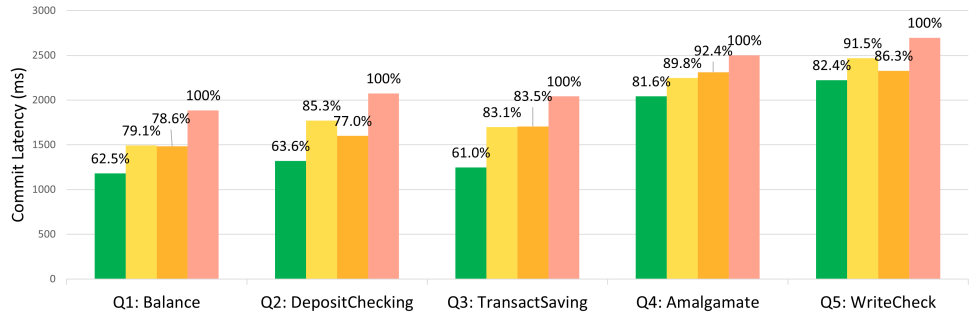
5.5 Evaluation

5.5.1 Settings

We used Golang to build the GACM prototype system equipped with both the FTM and TDGA protocols. To evaluate the performance improvements brought by the FTM and TDGA protocol, we also developed three baseline systems with a workflow similar to GACM: GACM (w/o. FTM), GACM (w/o. TDGA), and GACM (null). (1) GACM (w/o. FTM) does not implement the FTM protocol. Instead, GACM (w/o. FTM) forwards the transactions to the TDGA protocol once it receives transactions from clients. (2) GACM (w/o. TDGA) does not implement the TDGA protocol. GACM (w/o. TDGA) only achieves transaction parallelism (Dimension I), without implementing the concurrent execution of statements (Dimension II) and FHE operators (Dimension III). (3) GACM (null) implements neither the FTM protocol nor the TDGA protocol. For the FHE scheme, we adopted the state-of-the-art GPU-accelerated implementation [62, 85, 90] of the CKKS scheme [19]. We used the SmallBank workload [42] as the performance benchmark.



(a) Effective throughput



(b) Commit latency

Figure 5.2: Comparison of effective throughput and commit latency among GACM, GACM (w/o. FTM), GACM (w/o. TDGA), and GACM (null).

In our evaluation, we monitored three crucial metrics: the *effective throughput*, the *commit latency*, and the *abort rate*. For the effective throughput, we measured the average number of client transactions per second (TPS) that were successfully commited to the blockchain. The commit latency denotes the time interval starting from the preparation of a transaction and ending with its successful commit. The abort rate represents the proportion of client transactions that were aborted as a result of read-write conflicts.

We conducted evaluations within a cluster consisting of 16 machines. Each machine has a 3.5GHz AMD EPYC 7763 CPU and a 128GB of main memory. Specifically, one machine has an NVIDIA A100 GPU with 80GB GPU memory and is dedicated to running the GACM middleware.

5.5.2 End-to-End Performance

We evaluated the end-to-end performance of GACM, GACM (w/o. FTM), GACM (w/o. TDGA), and GACM (null) on top of Hyperledger Fabric [5] (HLF). We deployed one instance of the GACM middleware, three HLF executors, four HLF orderers, and a thousand clients. We carried out all evaluations ten times and reported the average values of the metrics. For each type of query in SmallBank, we generated and submitted one million transactions, with a predefined transaction conflict rate of 20%.

GACM exhibits outstanding end-to-end performance, as shown in Table 5.2 and

Figure 5.2. When compared with the baseline systems, GACM demonstrates an effective throughput that is up to 7.61 times higher and a commit latency that is at most 39.0% lower. Moreover, GACM attains a transaction abort rate of 0%, while the abort rates of GACM (w/o. TDGA) and GACM (null) are as high as 17% and 26%, respectively.

The high-performance of GACM can be ascribed to two key aspects. First, the FTM protocol in GACM (§5.4.2) not only reduces the number of computationally intensive FHE transactions that need to be executed, but also decreases the likelihood of conflicting aborts. This can be confirmed by the performance gain of GACM (w/o. TDGA), which achieves an effective throughput that is up to 1.40× higher and a commit latency that is 23% lower. Secondly, the TDGA protocol in GACM (§5.4.2) leverages GPU to concurrently execute conflict-free FHE transactions in three distinct dimensions. This allows TDGA to eliminate transaction conflicting aborts and realize significant performance enhancements. As illustrated in Figure 5.2, GACM (w/o. FTM) achieves a throughput that is up to 5.51× higher and a latency that is 20.9% lower.

5.6 Conclusion

We present GACM, a high-performance and confidentiality-preserving middleware for permissioned blockchains. GACM safeguards data confidentiality by effectively integrating FHE schemes into permissioned blockchains. GACM significantly improves the performance of FHE transaction execution via merging mergeable transactions (FTM protocol) and leveraging GPU to accelerate FHE transaction execution from three dimensions (TDGA protocol). We implemented and evaluated GACM. Our evaluations demonstrate that GACM achieves outstanding performance compared to the baselines, with a significant 7.61× increase in effective throughput and a notable 39.0% decrease in commit latency.

Chapter 6

Conclusion and Future Work

TODO

Bibliography

[1] Abbas Acar et al. "A Survey on Homomorphic Encryption Schemes: Theory and Implementation". In: *ACM Comput. Surv.* 51.4 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3214303. URL: https://doi.org/10.1145/3214303.

[2] Abbas Acar et al. "A survey on homomorphic encryption schemes: Theory and implementation". In: *ACM Computing Surveys* 51.4 (2018), pp. 1–35.

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. "Caper: a cross-application permissioned blockchain". In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1385–1398.

[4] Mohammad Javad Amiri et al. "Qanaat: A scalable multi-enterprise permissioned blockchain system with confidentiality guarantees". In: *arXiv preprint arXiv:2107.10836* (2021).

[5] Elli Androulaki et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the thirteenth EuroSys conference*. New York, NY, USA: ACM, 2018, pp. 1–15.

[6] Elli Androulaki et al. "Privacy-preserving auditable token payments in a permissioned blockchain system". In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. 2020, pp. 255–267.

[7] Tomaso Aste, Paolo Tasca, and Tiziana Di Matteo. "Blockchain technologies: The foreseeable impact on society and industry". In: *computer* 50.9 (2017), pp. 18–28.

[8] Roberto Baldoni et al. "A survey of symbolic execution techniques". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39.

[9] Nick Baumann et al. *zkay v0.2: Practical Data Privacy for Smart Contracts*. 2020. arXiv: 2009.01020. URL: https://arxiv.org/abs/2009.01020.

[10] 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*. IEEE. 1730 Massachusetts Ave., NW Washington, DCUnited States: IEEE Computer Society, 2014, pp. 355–362.

[11] Nir Bitansky et al. "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*. New York, NY, USA: ACM, 2012, pp. 326–349.

[12] Dan Boneh and Victor Shoup. "A graduate course in applied cryptography". In: *Draft 0.5* (2020).

[13] 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*. Springer. Heidelberg: Springer Berlin, 2012, pp. 868–886.

[14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ACM Transactions on Computation Theory (TOCT)* 6.3 (2014), pp. 1–36.

[15] Benedikt Bünz et al. "Zether: Towards privacy in a smart contract world". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2020, pp. 423–443.

[16] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[17] Miguel Castro, Barbara Liskov, et al. "Practical byzantine fault tolerance". In: *OSDI*. Vol. 99. New York, NY, USA: ACM, 1999, pp. 173–186.

[18] Raymond Cheng et al. "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 185–200.

[19] Jung Hee Cheon et al. "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*. Springer. Hong Kong, China: Springer, 2017, pp. 409–437.

[20] consensys. *Gnark*. 2023. URL: https://docs.gnark.consensys.net/.

[21] ConsenSys. *Quorum*. 2021. URL: https://consensys.net/quorum.

[22] P David Coward. "Symbolic execution systems—a review". In: *Software Engineering Journal* 3.6 (1988), pp. 229–239.

[23] 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*. Springer. 2019, pp. 344–373.

[24] Yangshen Deng, Muxi Yan, and Bo Tang. "Accelerating Merkle Patricia Trie with GPU". In: *Proceedings of the VLDB Endowment* 17.8 (2024), pp. 1856–1869.

[25] Thi Van Thao Doan et al. "A survey on implementations of homomorphic encryption schemes". In: *The Journal of Supercomputing* (2023), pp. 1–42.

[26] 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)*. IEEE. Halifax, Nova Scotia, Canada: IEEE, 2018, pp. 1084–1091.

[27] 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.

[28] Hyperledger Fabric. *Fabric Gateway*. URL: https://hyperledger-fabric.readthedocs.io/en/latest/gateway.html.

[29] Hyperledger Fabric. *Hyperledger/fabric at release-2.5*. 2023. URL: https://github.com/hyperledger/fabric/tree/release-2.5.

[30] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144. 2012. URL: https://eprint.iacr.org/2012/144.

[31] Uriel Feige, Dror Lapidot, and Adi Shamir. "Multiple noninteractive zero knowledge proofs under general assumptions". In: *SIAM Journal on computing* 29.1 (1999), pp. 1–28.

[32] The Hyperledger Foundation. *Private Data Collections: A high-level Overview – Hyperledger Foundation*. 2018. URL: https://www.hyperledger.org/blog/2018/10/23/private-data-collections-a-high-level-overview.

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

[34] Arthur Gervais et al. "On the security and performance of proof of work blockchains". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 3–16.

[35] Christian Gorenflo et al. "FastFabric: Scaling hyperledger fabric to 20 000 transactions per second". In: *International Journal of Network Management* 30.5 (2020), e2099.

[36] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances in Cryptology – EUROCRYPT 2016*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326. ISBN: 978-3-662-49896-5.

[37] Jens Groth and Mary Maller. "Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs". In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Cham: Springer International Publishing, 2017, pp. 581–612. ISBN: 978-3-319-63715-0.

[38] H-Store. *H-store: SmallBank benchmark*. 2013. URL: https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[39] Change Healthcare. *Change healthcare case study*. 2023. URL: https://www.hyperledger.org/learn/publications/changehealthcare-case-study.

[40] Mike Hearn and Richard Gendal Brown. "Corda: A distributed ledger". In: *Corda Technical White Paper* 2016 (2016).

[41] Daira Hopwood et al. "Zcash protocol specification". In: *GitHub: San Francisco, CA, USA* 4.220 (2016), p. 32.

[42] Hyperledger. *SmallBank*. 2020. URL: https://github.com/hyperledger/caliper-benchmarks/tree/master/benchmarks/scenario/smallbank.

[43] Rares Ifrim, Dumitrel Loghin, and Decebal Popescu. "Baldur: A Hybrid Blockchain Database with FPGA or GPU Acceleration". In: *Proceedings of the 1st Workshop on Verifiable Database Systems*. 2023, pp. 19–27.

[44] Don Johnson, Alfred Menezes, and Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International journal of information security* 1 (2001), pp. 36–63.

[45] Harry Kalodner et al. "Arbitrum: Scalable, private smart contracts". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1353–1370.

[46] Hui Kang et al. "Fabzk: Supporting privacy-preserving, auditable smart contracts in hyperledger fabric". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. Portland, Oregon, USA: IEEE, 2019, pp. 543–555.

[47] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[48] Sunny King and Scott Nadal. "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake". In: *self-published paper, August* 19.1 (2012).

[49] Ahmed Kosba et al. "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts". In: *2016 IEEE symposium on security and privacy (SP)*. IEEE. 2016, pp. 839–858.

[50] Esteban Landerreche and Marc Stevens. "On immutability of blockchains". In: *Proceedings of 1st ERCIM Blockchain Workshop 2018*. European Society for Socially Embedded Technologies (EUSSET). 2018.

[51] Chao Lin et al. "Ppchain: A privacy-preserving permissioned blockchain architecture for cryptocurrency and other regulated applications". In: *IEEE Systems Journal* 15.3 (2020), pp. 4367–4378.

[52] Medicalchain. *MedicalChain*. https://medicalchain.com. 2022.

[53] Andreas V Meier. "The elgamal cryptosystem". In: *Joint Advanced Students Seminar*. 2005.

[54] *Microsoft SEAL (release 4.0)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Mar. 2022.

[55] Monero. *The Monero Project*. URL: https://www.getmonero.org/.

[56] Christian Vincent Mouchet et al. "Lattigo: A multiparty homomorphic encryption library in Go". In: *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*. .: HomomorphicEncryption.org, 2020, pp. 6. 64–70. DOI: https://doi.org/10.25835/0072999. URL: http://infoscience.epfl.ch/record/299025.

[57] Ujan Mukhopadhyay et al. "A brief survey of cryptocurrency systems". In: *2016 14th annual conference on privacy, security and trust (PST)*. IEEE. 2016, pp. 745–752.

[58] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[59] NCCGroup. 2016. URL: https://research.nccgroup.com/wp-content/uploads/2021/06/NCC_Group_ProtocolLabs_FilecoinGroth16_Report_2021-06-02.pdf.

[60] Nvidia. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. 2025.

[61] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 USENIX annual technical conference (USENIX ATC 14)*. 2014, pp. 305–319.

[62] Ali Şah Özcan et al. “Homomorphic Encryption on GPU”. In: *IEEE Access* (2023).

[63] Pascal Paillier. “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In: *Advances in Cryptology — EUROCRYPT ’99*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-48910-8.

[64] Jianli Pan et al. “EdgeChain: An Edge-IoT Framework and Prototype Based on Blockchain and Smart Contracts”. In: *IEEE Internet of Things Journal* 6.3 (2019), pp. 4719–4732. DOI: 10.1109/JIOT.2018.2878154.

[65] Zeshun Peng et al. “NeuChain: a fast permissioned blockchain system with deterministic ordering”. In: *Proc. VLDB Endow.* 15.11 (July 2022), 2585–2598. ISSN: 2150-8097. DOI: 10.14778/3551793.3551816. URL: https://doi.org/10.14778/3551793.3551816.

[66] Ciprian Pungila and Viorel Negru. “Improving blockchain security validation and transaction processing through heterogeneous computing”. In: *International Joint Conference: 12th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2019) and 10th International Conference on EUropean Transnational Education (ICEUTE 2019) Seville, Spain, May 13th-15th, 2019 Proceedings 12*. Springer. 2020, pp. 132–140.

[67] Ji Qi et al. “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. Virtual Event, Germany: Association for Computing Machinery, 2021, 18–34. ISBN: 9781450387095. DOI: 10.1145/3477132.3483574. URL: https://doi.org/10.1145/3477132.3483574.

[68] Mateusz Raczyński. *What is the fastest blockchain and why? analysis of 43 blockchains*. 2021. URL: https://alephzero.org/blog/what-is-the-fastest-blockchain-and-why-analysis-of-43-blockchains/.

[69] Pingcheng Ruan et al. “A Transactional Perspective on Execute-Order-Validate Blockchains”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. Portland, OR, USA: Association for Computing Machinery, 2020, 543–557. ISBN: 9781450367356. DOI: 10.1145/3318464.3389693. URL: https://doi.org/10.1145/3318464.3389693.

[70] Pingcheng Ruan et al. “LedgerView: access-control views on hyperledger fabric”. In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 2218–2231.

[71] Fahad Saleh. “Blockchain without waste: Proof-of-stake”. In: *The Review of financial studies* 34.3 (2021), pp. 1156–1190.

[72] Fabian Schär. “Decentralized finance: On blockchain-and smart contract-based financial markets”. In: *FRB of St. Louis Review* (2021).

[73] Ankur Sharma et al. “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. Amsterdam, Netherlands: Association for Computing Machinery, 2019, 105–122. ISBN: 9781450356435. DOI: 10.1145/3299869.3319883. URL: https://doi.org/10.1145/3299869.3319883.

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

[75] 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*. 2022, pp. 2735–2749.

[76] Samuel Steffen et al. "ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs". In: *2022 IEEE Symposium on Security and Privacy (SP)*. San Francisco, California, USA: IEEE, 2022, pp. 179–197. DOI: 10.1109/SP46214.2022.9833732.

[77] Samuel Steffen et al. "Zkay: Specifying and Enforcing Data Privacy in Smart Contracts". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, 1759–1776. ISBN: 9781450367479. DOI: 10.1145/3319535.3363222. URL: https://doi.org/10.1145/3319535.3363222.

[78] Pascal Strebel. *Implementation of a Cloud-Based Mobile Application for Time Tracking*. Bachelor's Thesis. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Pascal_Strebel_BA_Report.pdf. ETH Zurich, 2023.

[79] Xiaoqiang Sun et al. "A survey on zero-knowledge proof in blockchain". In: *IEEE network* 35.4 (2021), pp. 198–205.

[80] *The California Privacy Rights Act*. URL: https://thecpra.org.

[81] *The Z3 Theorem Prover (release 4.3.13)*. https://github.com/Z3Prover/z3. Microsoft Research, Redmond, WA. Oct. 2024.

[82] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. "SoK: Fully Homomorphic Encryption Compilers". In: *2021 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2021, pp. 1092–1108. DOI: 10.1109/SP40001.2021.00068.

[83] Paul Voigt and Axel Von dem Bussche. "The eu general data protection regulation (gdpr)". In: *A Practical Guide, 1st Ed., Cham: Springer International Publishing* 10.3152676 (2017), pp. 10–5555.

[84] Shan Wang et al. "On private data collection of hyperledger fabric". In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021, pp. 819–829.

[85] Zhiwei Wang et al. "HE-Booster: An Efficient Polynomial Arithmetic Acceleration on GPUs for Fully Homomorphic Encryption". In: *IEEE Transactions on Parallel and Distributed Systems* 34.4 (2023), pp. 1067–1081.

[86] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[87] Huixin Wu and Feng Wang. "A survey of noninteractive zero knowledge proof system and its applications". In: *The scientific world journal* 2014.1 (2014), p. 560484.

[88] Yang Xiao et al. "A survey of distributed consensus protocols for blockchain networks". In: *IEEE Communications Surveys & Tutorials* 22.2 (2020), pp. 1432–1465.

[89] Lei Xu, Yuewei Zhang, and Liehuang Zhu. "Regulation-Friendly Privacy-Preserving Blockchain Based on zk-SNARK". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2023, pp. 167–177.

[90] Hao Yang et al. "Phantom: A CUDA-Accelerated Word-Wise Homomorphic Encryption Library". In: *IEEE Transactions on Dependable and Secure Computing* (2024), pp. 1–12. DOI: 10.1109/TDSC.2024.3363900.

[91] Minghao Yuan et al. "An Examination of Multi-Key Fully Homomorphic Encryption and Its Applications". In: *Mathematics* 10.24 (2022), p. 4678.

[92] Tao Zhang and Zhigang Huang. "Blockchain and central bank digital currency". In: *ICT Express* 8.2 (2022), pp. 264–270.