Decentralized FHE Computer

Gurgen Arakelov¹, Sergey Gomenyuk¹ and Hovsep Papoyan¹

Fair Math,

{gurgen,sergey,hovsep}@fairmath.xyz

Abstract.

The concept of a decentralized computer is a powerful and transformative idea that has proven its significance in enabling trustless, distributed computations. However, its application has been severely constrained by an inability to handle private data due to the inherent transparency of blockchain systems. This limitation restricts the scope of use cases, particularly in domains where confidentiality is critical.

In this work, we introduce a model for a Fully Homomorphic Encryption (FHE) decentralized computer. Our approach leverages recent advancements in FHE technology to enable secure computations on encrypted data while preserving privacy. By integrating this model into the decentralized ecosystem, we address the long-standing challenge of privacy in public blockchain environments. The proposed FHE computer supports a wide range of use cases, is scalable, and offers a robust framework for incentivizing developer contributions.

Keywords: fully homomorphic encryption, privacy-enhancing technologies, challenges, cryptography, privacy, fhe-computer, co-processor

Contents

[C CIFAR10 compiled code for FHE Computer](#page-37-0) 38

1 Introduction

Decentralized technologies have transformed the landscape of computing, enabling trustless systems, distributed ownership, and programmable value transfer. Web3, as a decentralized internet paradigm, is at the forefront of this transformation, powering applications in decentralized finance (DeFi), non-fungible tokens (NFTs), and governance. However, the full potential of decentralized systems is constrained by a critical challenge: privacy. Public blockchains like Ethereum [\[But15\]](#page-28-0) operate transparently, where all transaction data is visible to anyone. While this transparency ensures trust and verifiability, it simultaneously limits the applicability of these systems for use cases requiring sensitive or confidential data.

Privacy is essential for expanding decentralized technologies into domains such as healthcare, finance, enterprise operations and others, where sensitive data must remain confidential. For instance, securely sharing medical records, conducting private financial transactions, or managing proprietary business data is infeasible on a fully transparent blockchain [\[GMGT16,](#page-29-2) [Gro16\]](#page-29-3). Without robust privacy-preserving mechanisms, the adoption of Web3 technologies in these areas remains limited.

1.1 Related Works

To address the challenge of privacy in decentralized systems, researchers have developed various protocols based on Zero-Knowledge Proofs (ZKPs), Secure Multi-Party Computation (SMPC), and Trusted Execution Environments (TEEs). These protocols leverage the strengths of cryptographic technologies to enable privacy-preserving computations in decentralized environments.

Zero-Knowledge Proofs (ZKPs) allow the validation of statements without revealing the underlying data. Protocols such as zk-SNARKs [\[Gro16\]](#page-29-3) and zk-STARKs [\[BSBHR18\]](#page-28-1) have been successfully implemented in privacy-preserving systems like Zcash [\[GMGT16\]](#page-29-2), enabling secure transactions while maintaining confidentiality. These advancements have facilitated efficient proof systems, yet ZKPs are inherently limited to specific computations and lack support for general-purpose data processing.

Protocols based on Secure Multi-Party Computation (SMPC) enable multiple parties to collaboratively compute a function without exposing their individual inputs [\[Yao82\]](#page-29-4). Frameworks like SPDZ [\[SV13\]](#page-29-5) and ABY [\[DSZ15\]](#page-28-2) have extended SMPC's applicability to practical scenarios, providing robust mechanisms for privacy-preserving computation. However, despite significant progress, SMPC systems face scalability and latency challenges, especially in high-throughput and real-time environments.

Trusted Execution Environments (TEEs) provide a hardware-based mechanism for maintaining privacy. TEEs, such as Intel SGX [\[CD16\]](#page-28-3), establish secure enclaves for executing sensitive computations in a tamper-proof environment. Protocols like Ekiden [\[CSPS18\]](#page-28-4) have demonstrated how TEEs can be integrated into blockchain-based systems to protect sensitive data. However, reliance on centralized hardware providers introduces concerns about trust, availability, and the susceptibility to hardware vulnerabilities [\[KP20\]](#page-29-6).

Among other approaches, solutions based on Fully Homomorphic Encryption (FHE) are particularly promising. Unlike ZKPs, SMPC, or TEEs, FHE enables arbitrary computations to be performed directly on encrypted data, preserving confidentiality throughout the entire computation process [\[Gen09\]](#page-28-5). Recent advancements in FHE [\[AKP24\]](#page-27-1) [\[BKSS24\]](#page-28-6) [\[DDD](#page-28-7)⁺24b] [\[LLW24\]](#page-29-7) demonstrate its growing practicality for real-world applications. By supporting general-purpose encrypted computation, FHE overcomes the limitations of other privacy-preserving methods and holds the potential to transform decentralized systems by enabling a wide range of new use cases.

Despite significant progress in the development of Fully Homomorphic Encryption technologies, several critical challenges remain to be addressed [\[AKPP24\]](#page-27-2). These challenges include:

- **Lack of FHE Components:** The current ecosystem lacks a comprehensive library of modular, reusable FHE components tailored for diverse application needs. This gap limits the rapid development of complex FHE-based solutions.
- **High Computational Requirements:** FHE operations, while enabling computations on encrypted data, incur significant computational overhead compared to traditional processing. This constraint hinders scalability and widespread adoption for resource-intensive applications.
- **FHE Standardization:** The absence of standardized protocols, formats, and APIs across FHE implementations complicates interoperability and the integration of FHE into existing systems. Standardization efforts are crucial for unifying fragmented approaches and fostering collaboration across the field.

Addressing these challenges will be pivotal in advancing FHE as a cornerstone technology for privacy-preserving computation. Solving these issues will bring us closer to realizing robust solutions for privacy in decentralized and data-sensitive domains.

In this work, we introduce a decentralized FHE computer designed to address privacy limitations in both Web2 and Web3 spaces. Acting as a co-processor for Layer 1 and Layer 2 blockchains, the FHE computer enables privacy-preserving computations that complement the transparent nature of public blockchains.

With the FHE Computer and its surrounding ecosystem, we believe it will become possible to unlock new use cases for Ethereum and the broader Web3 space, as well as for Web2 applications.

1.2 Organization

In Section [2,](#page-4-1) we provide an overview of Fully Homomorphic Encryption and Blockchain technologies, offering the necessary background to understand the concepts discussed in later sections. Section [3](#page-7-0) introduces the high-level architecture of the FHE Computer, detailing its main components and their interactions within the system.

In Section [4,](#page-10-0) we describe the Instruction Set Architecture (ISA) model used by the FHE Computer and explain the motivations behind adopting this design approach. Section [5](#page-12-0) delves into the operating system that manages the FHE Computer, discussing its core functionalities and role in orchestrating computations.

Section [6](#page-24-0) focuses on the ecosystem surrounding the FHE Computer, examining its key components and their importance in creating a robust and scalable environment. Finally, in Section [7,](#page-25-1) we explore several potential use cases for the FHE Computer, demonstrating its applicability in both Web2 and Web3 contexts.

2 Preliminaries

2.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) is a groundbreaking cryptographic technique that allows arbitrary computations to be performed on encrypted data without requiring decryption. This capability ensures that sensitive data remains secure throughout computation, addressing critical privacy concerns in data processing.

Formally, an FHE scheme is defined over a plaintext space P , a ciphertext space C , and a key space K . It consists of the following components:

• Key Generation: $KeyGen(\lambda) \rightarrow (pk, sk)$, where λ is the security parameter, *pk* is the public key, and *sk* is the secret key.

- **Encryption:** Enc $(pk, m) \rightarrow c$, where $m \in \mathcal{P}$ is the plaintext and $c \in \mathcal{C}$ is the ciphertext.
- **Decryption:** Dec(sk, c) \rightarrow *m*, where $c \in C$ is the ciphertext and $m \in \mathcal{P}$ is the decrypted plaintext.
- **Evaluation:** Eval $(pk, f, c_1, \ldots, c_n) \rightarrow c'$, where f is a computable function, $c_1, \ldots, c_n \in$ C are input ciphertexts, and $c' \in C$ is the ciphertext of the result $f(m_1, \ldots, m_n)$ with $m_i = \text{Dec}(sk, c_i).$

The Turing completeness of FHE implies the ability to execute any computable function on encrypted data. However, achieving this capability is constrained by the underlying encryption scheme and computational efficiency.

Modern FHE schemes are broadly classified based on the data types they support:

- **Modular Arithmetic over Finite Fields:** Schemes like Brakerski-Gentry-Vaikuntanathan (BGV) [\[BGV11\]](#page-27-3) and Brakerski/Fan-Vercauteren (BFV) [\[FV12,](#page-28-8) [Bra12\]](#page-28-9) support computations on vectors of integers modulo a prime or a prime power. These schemes are ideal for applications involving exact integer arithmetic and small-integer computations.
- **Boolean Circuits and Decision Diagrams:** Schemes such as Ducas-Micciancio (DM) [\[DM14\]](#page-28-10) and Chillotti-Gama-Georgieva-Izabachene (CGGI) [\[CGGI16\]](#page-28-11) are optimized for logical operations and binary decision-making. These schemes offer fast bootstrapping but limited SIMD (Single Instruction, Multiple Data) packing capabilities.
- **Approximate Arithmetic over Real and Complex Numbers:** The Cheon-Kim-Kim-Song (CKKS) scheme [\[CKKS16\]](#page-28-12) enables computations on real and complex numbers with approximate precision. This class is particularly well-suited for machine learning, signal processing, and applications involving continuous data.

Most FHE schemes rely on the hardness of the Learning With Errors (LWE) [\[Reg09\]](#page-29-8) problem or its Ring variant (RLWE) [\[LPR09\]](#page-29-9) for security. Noise is added during encryption to ensure security, but it grows with each operation. When the noise level becomes too high, further computations become infeasible. A process called **bootstrapping** resets the noise, enabling continued computation. While bootstrapping is computationally expensive, advances in its efficiency have significantly expanded the practical applicability of FHE.

FHE schemes can also be categorized into two major types:

- **Exact Arithmetic:** Schemes like BGV and BFV perform computations with exact results.
- **Approximate Arithmetic:** CKKS trades exactness for broader applicability by allowing approximate computations.

FHE's Turing completeness signifies its ability to perform any computable function on encrypted data. However, this theoretical capability does not inherently provide practical algorithms, making the development process complex and challenging. Performance limitations often restrict feasible implementations to algorithms (or circuits) with a multiplicative depth below a predefined threshold *L*. These constraints significantly complicate application development, imposing limitations on developers and requiring innovative approaches to overcome these challenges.

2.2 Blockchain and Ethereum

Blockchain technology serves as the foundation for decentralized systems, providing a secure, immutable, and distributed ledger. A blockchain is a sequence of blocks, each containing a cryptographically linked list of transactions. This linkage ensures data integrity and prevents tampering by leveraging hash functions and consensus mechanisms [\[Nak08\]](#page-29-10).

Ethereum [\[But15\]](#page-28-0) extends the traditional blockchain architecture by introducing programmability through a Turing-complete virtual machine known as the Ethereum Virtual Machine (EVM). Ethereum supports the deployment and execution of *smart contracts*, which are self-contained, executable pieces of code stored on the blockchain. These contracts enable decentralized applications (dApps), providing a framework for use cases such as tokenization, decentralized finance (DeFi), and automated governance.

State Machine Model Ethereum operates as a **state machine**, where the global state *S* represents the current status of all accounts and contracts on the network. The state transition function γ modifies the state based on incoming transactions *T*, such that:

$$
\gamma(S, T) \to S'
$$

where S is the current state, T is a transaction, and S' is the new state after applying T . The state *S* includes:

- Account balances.
- Contract storage and code.
- Nonces and other metadata.
- A transaction *T* is formally represented as:

 $T = (t_{\text{sender}}, t_{\text{recipient}}, t_{\text{value}}, t_{\text{data}}, t_{\text{gas}}, t_{\text{signature}})$

where t_{sender} is the sender's address, $t_{\text{recipient}}$ is the recipient's address (or contract being invoked), t_{value} is the amount of ether transferred, t_{data} contains any input data, t_{gas} specifies computational resources, and $t_{\text{signature}}$ authenticates the transaction.

Consensus and Execution Ethereum's execution of transactions relies on consensus mechanisms to ensure consistency and validity. Initially, Ethereum employed Proof of Work (PoW) [\[Woo14\]](#page-29-11), which required solving computationally intensive puzzles to validate blocks. With the Ethereum 2.0 upgrade, the network transitioned to Proof of Stake (PoS) [B ⁺[21\]](#page-27-4), where validators stake cryptocurrency to propose and attest to blocks, significantly reducing energy consumption and improving scalability.

Smart contract execution is deterministic, with all nodes independently executing the same transactions to achieve state convergence. Gas fees are introduced as a mechanism to incentivize validators and mitigate denial-of-service attacks by associating a cost with computational operations.

Privacy and Challenges While Ethereum enables programmability and decentralization, it operates transparently, with all transactions and states visible to the public. This transparency introduces privacy concerns, limiting the adoption of blockchain technology in sensitive domains such as healthcare, enterprise applications, and confidential financial transactions [\[GMGT16,](#page-29-2) [Gro16\]](#page-29-3).

Efforts to enhance privacy in Ethereum include:

- **zkEVM**: Layer 2 solutions such as zk-Rollups $[W^+18]$ $[W^+18]$ and zkEVM $[P^+22]$ $[P^+22]$ have been proposed to enable privacy-preserving computations compatible with Ethereum's infrastructure.
- **FHE Integration**: Research on integrating Fully Homomorphic Encryption into the Ethereum Virtual Machine is underway $[DDD+24a]$ $[DDD+24a]$, aiming to perform encrypted computations while preserving privacy of user input .

Despite progress, challenges remain, including the computational overhead of cryptographic operations and the lack of standardized solutions for privacy-preserving computation on Ethereum. These challenges highlight the need for continued innovation to unlock Ethereum's full potential in privacy-sensitive applications.

3 System Architecture

3.1 Overview

Fair Math Computer is a decentralized execution environment with heterogeneous execution nodes (actors) designed to perform computations on encrypted data. Its architecture as depicted on Figure [1](#page-8-0) is structured into five primary layers:

- Application Layer
- Orchestration Layer
- Verification Layer
- Execution Layer
- Data Layer.

The first three layers form the Decentralized Operating System based on blockchain. The Execution Layer serves as the hardware equivalent, comprising a scalable network of computational nodes, while the Data Layer, implemented as IPFS-based storage, functions as the system's permanent repository for encrypted and plaintext data.

Our Operating System incorporates core concepts such as applications, processes, threads, and virtual memory, drawing inspiration from UNIX-like systems. This system provides the foundation for application management, resource allocation, and task scheduling while maintaining the security and decentralization. We consider each layer in more details in the following subsection.

3.2 Architectural Layers

3.2.1 Application Layer

The Application Layer is the highest level in our multi-layered architecture. This layer acts as the primary interface for the interaction with the system.

Each application stored at the Application Layer is associated with a security policy that governs its execution. These policies define the conditions under which applications can be accessed and executed. For example, some applications may be publicly available to all users, while others may restrict access to a predefined group of authorized users.

3.2.2 Orchestration Layer

The Orchestration Layer is a responsible for managing the execution workflow of applications deployed on the system. It coordinates task distribution among actors, creates execution contexts, and ensures that applications are efficiently and securely executed. The orchestration process involves several key phases:

Figure 1: FHE Computer Layers

Application Execution Graph In the first phase of execution, an application is analyzed and transformed into an *execution graph*. Each application may define multiple *entry points* (functions) as its potential starting points. During the application launch, the specific entry point is selected and execution begins from that point.

The execution graph represents the logical flow of the application. The nodes of this graph correspond to *blocks of instructions (tasks)*, which are the fundamental units of execution. Each block is designed to be assigned to actors for processing. A block contains one or more instructions, which are categorized into two types:

- **Atomic Instructions:** These are indivisible operations within the system's Instruction Set Architecture (ISA) that can be executed directly by actors without further decomposition.
- **Composite Instructions:** These represent high-level operations that may be expanded into a sequence of atomic instructions during the execution planning phase. For instance, a composite instruction like a polynomial evaluation or matrix multiplication might be unrolled into smaller, atomic operations to optimize execution.

The orchestration layer is responsible for constructing the execution graph. It determines dependencies between the blocks and ensures that the graph adheres to the logical flow of the application. The edges in the graph define the dependencies between blocks, ensuring that the execution proceeds in a consistent and orderly manner. This structure allows the orchestration layer to distribute execution efficiently across available actors, optimizing parallelism and resource utilization. Dependencies between blocks are also captured on Orchestration Layer, enabling efficient scheduling and execution. Further details are provided in Section [5.3.](#page-15-0)

Process Once the execution graph is constructed, the system creates a *process* for the application, similar to traditional Operating systems. A process is an isolated execution entity that encapsulates all orders (blocks of instructions), data and the associated *execution context*. This approach ensures isolation between different application runs, providing security and stability for the decentralized environment.

The process serves as a container for managing the application's lifecycle, including task execution, resource management, and context handling. It maintains the state of the execution and acts as a boundary for resource allocation and data flow.

Task Assignment and Actor Selection. The assignment of tasks to specific actors is managed through a *decentralized order book*. This mechanism allows actors to compete for tasks based on various factors, such as their capabilities, availability, performance history and cost of the computation. The decentralized order book ensures fairness and transparency in task management.

Execution and Result Aggregation. Once an actor is assigned an order, it retrieves the associated execution context and begins processing the instructions within the block. Key aspects of this phase include:

- Actors execute instructions based on the provided context, handling both atomic and complex operations.
- Dependent orders in the execution graph are activated as their prerequisites are completed, enabling parallelism and efficient resource utilization.
- The Orchestration Layer monitors task progress and ensures that results are correctly propagated through the graph.

3.2.3 Verification Layer

In the security model we consider, both computational nodes (actors) and users can potentially act maliciously. For instance, an actor might intentionally fail to execute the assigned instructions and instead return a random ciphertext, falsely claiming it as the computation result. Conversely, a user—after receiving the correct result—may unjustifiably claim the output is invalid to avoid compensating the actor for their work. In extreme cases, both parties may act as active adversaries, requiring robust mechanisms to address such scenarios.

The problem is further complicated by the encrypted nature of the computational results. Since the outputs remain encrypted, there is no direct way to verify the correctness of the result against the expected output without revealing sensitive data. This limitation necessitates the development of verification mechanisms that preserve privacy while ensuring reliability.

- **Static Circuit Requirements:** Modern zk-proof systems, such as zk-SNARKs [\[Gro16\]](#page-29-3) and zk-STARKs [\[BSBHR18\]](#page-28-1), rely on predefined circuits that explicitly describe the computation to be verified. This static nature requires that for each task assigned to an actor, an appropriate proof circuit must be constructed and available in advance. This constraint reduces flexibility and adaptability, particularly in dynamic environments where computational tasks may evolve unpredictably.
- **Lack of Flexibility in Execution Methods:** zk-based verification inherently requires the algorithm being verified to be entirely disclosed and structured in advance, as highlighted in $[P^+22]$ $[P^+22]$. This limitation conflicts with execution models that encourage actors to use their own innovative and potentially proprietary methods to execute instructions, provided the input-output behavior aligns with specifications. For instance, an actor performing matrix multiplication could optimize the operation with a novel algorithm; however, zk-proof systems would require this algorithm to be explicitly included in the verification circuit, deterring proprietary or innovative approaches.

In addition to zk-based approaches, there exists an alternative methodology known as Verifiable FHE (vFHE) [\[VKH23\]](#page-29-14). This technique leverages properties of the computation's output to verify correctness without compromising data confidentiality. While vFHE offers promising possibilities, it is still a developing field and currently lacks the universality and practicality needed for broad adoption. Given the absence of a universal protocol suitable for all scenarios, our model emphasizes flexibility. Different verification mechanisms must be supported to accommodate the diverse requirements of various applications. At this layer, the system is designed to enable the integration of multiple verification approaches, allowing developers to tailor solutions to their specific use cases while maintaining the integrity, security, and flexibility that define the Fair Math Computer.

3.2.4 Execution Layer

The Execution Layer is composed of a heterogeneous network of computational nodes, referred to as actors, which are capable of executing instructions defined by the virtual machine. Each actor may support a subset of the available instructions. For example, some actors may specialize in instructions related to a specific encryption scheme, while others may support a single operation, such as ciphertext multiplication.

When registering as an actor, the node specifies the list of instructions it can execute, along with metadata describing its performance characteristics. This information is crucial for task allocation and ensures that actors are matched to tasks that align with their capabilities. Actors also have the flexibility to modify their supported instruction set at any time, enabling them to adapt to changing workloads or optimize for specific tasks.

As described in Section [5.5,](#page-17-1) during the assignment of an actor to a task(block of instructions), the system considers the actor's declared capabilities. It ensures that actors are only assigned blocks consisting of instructions they explicitly support. This guarantees efficient and reliable execution, even in a heterogeneous environment with actors of varying specializations.

We discuss actors in more details in section [5.8](#page-22-0)

3.2.5 Data Layer

Given the strict limitations and high costs of data storage in blockchain networks like Ethereum, and considering that the volume of ciphertexts and keys for Fully Homomorphic Encryption can reach several gigabytes, we have adopted a strategy to avoid placing such large amounts of data directly on the blockchain. Instead, we utilize IPFS-based external data storage to manage the data. This approach not only addresses the storage limitations and cost inefficiencies but also ensures efficient, scalable, and decentralized data management. On the on-chain side, instead of storing the ciphertexts and keys themselves, we only place commitments and meta-data corresponding to the files located on the external storage. The blockchain serves as a form of a hash table, pointing to the location of the data and ensuring their integrity and authenticity through a system of data commitments. This allows us to optimize the costs of using the blockchain while providing a reliable and scalable infrastructure for working with encrypted data of arbitrary scale. This data storage model makes our platform ideally suited for a wide range of applications requiring high data confidentiality and security.

4 Instruction Set Architecture (ISA)

In the Fair Math FHE Computer, the **Execution Layer** functions as a multiprocessor system, where the computational cores are represented by heterogeneous nodes, referred to as actors. These actors are heterogeneous because they may vary in two key aspects: the subset of instructions they support and the specific implementations they use for those

instructions. This design provides the flexibility to accommodate a wide range of actors with diverse computational capabilities, enabling scalability and specialization.

The **Instruction Set Architecture (ISA)** of the FHE Computer defines a canonical set of instructions supported by Execution Layer. The instructions are the part of an executable program, stored on Application layer and executed on Execution Layer. Actors in the FHE Computer can implement arbitrary subset of instructions. This approach allows specialization; for instance, some actors may focus exclusively on specific operations, such as encrypted arithmetic for a particular FHE scheme, while others may implement more general-purpose functionality. Importantly, the system guarantees that, for every instruction in the canonical instruction set, there is at least one actor capable of executing it at any given time.

To support this heterogeneous and extensible model, the ISA adopts a declarative approach. Inspired by the **MLIR framework**, instructions are organized into logical groups called **dialects**, which categorize operations based on their purpose and domain. Each instruction specifies its required inputs and expected outputs but does not prescribe the exact implementation. This abstraction allows actors to optimize execution based on their capabilities while adhering to a standardized interface.

Hierarchy of Operations

The instruction set is modularly organized into dialects, ensuring clarity, extensibility, and specialization. Each dialect groups a set of related instructions, facilitating straightforward management and seamless extension. Dialects are hierarchically structured, allowing sub-dialects to be nested within parent dialects. This approach supports fine-grained categorization and flexibility in defining operations across various computational domains.

For instance, the top level fhe dialect includes sub-dialects such as fhe.bgv and fhe.ckks, which define instructions specific to the BGV and CKKS encryption schemes, respectively. At the meantime the fhe dialect defines unified data types, such as RLWECiphertext, RLWEPlaintext that shared across sub-dialects. While different encryption schemes may utilize different encoding techniques, we would like to provide some universal layer for FHE schemes based on the same math problem, like (R)LWE. This design usefull for adding support for new scheme, scheme switching, and DM based bootstrapping approaches.

In the initial version of the FHE Computer, the following four top-level dialects are defined:

1. Arithmetic Dialect (arith) The arith dialect provides foundational operations for working with basic types. It includes instructions for basic arithmetic, logical operations, and comparisons. This dialect is flat, with no subgroups, and focuses entirely on nonencrypted types.

• **Examples:**

- **–** %sum = arith.add %a, %b : (i32) -> i32 Adds two integers and returns an integer result.
- **–** %eq = arith.eq %x, %y : (i64) -> bool Compares two 64-bit integers for equality, returning a Boolean result.

2. Tensor Dialect (tensor) The tensor dialect supports operations on multidimensional arrays, commonly used in machine learning and linear algebra computations. While the initial implementation is flat, future versions may include subgroups for specialized tensor operations (e.g., sparse tensors).

- **Examples:**
	- **–** %tensor = tensor.create %shape : (i32) -> tensor<[n]> Creates a 1 dimensional tensor of size n.
	- **–** %result = tensor.add %t1, %t2 : (tensor<[4]x[4]>, tensor<[4]x[4]>) -> tensor<[4]x[4]> Performs element-wise addition on two 4x4 tensors.

3. FHE Dialect (fhe) The fhe dialect includes operations for encrypted data computations. This dialect is further divided into subgroups based on the encryption scheme, such as fhe.bgv and fhe.ckks, allowing tailored operations for each scheme. Within each subgroup, operations like addition, multiplication, and rotations are defined.

- **Examples:**
	- **–** %result = fhe.bgv.add %lhs, %rhs, %ctx : (RLWECiphertext, RLWECiphertext, BGVCryptoContext) -> RLWECiphertext Adds two ciphertexts using the BGV scheme.
	- **–** %rot = fhe.ckks.rotate %ciphertext, %offset, %ctx, %key : (RLWECiphertext, i32, CKKSCryptoContext, RLWEGaloisKey) -> RLWECiphertext Rotates coefficients of a CKKS ciphertext by the specified offset.

4. Polycircuit Dialect (polycircuit) The polycircuit dialect provides high-level operations mapped to reusable FHE components, enabling complex computations such as neural network activations and polynomial evaluations. These operations often work across multiple ciphertexts and contexts.

- **Examples:**
	- **–** %relu_out = polycircuit.RELU %ciphertext, %ctx : (RLWECiphertext, Context) -> RLWECiphertext Applies the ReLU activation function to an encrypted input.
	- **–** %eval = polycircuit.poly_eval %coeffs, %ciphertext, %ctx : (tensor<[n]>, RLWECiphertext, Context) -> RLWECiphertext Evaluates a polynomial defined by the coefficients on an encrypted input.

We provide the complete list of instructions in Appendix [B](#page-30-2) and compiled code example for CIFAR10 recognitiona app in Appendix [C.](#page-37-0)

5 Operating System Design

The operating system of the Fair Math FHE Computer is built on a blockchain infrastructure, providing a decentralized framework for resource management and task allocation. It functions as a distributed scheduler, orchestrating workloads across heterogeneous actors based on their computational capabilities and supported instruction subsets. Leveraging blockchain technology, the system ensures deterministic resource allocation, verifiable execution, and secure state transitions. This approach enables transparent interaction between actors, immutability of task assignments, and seamless integration of new computational nodes into the network.

5.1 Computer State

The computer can be conceptualized as a **state machine**, with its state evolving deterministically based on transactions recorded on the chain. At any given moment, the computer resides in a well-defined state, characterized by multiple parameters and factors such as:

- **Resource Availability:** The allocation and utilization of computational, memory, and network resources across the decentralized network of actors.
- **Task Queue:** The current set of tasks awaiting execution, their priority levels, and the mapping of tasks to specific actors.
- **Actor States:** The operational status of each actor, including supported instructions, current workloads, and performance metrics.
- **Deployed Applications:** Metadata and configurations for applications running on the computer, including active jobs, required cryptographic keys, and runtime dependencies.
- **Transaction History:** Immutable records of task submissions, resource allocations, and interactions between system components.

The blockchain acts as the single source of truth, ensuring that the state of the computer evolves in a transparent and tamper-proof manner. Transactions on the chain serve as the triggers for state transitions, which may include deploying new applications, assigning tasks to actors, or reallocating resources.

5.2 Application

An **application** in the Fair Math Computer is formally defined as a tuple:

$$
\mathcal{A}=(\mathcal{F},\mathcal{V}),
$$

where:

- F is a set of **external functions** $f : (\mathcal{X} \to \mathcal{Y})$, where X and Y are the input and output spaces, respectively. Each function *f* serves as an **entry point** and encapsulates a specific sequence of instructions. Importantly, functions in $\mathcal F$ do not support direct invocation of other functions within the same application.
- V is a set of global variables $v \in V$, each defined as $v : \mathcal{T}$, where \mathcal{T} is the domain of the variable's type (e.g., integers, arrays, ciphertexts).

Execution of an application starts by default with the main function. The main function serves as the default entry point and is executed unless a different function is explicitly specified during invocation.

When an application is launched, a **process** is created. This process persists until the application explicitly calls the exit function (or killed by Operating System), providing a termination code. The process manages the application's state, resources, and execution flow throughout its lifecycle. Details about processes are discussed in Section [5.4.](#page-17-0) Application deploy process is depicted on the Figure [2](#page-14-0)

Figure 2: Application Deploy Process

5.2.1 Application Invocation

Applications are invoked through **transactions**, which can perform two types of operations:

- 1. **Launching a New Application:** A transaction initiates a new application by specifying its identifier A_{id} and any required input arguments. This creates a new process P , associated with the application, that starts executing from the specified entry point or main function by default.
- 2. **Invoking a Specific Function:** A transaction can directly invoke an external function $f_{name} \in \mathcal{F}$ in a running application by specifying:
	- The **application ID** A_{id} , uniquely identifying the target application.
	- The **function name** f_{name} , indicating the entry point to invoke.
	- The associated **process ID** \mathcal{P}_{id} , linking the invocation to the relevant application process.

Input arguments are passed dynamically, allowing flexible interaction with the running application.

5.2.2 External Functions

Functions in $\mathcal F$ are strictly **external**, meaning they can only be invoked through transactions from outside the application or by the orchestration layer. Direct function calls from one function to another within the same application are not supported. This design ensures:

- Clear separation of entry points, simplifying execution flow.
- Modular interaction with the application, as each function operates independently.
- Improved transparency and security by explicitly restricting internal function calls.

5.2.3 Interactive Applications

The application model inherently supports the development of **interactive applications**. By defining multiple external entry points and supporting dynamic invocation of functions, developers can implement applications that:

- Trigger new computations in response to external events.
- Maintain shared state across invocations through global variables $\mathcal V$.
- Respond to user input or other asynchronous triggers, enabling dynamic and stateful workflows.

5.3 Execution Graph and Tasks

In the Fair Math Computer, a **task** is the smallest unit of execution that is managed by the orchestration layer. Formally, a task is defined as:

$$
T = \langle \mathcal{I}, \mathcal{V}, \mathcal{M} \rangle,
$$

where:

- $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ is a non-empty, ordered set of instructions associated with the task.
- $V = \{v_1, v_2, \ldots, v_p\}$ is the set of variables used or produced by the instructions in \mathcal{I} .
- M is a set of metadata associated with the task, which includes:
	- **–** Resource requirements (e.g., memory, compute capacity).
	- **–** Estimated complexity.
	- **–** Execution constraints (e.g., deadlines, actor capabilities).

A task is considered valid if $\mathcal{I} \neq \emptyset$ and all instructions $I_k \in \mathcal{I}$ are consistent with the dependencies of the application.

5.3.1 Atomic and Composite Instructions

Instructions in $\mathcal I$ are categorized as either **atomic** or **composite** operations:

• **Atomic Instruction:** An instruction is atomic if it represents a single, indivisible operation within the system's Instruction Set Architecture (ISA). For example:

arith.mul (multiplication of two ciphertexts)*.*

Atomic instructions are directly executable by actors without further decomposition.

• **Composite Instruction:** An instruction is composite if it represents a high-level operation that can be expressed as a sequence of atomic instructions. For example:

polycircuit.relu (non-linear activation function)*.*

A composite instruction like RELU might be implemented as a sequence of operations involving comparisons, multiplications, and additions.

During execution planning, the orchestration layer can dynamically unroll composite instructions into their constituent atomic instructions to optimize scheduling. Formally, let:

$$
I_{\text{composite}} \rightarrow \{I_{\text{atomic},1}, I_{\text{atomic},2}, \ldots, I_{\text{atomic},k}\},
$$

where *I*composite is a composite instruction, and *I*atomic*,i* are the atomic instructions that implement it. The orchestration layer may replace composite instructions in $\mathcal I$ with their expanded atomic sequences if:

Figure 3: Execution graph example

- No suitable actor is available to execute the composite instruction directly.
- A finer-grained decomposition enables better parallelism or resource utilization.

This process is referred to as **instruction unrolling** and allows the orchestration layer to dynamically adjust the granularity of tasks.

Dependencies between tasks are defined by the relationships between their variables. Formally, given two tasks $T_i = \langle \mathcal{I}_i, \mathcal{V}_i, \mathcal{M}_i \rangle$ and $T_j = \langle \mathcal{I}_j, \mathcal{V}_j, \mathcal{M}_j \rangle$, we say T_j depends on T_i , denoted $T_i \rightarrow T_j$, if:

- $j > i$ (i.e., T_i precedes T_j in the sequence of tasks), and
- T_j uses one or more variables whose values are modified by T_i .

Formally, let:

 $Out(T_i) = \{v \mid v \text{ is a variable modified by } T_i\},\$

 $In(T_i) = \{v \mid v \text{ is a variable used by } T_i\}.$

Then $(T_i, T_j) \in E$ if and only if:

$$
j > i \quad \text{and} \quad \text{Out}(T_i) \cap \text{In}(T_j) \neq \emptyset.
$$

5.3.2 Execution Graph

The decomposition of an function into tasks is represented as a directed acyclic graph (DAG) called the **execution graph**, $G = (V, E)$, where:

- $V = \{T_1, T_2, \ldots, T_m\}$ is the set of tasks.
- $E \subseteq V \times V$ is the set of directed edges, where $(T_i, T_j) \in E$ represents a dependency between tasks.

During execution, the orchestration layer may:

- Dynamically refine tasks into subtasks if finer granularity is required for scheduling.
- Expand composite instructions within tasks to their atomic equivalents, enabling execution by available actors.

This dynamic refinement ensures that the system can adapt to varying resource availability and workload requirements while respecting task dependencies. Example of the execution graph is depicted on Figure [3.](#page-16-0)

5.4 Process

In the Fair Math Computer, a **process** represents an instance of an application in execution. Processes are ephemeral entities that encapsulate the state, resources, and execution context required to perform computations. They serve as the primary abstraction for managing the execution lifecycle of applications, ensuring isolated, secure, and efficient operations in a decentralized environment.

5.4.1 Lifecycle of a Process

The lifecycle of a process in the Fair Math Computer includes the following key stages:

- 1. **Initialization:** A process is created when a user or actor invokes an application. During initialization, the process is assigned a unique identifier and its initial state is constructed based on the application's configuration file and input arguments.
- 2. **Execution:** The process executes the instructions defined in the application. Execution is distributed across the network, leveraging the computational resources of heterogeneous actors. The system ensures the correctness and integrity of execution through cryptographic guarantees.
- 3. **Suspension and Resume:** In cases where a process requires external input or encounters resource constraints, it can be suspended. The process state is serialized and stored securely on the blockchain, allowing it to be resumed later without loss of progress.
- 4. **Termination:** Once the execution completes, the process is terminated. The final state, including any results, is recorded on the blockchain, and all associated resources are released.

5.4.2 Resource Allocation and Isolation

Each process operates in a sandboxed environment, ensuring that it cannot interfere with other processes or access unauthorized resources. Resource allocation is managed by the blockchain-based operating system, which dynamically assigns actors and computational capacity to processes based on workload, priority, and availability.

The concept of processes in the Fair Math Computer ensures that applications execute reliably, securely, and efficiently, forming a robust foundation for decentralized computations.

5.5 Order Book

In the system, the fundamental unit of planning is a **task**. A task is defined as a set of one or more instructions from the Instruction Set Architecture (ISA). The allocation of tasks to actors is managed at the Orchestration Layer through the **order book**.

Each task that requires execution is transformed into an **order**. An order encapsulates the task along with its associated metadata, which specifies the requirements for execution. This metadata includes:

- **Deadline**: The maximum allowable time for task completion.
- **Maximum Reward**: The upper limit of compensation for completing the task.
- **Complexity**: A quantitative measure of the computational resources required for execution.

5.5.1 Order Matching Mechanism

Orders are matched with bids from actors through the order book, which operates on the principles of a classic order matching system. Actors submit offers to take on tasks based on their available resources and capabilities. The order book facilitates the following processes:

- **Task Assignment**: Matching tasks to actors that propose acceptable bids.
- **Execution Monitoring**: Ensuring tasks are executed within the specified parameters, including deadline and complexity constraints.
- **Incentive Structuring**: Rewarding actors for successful task completion and enforcing penalties for unmet deadlines.

This mechanism ensures a decentralized, efficient, and scalable distribution of workloads, while maintaining a transparent framework for actor participation and performance evaluation.

Each task in the system is associated with a **complexity metric**, which quantifies the computational resources required for its execution. The complexity of a task, denoted as C_{task} , is defined as the sum of the complexities of all instructions it contains:

$$
C_{\text{task}} = \sum_{i=1}^{n} C_{\text{instr},i},
$$

where *n* is the number of instructions in the task, and $C_{\text{instr},i}$ represents the complexity of the *i*-th instruction.

5.5.2 Instruction Complexity and Input-Dependent Parameters

The complexity of an instruction *C*instr is determined by the specific instruction type and its input arguments, as well as cryptographic parameters that affect its execution. For operations involving encrypted data, the same instruction can exhibit different complexities depending on the parameters of the input ciphertexts and the associated cryptographic context.

Formally, the complexity of an instruction can be expressed as a function:

$$
C_{\text{instr}} = f_{\text{instr}}(\text{inputs}, \text{crypto_params}),
$$

where:

- inputs refers to the arguments passed to the instruction, such as ciphertexts.
- crypto params represents parameters from the cryptographic context that define the operational environment.

Consider the instruction fhe.bgv.mult, which performs a multiplication of encrypted values. The input arguments for this instruction include:

- **Ciphertexts**: The encrypted operands, each characterized by properties such as:
	- **– Multiplicative Depth (***D***)**: The current depth of operations performed on the ciphertext. Larger depths increase computational complexity due to the need for relinearization and modulus switching.
- **CryptoContext Parameters**: Contextual cryptographic parameters, including:
	- **– Ring Dimension (***N***)**: The size of the polynomial ring, impacting the cost of polynomial arithmetic.
- **– Scaling Factor (**∆**)**: Defines the precision of computations.
- **– Security Level**: Specifies the cryptographic strength.

The complexity C_{instr} for bgv.mult can be expressed as:

 $C_{\text{instr}} = f_{\text{mult}}(N, \Delta, D, \ldots),$

where f_{mult} models the cost of underlying cryptographic operations, including polynomial multiplications, relinearizations, and modulus switching. Thus, for each instruction, a cost function is defined that determines the complexity of the specific instruction for the given input.

5.5.3 Role of Complexity Metrics in Task Execution

The task complexity metric plays a critical role in the system by:

- **Resource Estimation**: Allowing actors to evaluate the computational effort required for a given task.
- **Load Balancing**: Enabling efficient distribution of tasks among actors based on their capabilities.
- **Cost Modeling**: Providing a foundation for determining the rewards and penalties associated with task execution.

By accounting for both instruction-level complexity and input-dependent parameters, the system provides a detailed and accurate representation of the resources required for task execution, especially in cryptographic computations where performance is heavily influenced by ciphertext properties and cryptographic settings.

Each task in the order book is characterized by two critical parameters:

- 1. **Deadline** the maximum time within which the task must be completed.
- 2. **Maximum cost** the upper limit of compensation for completing the task.
- 3. **Complexity** a quantitative measure of the computational resources required to execute the task, based on the complexities of its constituent instructions.

Actors that take tasks from the order book are obligated to complete them within the specified deadlines. Failure to meet the deadline results in penalties, encouraging efficient resource allocation and timely task completion.

5.5.4 Task re-Delegation

Actors can re-delegate tasks or parts of tasks back to the order book. This introduces a dynamic and scalable approach to task execution.

For instance, an actor may take on a complex task consisting of several components. During execution, the actor can decide:

- To execute the components locally if it is efficient in terms of time and resources.
- To place some instructions back into the order book if delegation is deemed more cost-effective.

This capability transforms actors into local, partial orchestrators capable of dynamically adapting to changing conditions. If, during task execution, an actor determines that delegating specific instructions to other actors will result in faster or cheaper execution, it can initiate this process. The delegated instructions are returned to the order book, becoming available to other actors.

The detailed process of Application running is depicted on the Figure [4](#page-20-1)

Figure 4: App running

5.6 Context

The **Context** is a concept in our Computer model, representing the state of the process. It acts as a container for all relevant data, including variables, their current values, and additional parameters required for executing assigned tasks.

When an actor is assigned a task, the associated context is passed to it. This context includes the current state of all accessible variables and any necessary cryptographic information, such as keys or the cryptographic context. As the actor executes the instructions, it generates a new context that reflects the updated state. This new context is then submitted to the blockchain, where it updates the global state of the relevant process.

The context serves as the medium through which actors communicate the results of their computations, including intermediate results. Throughout the execution of the application, the context evolves, and a final context is formed and saved once the application completes. While the context itself is stored on the blockchain, it is designed to be lightweight. Large objects, such as arrays and ciphertexts, are stored on external data layers, with the context containing only references (e.g., hashes) to these objects. This design ensures efficiency and scalability.

5.6.1 Structure of the Context

The context is represented as a JSON-like object with the following sections:

- **fhe**: Holds cryptographic data such as:
	- **–** Public keys.
	- **–** Rotation keys.
	- **–** Cryptographic contexts.
- **args**: Represents the command-line-style arguments provided to the application.
- **vars**: Describes all variables used in the process. Each variable entry contains:
	- **–** id: The identifier of the variable.
	- **–** basetype: The base type of the variable (e.g., i32).
	- **–** is_secret: Boolean indicating if the variable is encrypted.
	- **–** is_array: Boolean indicating if the variable is an array.
	- **–** storage: Location where the variable's value is stored (e.g., local or ipfs).
	- **–** value: The current value of the variable, either directly or as a reference (e.g., an IPFS hash).

5.6.2 Example Context File

Below is an example of a context file:

```
{
    "vars": {
        "%arg0": {
            "basetype": "",
            "is_array": false,
            "is_secret": false,
            "storage": "ipfs",
            "value": "ipfs.QmXR5FDSupU6ZKxxkU95WcywhmAvykLZbNEqtcu2P3M43a"
        },
        "%arg2": {
            "basetype": "i32",
            "is_array": true,
            "is_secret": false,
            "storage": "ipfs",
            "value": "ipfs.QmfZhqPFDZmyK4rAxwDH5FTHeQGrJjDFbRJ1WDzW4qpF1P"
        },
        "%mul_key": {
            "basetype": "CKKSMulKey",
            "is_array": false,
            "is_secret": false,
            "storage": "ipfs",
            "value": "ipfs.QmT4xPQnAkZXR6PGDySaVZ4kBPbHdkVi2TqZgUZdREhfa7"
        },
        "pk": {
            "basetype": "CKKSPublicKey",
            "is array": false,
            "is_secret": false,
            "storage": "ipfs",
            "value": "ipfs.QmeA4jDooaoR4G2amHtnQKGpRD4eEdfVZSLjARDFF5HJq4"
        }
    }
}
```
5.7 Component Repository and Component based App

We utilize a **Component Repository** which is a dynamic library of FHE components that act as modular building blocks for applications. Each component is implemented as a function composed of ISA instructions and represents a high-level abstraction, such as ReLU, Sign, or polynomial evaluation.

These components are analogous to **intrinsics** in traditional computing systems and matched 1:1 to polycircuit instruction dialect. Just as intrinsics provide optimized, low-level access to hardware instructions while maintaining a high-level programming interface, FHE components encapsulate specialized cryptographic computations in a reusable, standardized format. This abstraction enables developers to use sophisticated functionality without dealing with the underlying complexity of the implementation.

Components in the repository dynamically form the **polycircuit** dialect within the ISA. The ISA includes instructions specifically designed for invoking components from this dialect. These instructions provide a seamless interface for integrating components into the computational pipeline, ensuring both flexibility and efficiency.

Components are composable, meaning they can use other components within their implementation. For example, a neural network component may internally invoke other components, such as activation functions (ReLU, Sigmoid) or pooling operations, making it easier to construct and maintain complex algorithms.

To encourage upgradability developers are incentivized to contribute to the repository. When a component is utilized in an application, its creator receives a proportional share of the rewards generated from the application's execution. This incentive mechanism fosters a vibrant ecosystem of high-quality, reusable components.

The repository is managed through the $FHERMA¹$ $FHERMA¹$ $FHERMA¹$ challenges and benchmarking platform. This platform validates, benchmarks, and version-controls submitted components, ensuring security, compatibility, and optimal performance.

5.8 Fair Math Actors

The execution layer of Fair Math relies on *actors*, entities responsible for carrying out tasks. When an application is executed, it is decomposed into a set of independent tasks T_1, T_2, \ldots, T_n . Each task T_i is assigned a *actor* $A_{m,i}$, responsible for executing the task.

At the blockchain level, directly monitoring the state *S*(*A*) of an actor *A* during task execution introduces significant complexity. Ensuring that an actor remains alive and will completes its task is non-trivial. To simplify the blockchain's role, we delegate state tracking responsibilities to the actors themselves.

To achieve this, we introduce the concept of **Execution Pairs**,which consist of a main actor $A_{m,i}$ and a fallback actor $A_{f,i}$.

An **execution pair** $P_i = \{A_{m,i}, A_{f,i}\}\$ is defined as follows:

- *Main Actor* $A_{m,i}$: Executes the assigned task T_i and monitors the state of its fallback actor *Af,i*.
- *Fallback Actor* $A_{f,i}$: Monitors the state of the main actor $A_{m,i}$. If the main actor fails $(S(A_{m,i}) = \text{failed})$, the fallback actor assumes responsibility for executing T_i and notifies the orchestration layer to assign a new fallback actor.

The state of an actor A is defined as $S(A)$, where:

$$
S(A) \in \{\text{active}, \text{failed}\}.
$$

The state of an execution pair P_i is considered *operational* if at least one actor in the pair is active:

 $S(P_i)$ = **operational** $\iff S(A_{m,i})$ = active $\lor S(A_{f,i})$ = active.

¹https://fherma.io

Figure 5: App running scheme

If both $S(A_{m,i})$ = failed and $S(A_{f,i})$ = failed, the orchestration layer intervenes to reassign both roles in the execution pair *Pⁱ* .

At the blockchain level, task monitoring is simplified. Instead of continuously tracking the execution status, the blockchain verifies the completion of tasks only upon the expiration of their deadlines. If a deadline is reached and the task T_i remains incomplete, the orchestration layer resolves the issue by reassigning the task or penalizing the responsible actors(Pair). This reduces computational overhead and ensures scalability.

For each task T_i , the reward R_i and penalty P_i are distributed between the main actor $A_{m,i}$ and fallback actor $A_{f,i}$:

The general scheme is depicted on figure [5](#page-23-1)

5.9 Universal Encrypted Data Format

Most modern FHE schemes are based on a shared mathematical foundation: the **Ring Learning with Errors (RLWE)** problem. While these schemes differ in their internal representations and computational processes, their common foundation allows for a **universal format for encrypted data representation**.

This universal format abstracts encrypted data into a structure compatible with all RLWE-based schemes. At its core, the format represents any ciphertext as a collection of *LWE ciphertexts*, serving as the building blocks of RLWE systems. This abstraction bridges differences between specific FHE schemes, such as BGV, CKKS, or TFHE, and provides a common ground for computations.

This approach is important for extensibility. Adding support for new encryption schemes based on the same problem requires only defining their mapping to and from the universal format. This eliminates the need for extensive updates to existing components and ensures that the system remains adaptable to emerging technologies. Furthermore, this abstraction facilitates **interoperability** among actors using different FHE libraries. By relying on a shared data format, actors can collaborate seamlessly, regardless of the libraries they utilize.

The universal representation also simplifies the instruction set architecture (ISA). Instead of tailoring instructions to the intricacies of individual schemes, operations like addition, multiplication, and rotations can be uniformly applied to ciphertexts in the

Figure 6: Ecosystem

universal format. This reduces complexity and enhances consistency across the system.

The implementation of this approach organizes ciphertexts into standardized components that actors can interpret using their preferred FHE library. For example: The details on the format is presented in Appendix 3.

6 Ecosystem

Our ecosystem is structured around several key elements, each contributing to the overall goal of fostering innovation and simplifying the development of FHE applications. The ecosystem is designed to ensure openness, modularity, and continuous evolution, addressing the challenges of privacy-preserving computations in both Web2 and Web3 domains.

6.1 FHE Computer

At the heart of the ecosystem is the FHE Computer, which serves as the primary platform for executing privacy-preserving computations. The computer provides a decentralized infrastructure capable of securely processing encrypted data, ensuring the confidentiality of sensitive information while enabling complex computations.

6.2 Polycircuit

The **Polycircuit**^{[2](#page-24-4)} repository is a collection of modular, reusable components designed to simplify the development of FHE applications. These components represent high-level FHE operations and functionalities, allowing developers to focus on application logic without needing to implement low-level cryptographic details. Polycircuit provides an extensible library that evolves with the needs of the community.

6.3 FHERMA

The **FHERMA**[3](#page-24-5) platform supports the continuous improvement and expansion of the Polycircuit repository. Through challenges hosted on FHERMA, developers can propose new components or optimize existing ones. Submitted components are benchmarked in

³https://fherma.io

²https://github.com/fairmath/polycircuit

a fair and transparent environment, and top-performing solutions are integrated into Polycircuit. This approach incentivizes innovation and ensures that only the most efficient and reliable components are added to the ecosystem.

6.4 fhelang

The **fhelang**^{[4](#page-25-5)} compiler is built on the MLIR stack and is specifically tailored to the concept of high-level FHE components. It enables developers to describe applications in terms of modular FHE operations, which are then translated into optimized instructions for execution on the FHE Computer. By abstracting the complexity of FHE implementation, fhelang reduces barriers for developers and ensures that applications are both performant and secure.

7 Usecases

Our model is designed to support a broad range of use cases. Below, we outline several potential applications that can be executed on the FHE Computer.

7.1 Collaborative AI

Collaborative AI allows multiple parties to securely and privately collaborate on AI models without exposing their individual data. Each party encrypts their data and submits it to the decentralized network. The FHE computer performs computations on the encrypted data, ensuring privacy and security throughout the process. The results are then decrypted and shared with the participants, allowing them to benefit from collective insights while maintaining data confidentiality. This approach leverages the strengths of decentralized architecture and FHE to facilitate secure, collaborative AI development and deployment.

7.2 Private Finance

Private finance enables secure and confidential financial services, such as lending, without revealing sensitive data. In a lending scenario, borrowers and lenders encrypt their financial information using Fully Homomorphic Encryption and submit it to the decentralized network. The FHE computer processes the encrypted data to assess creditworthiness, calculate loan terms, and perform other necessary computations while keeping the underlying data private. The results are then decrypted and provided to the participants, allowing them to make informed decisions without compromising their financial privacy. This approach ensures secure, transparent, and efficient financial transactions in a decentralized environment.

7.3 Peer-to-peer private transactions

In this example we would like to build blockchain where user balances are encrypted with the only corresponding user key, and it is possible to send tokens from one wallet to other keeping in secret the amount of tokens. Here's how it might work: Let *alice*_*encrypted*_*balance* and *bob*_*encrypted*_*balance* two ciphertexts represented the current balances of Alice and Bob respectively. Alice wants to send X tokens to Bob, but in such a way that no one except Bob can know how many tokens were sent. In order to do this, Alice generate two ciphertexts $e_1 = Enc_{alice}k(X)$ $e_2 = Enc_{bob}k(X)$. To send tokens from Alice to Bob it is needed to update the Alice and Bob balances with:

alice encrypted balance = *alice* encrypted balance1 − e_1

⁴ETA: Q4 2024

bob encrypted balance = *bob* encrypted balance + e_2

But we don't just do it. We need to prove a few statements:

- 1. Alice has enough tokens
- 2. e_1 and e_2 are store the same value.

To prove second statement we just use the **Proof of Ciphertext Equality** protocol. To prove the first statement we will use the *sign* function, defined as follows:

$$
sign(x) = \begin{cases} -1 & \text{if } x < 0, \\ 1 & \text{if } x \ge 0. \end{cases}
$$

Alice should prove that *alice*_*encrypted*_*balance*−*e*¹ *>*= 0, which is equal to *sign*(*alice*_*encrypted*_*balance*− e_1) = 1. let e_{12} = *alice_encrypted_balance* – e_1 Now can use the **Proof of Ciphertext Content** protocol and prove that ciphertext *e*12 is equal to 1, which will be the proof that Alice has enough tokens. At the last stage, after confirming transaction on-chain the user balances will be updated.

7.4 Privacy-Preserving Transactions (general case)

Web3 emphasizes user privacy and security. With the platform one can ensure that a transaction or an operation on a blockchain network meets specific criteria (like a minimum transaction value) without revealing the exact details of the transaction. This is crucial in scenarios like private or anonymous transactions on decentralized platforms.

7.5 Decentralized Identity Verification

In Web3, users control their own identity without relying on a central authority. On the platform with **PoCC** and **PoCE** protocols it is possible to prove certain attributes or credentials in a user's identity (like age or membership) without revealing the actual data or other sensitive information tied to their identity.

7.6 Voting Systems

On the platform we can organize decentralized voting systems, and based on the **PoCC** protocol it is possible to prove that a vote has been cast for a particular option without revealing the voter's identity or how others have voted. This maintains the secrecy of the ballot while ensuring the vote's validity.

7.7 Compliance and Auditing

Polycircuit Ecosystem can be used to demonstrate compliance with regulatory requirements or internal rules within decentralized organizations, without exposing sensitive data. For example, proving that funds are being used for a specific purpose without revealing the details of all transactions.

A key part of our design is the choice of a security model. The protocol assumes that each computing node (Fair Math Node) could be malicious. To address security concerns, we have developed a Verification Framework that enables verifiable FHE computations. Our monetization model is designed to align with the interests of all participants, including developers who contribute to the platform's technological advancement.

We consider nodes in the network as malicious actors in an untrusted environment. There are two main reasons for this approach:

- In reality, many nodes act maliciously, and we must accept this as a fact.
- Low trust requirements towards computation providers make it easier to scale the protocol since adding new nodes does not require extensive verification.

This approach requires designing the system with the idea that the environment may be hostile, where nodes might try to commit fraud, collude, or intentionally breach data confidentiality.

Accepting this assumption influences all aspects of the platform's design and operation:

- 1. **Develop Security Protocols**: Security strategies and mechanisms must be designed to counter potential attacks, ensuring data and process protection.
- 2. **Verification and Auditing**: Methods are needed to verify that nodes perform computations correctly and to audit their actions to detect and prevent unauthorized or malicious behaviour.
- 3. **Decentralization and Risk Sharing**: Without a single trusted environment, it is important to distribute tasks and data across multiple nodes to minimize the risk of centralized attacks.
- 4. **Development of Attack-Resistant Protocols**: Communication and data processing protocols must be resistant to various types of attacks, including those targeting data confidentiality and integrity.

8 Conclusion

We have introduced a model for a decentralized FHE Computer. Our model is designed to support a wide range of use cases, addressing the diverse needs of both Web2 and Web3 applications. By leveraging the power of FHE, it provides a scalable and privacy-preserving framework for secure computations on encrypted data.

A key strength of our approach lies in its open and extensible ecosystem, which not only facilitates the development of advanced applications but also incentivizes contributions from the global developer community. Through mechanisms like the Polycircuit component repository, FHERMA benchmarking and challenge platform, and the fhelang compiler, our system fosters innovation while maintaining a fair and transparent revenue-sharing model for developers.

References

- [AKP24] Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. General functional bootstrapping using CKKS. Cryptology ePrint Archive, Paper 2024/1623, 2024.
- [AKPP24] Gurgen Arakelov, Nikita Kaskov, Daria Pianykh, and Yuriy Polyakov. FHERMA: Building the open-source FHE components library for practical use. Cryptology ePrint Archive, Paper 2024/612, 2024.
- [B⁺21] Vitalik Buterin et al. Ethereum 2.0 specification. [https://ethereum.org/](https://ethereum.org/en/eth2/) [en/eth2/](https://ethereum.org/en/eth2/), 2021.
- [BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Paper 2011/277, 2011. <https://eprint.iacr.org/2011/277>.
- [BKSS24] Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Bootstrapping small integers with CKKS. Cryptology ePrint Archive, Paper 2024/1637, 2024.
- [Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. Cryptology ePrint Archive, Paper 2012/078, 2012. <https://eprint.iacr.org/2012/078>.
- [BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46, 2018.
- [But15] Vitalik Buterin. Ethereum whitepaper: A next-generation smart contract and decentralized application platform. 2015.
- [CD16] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.
- [CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. Cryptology ePrint Archive, Paper 2016/870, 2016. [https://eprint.iacr.](https://eprint.iacr.org/2016/870) [org/2016/870](https://eprint.iacr.org/2016/870).
- [CKKS16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Paper 2016/421, 2016. <https://eprint.iacr.org/2016/421>.
- [CSPS18] Raymond Cheng, William Scott, Raluca Ada Popa, and Ion Stoica. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. Cryptology ePrint Archive, Paper 2018/909, 2018. [https:](https://eprint.iacr.org/2018/909) [//eprint.iacr.org/2018/909](https://eprint.iacr.org/2018/909).
- [DDD⁺24a] Morten Dahl, Clément Danjou, Daniel Demmler, Tore Frederiksen, Petar Ivanov, Nigel Smart, Louis Tremblay, Thibault Marc Joye, and Dragos Rotaru. fhevm: Confidential evm smart contracts using fully homomorphic encryption. [https://github.com/zama-ai/fhevm/blob/main/](https://github.com/zama-ai/fhevm/blob/main/fhevm-whitepaper.pdf) [fhevm-whitepaper.pdf](https://github.com/zama-ai/fhevm/blob/main/fhevm-whitepaper.pdf), 2024.
- [DDD⁺24b] Morten Dahl, Clément Danjou, Daniel Demmler, Tore Frederiksen, Petar Ivanov, Nigel Smart, Louis Tremblay, Thibault Thériault, Marc Joye, and Dragos Rotaru. fhevm: Confidential evm smart contracts using fully homomorphic encryption. Whitepaper, Zama AI, 2024.
- [DM14] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. Cryptology ePrint Archive, Paper 2014/816, 2014. <https://eprint.iacr.org/2014/816>.
- [DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby: A framework for efficient mixed-protocol secure two-party computation. *NDSS Symposium*, 2015.
- [FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. [https://](https://eprint.iacr.org/2012/144) eprint.iacr.org/2012/144.
- [Gen09] Craig Gentry. A fully homomorphic encryption scheme. 2009.
- [GMGT16] Matthew Green, Ian Miers, Christina Garman, and Zerocash Team. Zerocash: Decentralized anonymous payments from bitcoin. 2016.
- [Gro16] Jens Groth. On the size of pairing-based non-interactive zero-knowledge arguments. In *Advances in Cryptology - EUROCRYPT 2016*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.
- [KP20] David Kaplan and Jason Powell. Vulnerabilities and attacks in trusted execution environments. Technical Report, 2020. [https://doi.org/10.](https://doi.org/10.1109/MICRO.2019.00055) [1109/MICRO.2019.00055](https://doi.org/10.1109/MICRO.2019.00055).
- [LLW24] Zijing Li, Hongbo Li, and Zhengyang Wang. SIMD-style sorting of integer sequence in RLWE ciphertext. Cryptology ePrint Archive, Paper 2024/1648, 2024.
- [LPR09] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Paper 2009/623, 2009. Available at <https://eprint.iacr.org/2009/623>.
- [Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. [https:](https://bitcoin.org/bitcoin.pdf) [//bitcoin.org/bitcoin.pdf](https://bitcoin.org/bitcoin.pdf), 2008.
- $[P^+22]$ Polygon et al. zkevm: Zero-knowledge ethereum virtual machine. [https:](https://polygon.technology/solutions/polygon-zkevm/) [//polygon.technology/solutions/polygon-zkevm/](https://polygon.technology/solutions/polygon-zkevm/), 2022.
- [Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):1–40, 2009.
- [SV13] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic simd operations in spdz framework. *IACR Cryptology ePrint Archive*, 2013.
- [VKH23] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. Verifiable fully homomorphic encryption. *arXiv preprint arXiv:2301.07041*, 2023.
- [W⁺18] Barry Whitehat et al. zk-rollup: Scaling ethereum with zero knowledge proofs. [https://ethresear.ch/t/](https://ethresear.ch/t/rollup-scalable-transactions-using-snark/1763) [rollup-scalable-transactions-using-snark/1763](https://ethresear.ch/t/rollup-scalable-transactions-using-snark/1763), 2018.
- [Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.
- [Yao82] Andrew Chi-Chih Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS)*, 1982.

A Execution File Format

The execution file format defines how applications for the Fair Math FHE Computer are structured and represented. This format allows efficient orchestration and execution of tasks by providing all necessary information in a standardized manner.

A.1 Assumptions and Requirements

To construct the execution file, the following assumptions are made:

• The FHE Lang Compiler divides application code into blocks, each translated into executable tasks.

- The compiler provides dependency information between blocks, enabling correct task orchestration.
- Each block contains instructions interpretable by actors using the FHE interpreter.

A.2 Structure of the Execution File

The execution file consists of three primary sections:

Table 1: Structure of the Execution File

Size		Offset Purpose	
4 bytes	0x00	Magic number: [70 72 69 76] ("FHEL" in ASCII)	
2 bytes	0x04	Version: 16-bit file format version	
Variable	0x06	Version-specific payload of the executable file	

A.3 Version 0.1 Format

For version 0.1, represented as [0x01 0x00] in little-endian, the payload uses a marshaled JSON format.

A.3.1 Payload Structure

The payload consists of blocks. Each block contains the following fields:

N	Size	Offset (in payload)	Purpose
	4 bytes	0x00	Number of blocks (little-endian)
$\overline{2}$	4 bytes	0x04	Block ID (little-endian)
3	4 bytes	0x08	Length of dependent blocks array in elements (N)
4	4 bytes	0x0C	First element of dependent blocks array
$5\overline{)}$	4 bytes	0x10	Second element of dependent blocks array
\cdot			
6	8 bytes	$12 + N * 4$	Component ID^*
7	4 bytes	$20 + N * 4$	Length of block code (M) in bytes
8	M bytes	$24 + N * 4$	Block code

Table 2: Payload Format for Version 0.1

A.3.2 Component ID

If a block contains a single component call, the **Component ID** field specifies the component. Blocks containing custom code set the **Component ID** to 0xFFFFFFFFFFFFFFFF. For blocks with a single component call, the **Block Code Length** is 0, and the code section is absent. If the length is non-zero, the code section is ignored during execution.

B Instruction Sets

B.1 arith

B.2 fhe

B.2.1 fhe.bgv

BGV Instruction Set Operations

B.2.2 fhe.ckks

CKKS Instruction Set Operations

B.3 cggi

B.4 polycircuit

C CIFAR10 compiled code for FHE Computer

%721 = fhe.ckks.encode %ctx, %722_array : (lve.ckks_crypto_context, memref<4096xf64>) -> lve.rlve_plaintext
%722 = fhe.ckks.encode %ctx, %722_array : (lve.ckks_crypto_context, memref<4096xf64>) -> lve.rlve_plaintext
%723 = %30 = fhe.ckks.encode %ctx, %30_array : (lve.ckks_crypto_context, memref<4096xf64>) -> lve.rlve_plaintext
%30 = fhe.ckks.encode %ctx, %33_array : (lve.ckks_crypto_context, memref<4096xf64>) -> lve.rlve_plaintext
%33 = fhe. % input = fhe .ckks.encode %ctx, % v21_array : (lwe.ckks_crypto_context, memref<4096xf64) -> lwe.rlwe.plaintext % v22 = fhe.ckks.encode %ctx, % v23_array : (lwe.ckks_crypto_context, memref<4096xf64) -> lwe.rlwe.plaintext lwe.rlwe_ciphertext
%input_vec = fhe.ckks.mul %ctx, %input_vec, %w2 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_plaintext) -> lwe.rlwe_ciphertext
%tmp = fhe.ckks.mul %ctx, %input_vec, %w3 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_plaintext) -> lwe. rlwe_ciphertext
%enc out = fhe.ckks.a % enc_out = fhe . ckks . add % ctx , % tmp , % w4 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext

%tmp = fhe.ckks.mul %ctx, %input_vec, %w5 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_plaintext) -> lwe. %tmp = fhe.ckks.mul %ctx, %input_vec, %w5 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_plaintext) -> lwe.
rlwe_ciphertext
%tmp = fhe.ckks.rotate %ctx, %tmp, -1, %r_key : (lwe.ckks_crypto_context, lwe.rlwe_ciph % tmp = fhe . ckks . mul % ctx , % input_vec , % w6 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe.ckks.rotate %ctx, %tmp, -2, %r_key : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, i32, rotation_key) -> lwe
rlwe_ciphertext
%enc_out = fhe.ckks.add %ctx, %tmp, %enc_out : (lwe.ckks_crypto_context, lwe.rlwe_c rlwe_ciphertext
%enc_out = fhe.ckks .add %ctx, %tmp, %enc_out : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe
.rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , % input_vec , % w7 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -3, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe rlwe_ciphertext

"httle end (ctx, Xinput_vec, Xinput_vec, Xinkey : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.

rlwe_ciphertext, mult_key) -> lwe.rlwe_ciphertext

xt0 = fne.ckks.add (ctx, Xt0, Xt0 : (lwe.ckks_crypt % to influence was not multiply the . characterization of the . results in the . results in the context , lwe . rlwe ciphertext , lwe . rlwe plaintext) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t0 , % w9 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -1, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t0 , % w10 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext
%tmp = fhe .ckks .rotate %ctx , %tmp , -2, %r_key : (lwe .ckks_crypto_context , lwe .rlwe_ciphertext , i32 , rotation_key) -> lwe .
rlwe_ciphertext = % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t0 , % w11 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext
%tmp = fhe.ckps.rotate %ctx, %tmp, -3, %r_key : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, i32, rotation_key) -> lwe.
rlwe_ciphertext
%enc_out = fhe.ckks.add %ctx, %tmp, %enc_out : (lwe.ckks_crypto_cont . rlwe_ciphertext % t1 = fhe . ckks . mul % ctx , % t0 , % input_vec , % m_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext , mult_key) -> lwe.rlwe_ciphertext
%t1 = fhe.ckks.add %ctx, %t1, %t1 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe
rlwe_ciphertext %t1 = fhe.ckks.sub %ctx, %t1, %input_vec : (lve.ckks_crypto_context, lve.rlve_ciphertext, lve.rlve_ciphertext) -> lve.
| rlwe_ciphertext
%tmp = fhe.ckks.mul %ctx, %t1, %v12 : (lve.ckks_crypto_context, lve.rlve_ciphertext, rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t1 , % w13 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -1, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t1 , % w14 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -2, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext
%enc_out = fhe.ckks.add %ctx, %tmp, %enc_out : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe
.rlwe_ciphertext .
%tmp = fhe.ckks.mul %ctx, %t1, %w15 : (lwe.ckks_crypto_context, l rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -3, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % t2 = fhe . ckks . mul % ctx , % t0 , %t0 , % m_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext , mult_key) lwe.rlwe_ciphertext % t2 = fhe . ckks . add % ctx , % t2 , % t2 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext %t2 = fhe.ckks.sub %ctx, %t2, 1 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, f64) -> lwe.rlwe_ciphertext
%tmp = fhe.ckks.mul %ctx, %t2, %w16 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_plaintext) -> lwe. rlwe_ciphertext

% enc_out = fhe.ckks.add % ctx, % tmp, % enc_out : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe .rlve_ciphertext
"tump = fhe.ckks.mul "Lctx, "Lt2, "Lvi" : (lve.ckks_crypto_context, lve.rlve_ciphertext, lve.rlve_plaintext) -> lve.
"Ive_ciphertext = rlve_ciphertext
"Lump = fhe.ckks.rotate "Lctx, "Lump, -1, "Lr_key : (l rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t2 , % w18 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -2, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . r ine.cass.iocate Actx, Atmp, -z, Ai_sey . (iwe.cass_crypto_context, iwe.iiwe_crpnertext, ioz, rocation_sey) -> iwe.
|rlwe_ciphertext
out = fhe.ckks.add "Actx, "Atmp, "Aenc_out : (lwe.ckks_crypto_context, lwe.rlwe_cipherte %enc_out = fhe.ckks.add %ctx, %tmp, %enc_out : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe
"Iwe_ciphertext = flue.ckks.mul %ctx, %t2, %w19 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe rlwe_ciphertext
rlwe_ciphertext
%enc out = fhe.ckks.a . ckks. add % ctx , % tmp , % enc_out : (lwe. ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe . – ine.caas
rlwe_ciphert % t3 = fhe . ckks . mul % ctx , % t2 , % input_vec , % m_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext , mult_key) -> lwe.rlwe_ciphertext
%t3 = fhe.ckks.add %ctx, %t3, %t3 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe rlwe_ciphertext % t3 = fhe . ckks . sub % ctx , % t3 , % t1 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t3 , % w20 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t3 , % w21 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -1, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext
rlwe_ciphertext
%enc_out = fhe.ckks.a es..
s.add %ctx, %tmp, % enc_out : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t3 , % w22 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -2, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext
%enc_out = fhe.ckks.add %ctx, %tmp, %enc_out : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe
.rlwe_ciphertext .
%tmp = fhe.ckks.mul %ctx, %t3, %w23 : (lwe.ckks_crypto_context, l rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -3, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe .rlve_ciphertext
%t4 = fhe.ckks_crypto_context, lve.rlve_ciphertext, lve.rlve_ciphertext, lve.rlve_ciphertext, mult_key)
-> lve.rlve_ciphertext
%t4 = fhe.ckks.add %ctx, %t4, %t4 : (lve.ckks_crypto_context, lve.rlve_ciphert rlwe_ciphertext % t4 = fhe . ckks . sub % ctx , % t4 , % t0 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t4 , % w24 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe .rlve_ciphertext
"tump = fhe.ckks.mul "Lctx, "Lt4, "Lv25 : (lve.ckks_crypto_context, lve.rlve_ciphertext, lve.rlve_plaintext) -> lve.
"Ive_ciphertext = rlve_ciphertext
"Lump = fhe.ckks.rotate "Lctx, "Lump, -1, "Lr_key : (l rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe .rlve_ciphertext
"tump = fhe.ckks.mul "Lctx, "Lt4, "Lv26 : (lve.ckks_crypto_context, lve.rlve_ciphertext, lve.rlve_plaintext) -> lve.
"Ive_ciphertext = rlve_ciphertext
"Lump = fhe.ckks.rotate "Lctx, "Lump, -2, "Lr_key : (l rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t4 , % w27 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -3, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % to = fhe .ckks .mul %ctx , %t2 , %t1 , %m_key : (lwe .ckks_crypto_context , lwe .rlwe_ciphertext , lwe .rlwe_ciphertext , mult_key) -> lwe.rlwe_ciphertext
%t0 = fhe.ckks.add %ctx, %t0, %t0 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe
rlwe_ciphertext % to = ine.cks. sub %ctx, % to, % to . (iwe.cks_crypto_context, iwe.riwe_ciphertext, iwe.riwe_ciphertext) -> lwe.
% to = fhe.ckks.sub %ctx, %t0, %input_vec : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_cipherte rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t0 , % w28 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t0 , % w29 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext
%tmp = fhe.ckps.rotate %ctx, %tmp, -1, %r_key : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, i32, rotation_key) -> lwe.
rlwe_ciphertext
%enc_out = fhe.ckks.add %ctx, %tmp, %enc_out : (lwe.ckks_crypto_cont . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t0 , % w30 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -2, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext
%enc_out = fhe.ckks .add %ctx, %tmp, %enc_out : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe
.rlwe_ciphertext % tmp = ime. ckks. and ******
"The ciphertext" - flue_ciphertext , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe
"Xtmp = fhe.ckks.mul "Xctx", "Xt0", "Xu31 : (lwe.ckks_crypto_context , lwe.rlwe_ciphertext , lwe.rlwe rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -3, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext
.ckks.add % ctx, % tmp, % enc_out : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe % $\text{enc_out} = \text{fhe.ckks.ad}$.rlwe_ciphertext

%ti = fhe.ckks.mul %ctx, %t2, %t2, %m_key : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext, mult_key)
- -> lwe.rlwe_ciphertext
%t1 = fhe.ckks.add %ctx, %t1, %t1 : (lwe.ckks_crypto_context, lwe.rlwe_ciph rlwe_ciphertext
%t1 = fhe.ckks_crypto_context, lwe.rlwe_ciphertext, f64) -> lwe.rlwe_ciphertext
%tmp = fhe.ckks.mul %ctx, %t1, %w32 : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_plaintext) -> lwe rlwe_ciphertext
%enc_out = fhe.ckks.a
.ckks.add %ctx. %tmp. %enc_out : (lwe.ckks_crypto_context. lwe.rlwe_ciphertext. lwe.rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t1 , % w33 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext

"The effectives are formed the circle of the case of the complement of the complement of the complement of the

"The ciphertext of the ciphertext of the complement of the complement

"The ciphertext of the rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % tmp , -2, % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % tmp , % enc_out : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . mul % ctx , %t1 , % w35 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_plaintext) -> lwe . rlwe_ciphertext

%tmp = fhe.ckks.rotate %ctx, %tmp, -3, %r_key : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, i32, rotation_key) -> lwe. %tmp = fhe.ckks.rotate %ctx, %tmp, -3, %r_key : (lve.ckks_crypto_context, lve.rlve_ciphertext, i32, rotation_key) -> lve.
Tlwe_ciphertext
%enc_out = fhe.ckks.add %ctx, %tmp, %enc_out : (lve.ckks_crypto_context, lve.rlve_ci % enc_out = fhe . ckks . add % ctx , % enc_out , % tmp : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % enc_out , 800 , % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lve.rlve_ciphertext
%enc_out = fhe.ckks.add %ctx, %enc_out, %tmp : (lve.ckks_crypto_context, lve.rlve_ciphertext, lve.rlve_ciphertext) -> lve
.rlve_ciphertext = fhe.ckks.add %ctx, %enc_out, 400, %r_key : (lve.ckks_crypto_c lwe . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % enc_out , % tmp : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % enc_out , 200 , % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lve.rlve_ciphertext
%enc_out = fhe.ckks.add %ctx, %enc_out, %tmp : (lve.ckks_crypto_context, lve.rlve_ciphertext, lve.rlve_ciphertext) -> lve
%tmp = fhe.ckks.rotate %ctx, %enc_out, 100, %r_key : (lve.ckks_crypto_context, l . rlwe_ciphertext % tmp = fhe .ckks .rotate %ctx , % enc_out , 50 , %r_key : (lwe .ckks_crypto_context , lwe .rlwe_ciphertext , i32 , rotation_key) -> lwe.rlwe_ciphertext
%enc_out = fhe.ckks.add %ctx, %enc_out, %tmp : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe . .rlwe_ciphertext
%enc_out1 = fhe.ckks.rotate %ctx, %enc_out, 40, %r_key : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, i32, rotation_key
) -> lwe.rlwe_ciphertext % tmp = fhe . ckks . rotate % ctx , % enc_out , 20 , % r_key : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , i32 , rotation_key) -> lwe.rlwe_ciphertext
%enc_out = fhe.ckks.add %ctx, %enc_out, %tmp : (lwe.ckks_crypto_context, lwe.rlwe_ciphertext, lwe.rlwe_ciphertext) -> lwe . .rlve_ciphertext
%tmp = fhe.ckks.rctate %ctx, %enc_out, 10, %r_key : (lwe.ckks_crypto_context, lwe.rlve_ciphertext, i32, rotation_key) ->
lwe.rlve_ciphertext
%enc_out = fhe.ckks.add %ctx, %enc_out, %tmp : (lwe.ckks_crypto_ . rlwe_ciphertext % enc_out = fhe . ckks . add % ctx , % enc_out , % enc_out1 : (lwe . ckks_crypto_context , lwe . rlwe_ciphertext , lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext % enc_out = return % enc_out : (lwe . rlwe_ciphertext) -> lwe . rlwe_ciphertext