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

Rongxin Guan, Tianxiang Shen, Sen Wang, Gong Zhang, Heming Cui, Ji Qi*

Huawei Technologies

The University of Hong Kong

Institute of Software,
Chinese Academy of Sciences

Background

Confidentiality Issue

- Notable blockchains like Hyperledger Fabric process and store data in plaintext, **exposing sensitive data** to anyone with access to the blockchain.
- Sensitive data of safety-critical applications such as finance and healthcare should be accessible to the data owners **only**.

A Promising Appraoch

- We can address the confidentiality issue with Homomorphic Encryption (**HE**) and Non-Interactive Zero-Knowledge Proofs (**NIZKPs**).
- HE enables arithmetic computation to be performed directly on ciphertexts.
- NIZKPs enables proving a statement without revealing any information beyond the validity of the statement itself.

Related Work and Their Deficiencies

Related Work

- **General-Purpose Blockchain**
  - ZeeStar
  - Arbitrum
  - Ekiden
  - Hawk
- **Specific-Purpose Blockchain**
  - Zcash
  - Monero
  - FabZK
  - Zether
  - RFPB

Deficiencies

- **Low Performance**
  - The utilization of HE and NIZKPs incurs **intensive computational costs** to blockchains.
  - For example, ZeeStar [1], a notable confidentiality-preserving blockchain, exhibits a long commit latency of tens of seconds.
- **No Support for General Smart Contracts**
  - Most prior work relies on partially homomorphic encryption (PHE) that supports **either addition or multiplication** on ciphertexts.

Insights

Our Question

Can we achieve both high performance and general smart contract support for confidentiality-preserving blockchains?

Our First Insight

*We can efficiently integrate fully homomorphic encryption (**FHE**) and blockchains by introducing **GPU acceleration** for transaction execution and FHE computation.*

FHE

- FHE supports both addition and multiplication on ciphertexts, making FHE ideal for supporting general smart contracts.

GPU acceleration

- Blockchain transactions that invoke the same smart contract are highly parallelizable.
- GPU offers superior parallel processing capabilities.

Insights

The Ciphertext Inconsistency Problem

When given identical inputs, different blockchain nodes perform FHE arithemtic calculation and generate inconsistent ciphertexts for the same plaintext result. This is because FHE schemes intentionally introduce random noise to prevent attackers from extracting information from ciphertexts.

Our Second Insight

*We can integrate **lightweight NIZKPs** with the **trusted execution mechanism** of the execute-order-validate blockchain workflow.*

Trusted Execution Mechanism

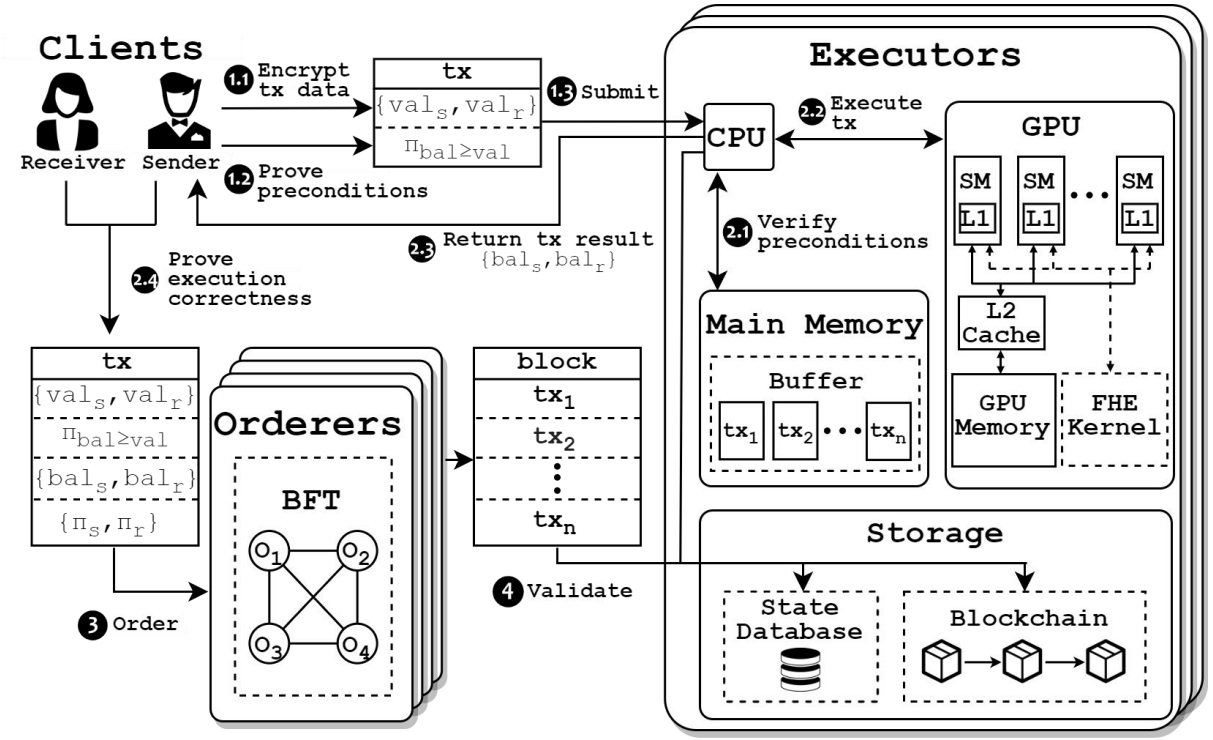
- This mechanism first executes a transaction on multiple nodes and checks iff a majority of nodes produce consistent results.

Lightweight NIZKPs

- NIZKPs decrypt all ciphertexts and check the consistency of quorom plaintexts.
- **No** costly FHE arithmetic calculation is involved inside NIZKPs.

Contributions

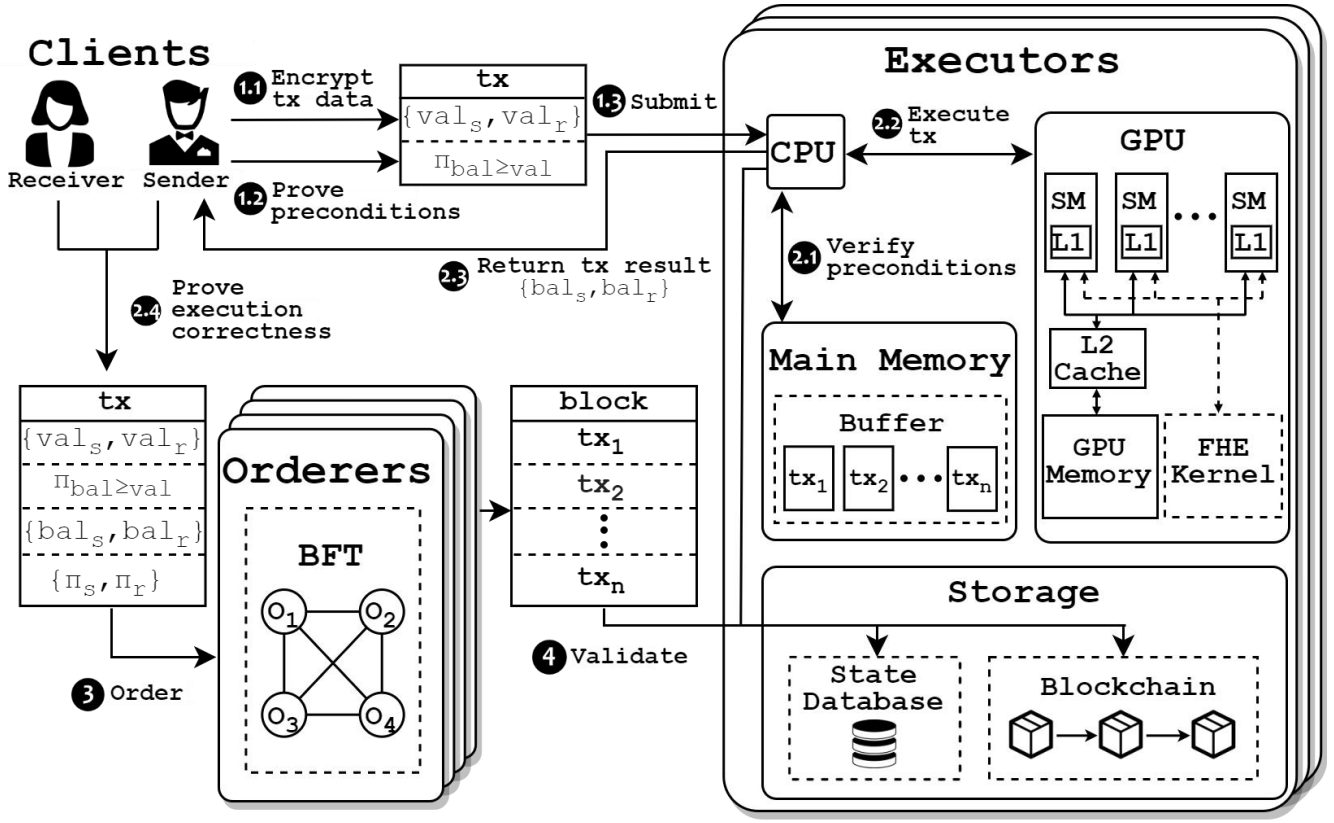
1. We propose a **GPU-accelerated transaction execution workflow** that integrates GPU-accelerated FHE into blockchain and ensures execution correctness through lightweight NIZKPs.
2. We implement a a high-performance confidentiality-preserving blockchain **prototype system** named GAFE that incorporates the aforementioned workflow.
3. We conduct **end-to-end evaluations** on GAFE to demonstrate its effectiveness and high performance.



System Overview

System Model

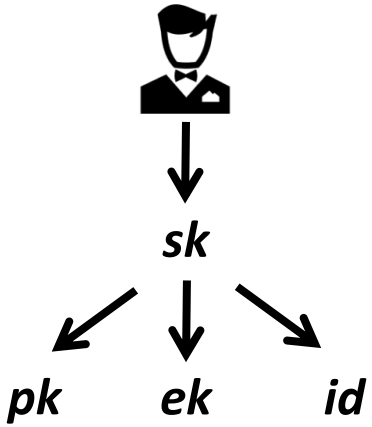
- **Clients**
  - Encrypt transaction input data;
  - Submit the encrypted transactions for execution;
  - Prove the execution correctness;
- **Executors**
  - Execute encrypted transactions with GPU acceleration;
  - Validate transaction results;
  - Maintain the latest local copy of blockchain and state database;
- **Orderers**
  - Run a BFT protocol to determine the order of transactions within each block;



Client Key Distribution

Before each client is ready to join the GAFE blockchain network, the client must run the key generation algorithm associated with the FHE scheme to generate a unique **public-private key set** for conducting FHE operations.

1. Firstly, the client generates a secret key ***sk*** for ciphertext decryption.
2. Secondly, the client derives a public key ***pk*** for plaintext encryption and a public key ***ek*** for on-ciphertext arithmetic computation from ***sk***.
3. Additionally, the client derives a unique fixed-length string ***id*** based on ***sk*** to serve as the client's unique identifier



Workflow Overview

Phase 1: Construction.

Clients encrypt transaction data, generate precondition NIZKPs, and submit the transaction.

Phase 2: Execution.

Executors independently execute transactions and accelerate execution with GPU.

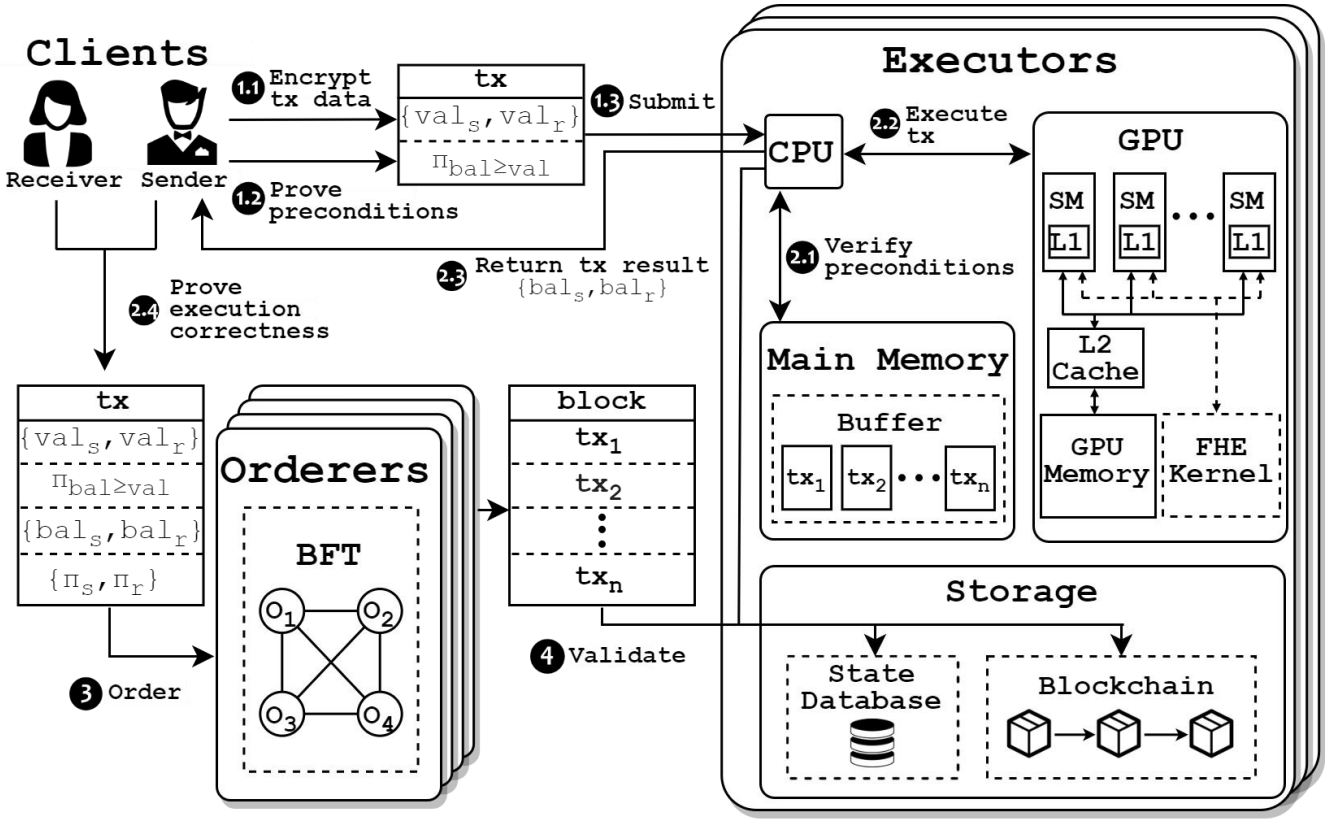
Clients generate NIZKPs to prove the execution correctness.

Phase 3: Ordering.

Orderers run a BFT to determine the order of transactions within each block, and disseminate the generated block to all executors for validation.

Phase 4: Validation.

Executors sequentially validate each transaction within the block in the determined order and commit valid transactions.



Workflow Phase 1: Construction

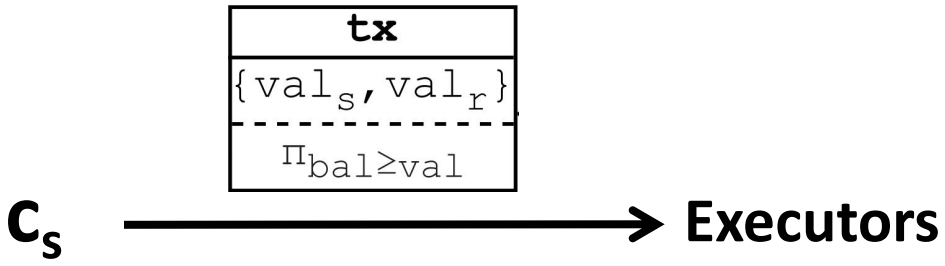
Phase 1.1: Encrypting transaction data.

Clients encrypt transaction input data using the encryption keys.

Phase 1.2: Proving preconditions.

$\pi_{bal \geq val}$

Phase 1.3: Submitting transactions.



Algorithm 1: Transaction execution workflow of the client.

input: Plaintext transaction input data val

// Phase 1: Construction

1 $pk_s \leftarrow$ GetEncryptionKey(id_s); $val_s \leftarrow$ Encrypt(val,pk_s);
2 $pk_r \leftarrow$ GetEncryptionKey(id_r); $val_r \leftarrow$ Encrypt(val,pk_r);
3 $bal_s \leftarrow$ GetBalance(id_s);
4 $\pi_{bal \geq val} \leftarrow$ GenerateNIZKP($bal_s \geq val_s$, sk_s);
5 $tx \leftarrow \{val_s, val_r, \pi_{bal \geq val}\}$;

// Phase 2: Execution

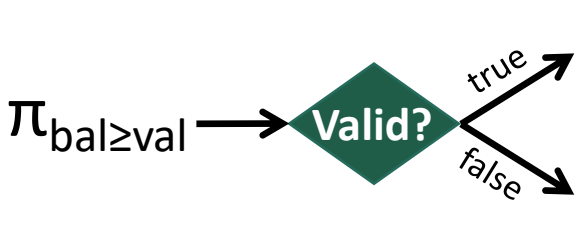
6 results $\leftarrow \emptyset$;
7 **For every executor e; do in parallel**
8 $\quad bal'_s, bal'_r \leftarrow$ SendForExecution(e,tx);
9 $\quad$ results $\leftarrow$ results $\cup \{bal'_s, bal'_r\}$;
10 $\pi_s \leftarrow$ GenerateNIZKP(a majority of bal'_s in results are consistent);
11 $\pi_r \leftarrow$ GenerateNIZKP(a majority of bal'_r in results are consistent);

// Phase 3: Ordering & Phase 4: Validation

12 $tx \leftarrow tx \cup \{bal'_s, bal'_r, \pi_s, \pi_r\}$;
13 SendForOrderingAndValidation(tx);

Workflow Phase 2: Execution

Phase 2.1: Verifying preconditions.



Add transactions to the buffer for waiting parallel execution on GPU.

Terminate execution.

Phase 2.2: Executing transactions with GPU-accelerated FHE.

When a timeout occurs or the number of buffered transactions reaches a specific threshold, the executor performs the following three steps:

1. Moves all buffered transactions from main memory to GPU memory;
2. Launches the corresponding GPU kernels of FHE computation.
3. Copies back the resulting ciphertexs back to main memory.

Phase 2.3: Returning transaction results.

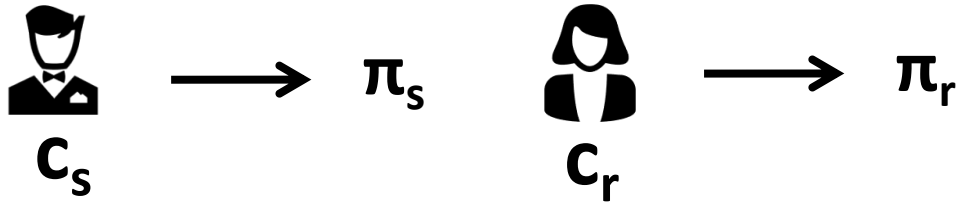
Algorithm 2: Execution phase of the executor.

```
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)
```

Workflow Phase 2: Execution

Phase 2.4: Proving execution correctness.

- The clients of the transaction generate NIZKPs to prove the consistency of the majority of the results.
- Take the example of 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 πr using c_r's decryption key to ensure the consistency of c_r's updated balance.



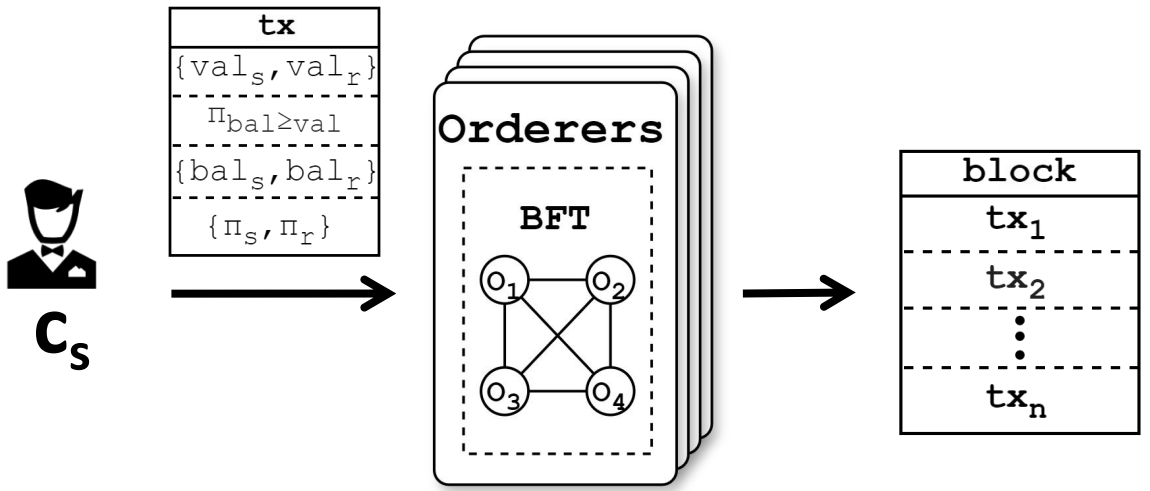
Algorithm 1: Transaction execution workflow of the client.

input: Plaintext transaction input data val
// Phase 1: Construction
1 $pk_s \leftarrow$ GetEncryptionKey(id_s); $val_s \leftarrow$ Encrypt(val,pk_s);
2 $pk_r \leftarrow$ GetEncryptionKey(id_r); $val_r \leftarrow$ Encrypt(val,pk_r);
3 $bal_s \leftarrow$ GetBalance(id_s);
4 $\pi_{bal \geq val} \leftarrow$ GenerateNIZKP($bal_s \geq val_s$, sk_s);
5 $tx \leftarrow \{val_s, val_r, \pi_{bal \geq val}\}$;
// Phase 2: Execution
6 results $\leftarrow \varnothing$;
7 **For every executor e; do in parallel**
8 $bal'_s, bal'_r \leftarrow$ SendForExecution(e,tx);
9 results $\leftarrow$ results $\cup \{bal'_s, bal'_r\}$;
10 $\pi_s \leftarrow$ GenerateNIZKP(a majority of bal'_s in results are consistent);
11 $\pi_r \leftarrow$ GenerateNIZKP(a majority of bal'_r in results are consistent);
// Phase 3: Ordering & Phase 4: Validation
12 $tx \leftarrow tx \cup \{bal'_s, bal'_r, \pi_s, \pi_r\}$;
13 SendForOrderingAndValidation(tx);

Workflow Phase 3 and 4: Ordering and Validation

Phase 3: Ordering.

1. After NIZKPs generation, the client sends the transaction to orderers for ordering.
2. Orderers run a BFT consensus protocol (e.g., PBFT) to collectively determine the transaction order within each block.
3. Once a consensus is reached among all orderers, they proceed to generate the block and disseminate the block to all executors for validation

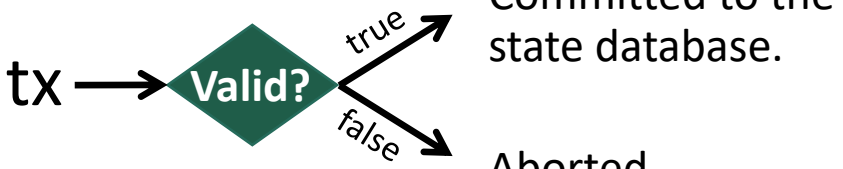


Phase 4: Validation.

The executor sequentially validates all transactions and only commit transactions that satisfy two conditions:

1. First, the transaction must not have any write conflict with previously committed transactions within the same block.
2. 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.

Once the executor has validated all transactions, the executor permanently appends the block to the local copy of the blockchain.



Implementation and Evaluation Metrics

Implementation

- We built a prototype system of Gafe based on Hyperledger Fabric v2.5 and simulated the business logic of digital payment.
- We implemented the CKKS scheme for GAFE based on the state-of-the-art studies on GPU-accelerated FHE.
- GAFE adopted the gnark [4] library's implementation for the Groth16 NIZKP system and employed our Golang implementation of the PBFT 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.

Evaluation Metrics

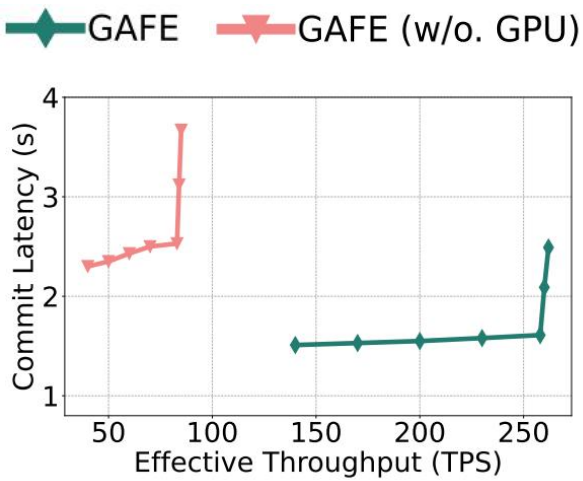
We evaluated the following metrics:

1. **Effective throughput**, which indicates the average number of transactions per second (TPS) committed to the blockchain;
2. **Commit latency**, which measures the time duration from transaction construction to commitment;
3. **The cumulative distribution function (CDF) of commit latency** for all committed transactions and **the latency of each phase** in the transaction execution workflow.

End-To-End Evaluations

Evaluation Methodology

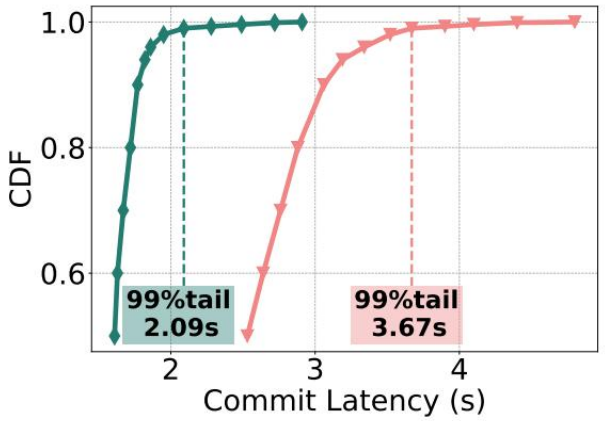
- We evaluated the end-to-end performance of GAFE and GAFE (w/o. GPU).
- For each evaluation, we created three executors, four orderers, and one thousand clients.
- We constructed and submitted 100,000 digital payment transactions.
- To prevent transaction aborts caused by write conflicts, we explicitly 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



End-To-End Performance

GAFE exhibited exceptional end-to-end performance:

- A **high throughput** of 258 TPS (**3.1×** increase compared to GAFE (w/o. GPU))
- A **low average latency** of 1.61 seconds (**37%** reduction compared to GAFE (w/o. GPU));
- A **shorter 99% tail latency** of 2.09 seconds (**43%** reduction compared to GAFE (w/o. GPU));

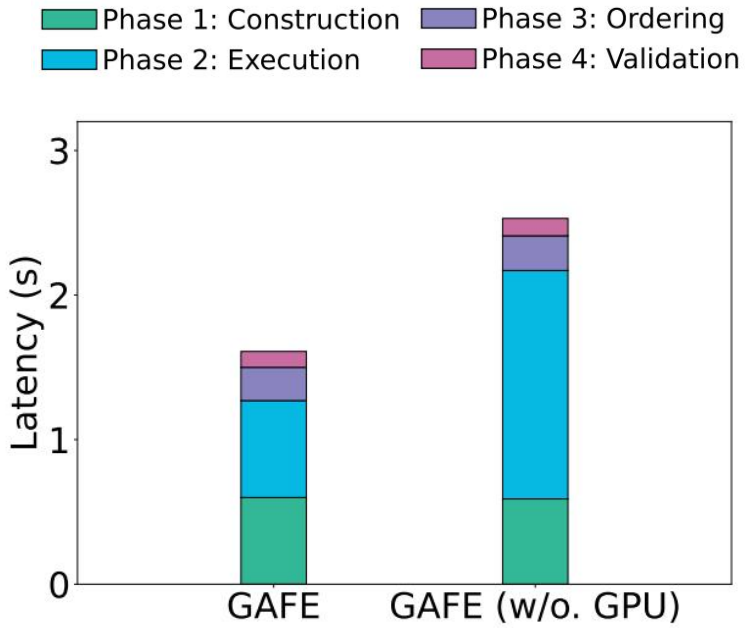


End-To-End Evaluations

Performance Analysis

GAFE's high performance is attributed to the concurrent FHE computations on GPU.

- 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.



Latency of each phase.

Conclusion

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.
- GAFE **supports general-purpose smart contracts** through the employment of FHE.
- 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).

Thanks for listening
and questions are welcome!

Speaker: Rongxin Guan