fhEVM

Confidential EVM Smart Contracts using Fully Homomorphic Encryption*

Zama (x: @zama_fhe)

1 INTRODUCTION

A blockchain as a decentralized mechanism for data storage and processing requires transparency for network members to reach consensus on the evolving state of the system. This transparency inherently comes with great challenges regarding confidentiality as all onchain data is widely distributed and publicly visible, even when hidden behind pseudonymous addresses. Solving this challenge, while ensuring the strong security guarantees of existing blockchains is an ongoing field of research, and is necessary for mass adoption of decentralized applications.

Our aim is to solve this challenge for general-purpose (meaning Turing-complete) blockchains such as Ethereum. Our design follows four key principles:

- There should be no impact on the security of the underlying blockchain
- Everything should be publicly verifiable
- Developers should be able to write confidential smart contracts directly in Solidity, without being experts in cryptography
- Confidential smart contracts should be fully composable with each other

Our solutions combines fully homomorphic encryption (FHE) for the confidential computation, threshold MPC protocols for FHE key generation and ciphertexts decryption, and Zero-Knowledge Proofs of Knowledge (ZKPoK) to ensure the correctness and integrity of encrypted inputs.

As all FHE computations are deterministic, our design allows for existing consensus protocols to be used. This ensures that while the data itself is confidential, the computation itself can be done publicly in a verifiable manner.

1.1 Our contributions

This document introduces the fhEVM, a new confidential smart contract protocol for the Ethereum Virtual Machine (EVM). Our contributions include:

- A native fhEVM protocol (fhEVM-native) that adds confidential smart contracts capabilities to Layer 1/2/3 chains via the integration of dedicated lifecycle functions.
- An fhEVM coprocessor (fhEVM-coprocessor), which enables confidential smart contracts in non-FHE enabled chains such as Ethereum, without any changes to the chain itself.
- A Solidity library that makes it easy for smart-contract developers to use encrypted data in their contracts, without any changes to compilation tools.

Note that while this document focuses on the EVM, our protocol can easily be adapted to support non-EVM ecosystems, such as Solana, Cosmos, Sui, Aptos and others.

1.2 Applications

While the fhEVM enhances the security and confidentiality of many applications, it also enables an entirely new design space where confidentiality is a strong requirement. Here are some examples.

*Version 2.0.0 (November 13, 2024).

1.2.1 Confidential ERC-20 tokens. The ERC-20 token standard for fungible tokens is an important standard for blockchains. However, by the public nature of blockchain systems, the amounts being transferred, as well as the balances of token holders are public. This leads both to privacy concerns for individuals [BSBQ21] as well as the impossibility for large institutions to use public blockchains without revealing their strategy to competitors.

The fhEVM solves this by enables both balances and amounts being transferred to remain encrypted end-to-end, while still being composable with DeFi and other applications. Furthermore, public and confidential ERC-20 tokens are composable, as one can convert a public ERC-20 token into a confidential one as easily as one would wrap a non-ERC-20 token (e.g. ETH) into an ERC-20 equivalent (e.g. wETH).

1.2.2 Confidential Swaps. Auto-Marker Makers (AMMs) such as Uniswap are a fundamental primitive in decentralized finance (DeFi), and are being used to swap hundreds of billions of dollars worth of assets onchain. A consequence of the transaction data being public is front-running and MEV, which aim to extract value from end-users by interposing other transactions around them. While this is beneficial to some network participants, it is typically detrimental to the end user.

The fhEVM solves this by enabling swaps to be done using confidential ERC-20 tokens, where the amounts being transferred are encrypted. While the optimal confidential AMM design is still a topic of research, it is now possible to have fully composable, confidential DeFi protocols.

1.2.3 Confidential Voting. Decentralized autonomous organizations (DAOs) have become increasingly popular in the blockchain space as a way to create self-governing communities without a central authority. They rely on smart contracts to enforce rules and make decisions based on the votes from their members, typically using the number of tokens held as voting power. As everything is done publicly, large token holders can be blackmailed or bribed to vote in a specific way, hindering the fairness of DAO governance.

The fhEVM solves this by enables votes to be casted confidentially: both the vote and the number of tokens voted with can be kept confidential, while being fully onchain. The result itself would be public, but the individual votes would not.

1.2.4 Decentralized Identities. Decentralized identifiers (DIDs) are a novel type of identifier that enables verifiable and self-sovereign digital identities for individuals and organizations. Utilizing encrypted data, smart contracts that live in the fhEVM have the capability to store and process sensitive information related to a user's identity securely, safeguarding user privacy throughout the process. For example, a central authority or government could publish the encrypted birth date of a consenting user to a smart contract. Subsequently, authorized parties could query the smart contract to gain information about the user's age (e.g., whether they have the age of majority) when necessary.

1.3 Related work

There are multiple ways to achieve smart contract computations on private inputs. Although some of the listed solutions can use a combination of the following techniques we (grossly) classify them into four categories based on the predominant technology being used: zero-knowledge (ZK) proofs, trusted execution environments (TEEs), secure multi-party computation (MPC), and homomorphic encryption (FHE).

1.3.1 Zero-knowledge (ZK) proofs. ZK proofs tackle the privacy challenge by keeping only committed data and proofs of correct computation onchain. However, the data must be known in plaintext to compute on it, meaning a plaintext copy of the data must be kept somewhere. This can work well when only data from a single party is required for any computation, but raises the issue of what to do for applications requiring data from multiple parties.

ZCash [BCG+14] and Monero [Mon23] provide anonymity to both sender and receiver ends on transaction while keeping the amount of exchanged coins shielded using Pedersen commitments [Ped92].

Zexe [BCG+20] and VeriZexe [XCZ+23] allow arbitrary scripts to be evaluated within zero-cash style blockchains using zkSNARKs. The limitation is that since multiple smart contracts / parties cannot access encrypted state onchain, the input data has to be known by at least one party

to generate the zkSNARKs. Currently it is impossible to update encrypted states without revealing them using this methodology.

Hawk [KMS+16] uses ZK proofs where the inputs to the smart contracts are revealed to a trusted manager that does the computation.

In our solution, the computation happens directly on the encrypted data, meaning that mixing data from multiple users is straightforward and the computation can happen onchain.

1.3.2 Trusted execution environments (TEEs). Blockchain systems based on TEEs only store encrypted data onchain, and perform computations by decrypting the data inside secure enclave that holds the decryption keys [YXC+18, KGM19, CZK+19, SCR23, Oas23, Pha23]. The security of these solutions depends on the decryption keys being safely contained within the secure enclaves. This makes the user depend on the secure enclave hardware and their manufacturers which rely on a *remote attestation* mechanism [CD16].

The enclave approach was shown several times to be vulnerable against several side-channel attacks [VMW+18, LSG+18, KHF+19, vMK+21, TKK+22, vSSY+22], including attacks that simply observe leakage from memory access-patterns [JLLJ+23].

1.3.3 Multi-party computation (MPC). zkHawk [BCT21] and V-zkHawk [BT22] replace the trusted manager from Hawk [KMS+16] with an MPC protocol where all the input parties need to be on-line to participate.

Eagle [BCDF23] improves upon the Hawk constructions by having the clients outsource their inputs to an MPC engine which does the computation for them. Eagle also adds features such as identifiable abort and public verifiability to the outsourced MPC engine. Even though in Eagle the input parties do not have to be on-line all the time, they need to do one round of interaction with the MPC engine to provide inputs [DDN+16].

Although very detailed, there is no information on how privacy-preserving storage is achieved in Partisia [Par23] in their yellow paper.

1.3.4 Homomorphic encryption (FHE). Some solutions based on homomorphic encryption have been proposed based on *partially* homomorphic encryption, which limits the type of operations that can be performed and therefore only support a certain class of smart contracts and applications. For example, Zether [BAZB20] uses an ElGamal-based encryption scheme which ensures private transfers of funds. Alas, this is not enough to achieve full confidentiality when it comes to more sophisticated smart contracts.

Zkay [SBG+19] defines how to execute Ethereum smart contracts private using FHE but uses a trusted third party that holds the decryption key.

smartFHE [SWA23] uses BFV [FV12] as the underlying (HE) block to build FHE. In their setting, each wallet locally runs the BFV key generation procedure to get a public and a secret key pair. Multiple wallets have different keys so in order to execute more complicated smart contracts such as blind auctions they need to run a distributed key generation protocol and produce a joint (pk, sk). The ciphertexts involved in the computation then need to be re-encrypted under the new pk, and after performing the blind auction homomorphically, they need to run a distributed decryption to get the result (although the distributed decryption protocol is not detailed in their paper).

PESCA [Dai22] uses a similar architecture to ours making use of a global public key pk which everyone uses to encrypt their balances and a threshold FHE protocol to help decrypt outputs. One major difference is that their threshold protocol works modulo $q = p$ a prime number whereas ours works for a ring $\mathbb{Z}_q$ with q being typically a power of 2. Another difference is that in PESCA, the threshold FHE modulus q is exponential in the number of parties n; i.e., the ciphertext modulus must increase by a factor of $(n!)^3$ compared to non-threshold schemes.

Some Ethereum Improvement Proposals (EIP) explored the idea of adding homomorphically encrypted storage to the EVM [Sil17].

2 KEY CONCEPTS

This section gives a high level overview of the key concepts behind the fhEVM.

2.1 fhEVM

The fhEVM is a protocol leveraging FHE, MPC and ZK to enable confidential smart contracts on EVM blockchains.

The fhEVM comes in two flavors: *fhEVM-native*, which

is integrated directly into an L1 through a set of lifecycle functions, and *fhEVM-coprocessor*, which augments existing L1s with FHE capabilities, without requiring changes to the L1 itself. We will use the generic term fhEVM when the distinction between the native and coprocessor versions isn't relevant. Similarly, we will use the term Layer 1 (L1) to refer to L1s, L2s and L3s regardless of their operating model.

We make no assumption about the consensus layer used in the L1, beyond relying on it for providing agreed-upon blocks of transactions to execute and public signature keys of current validators. Importantly, we make no changes to the consensus protocols.

2.2 Key Management System (KMS)

One of the core component of the fhEVM is the Key Management System (KMS), which is responsible for generating FHE keys, decrypting and reencrypting ciphertexts, generating Common Reference Strings (CRS) and verifying Zero-Knowledge Proofs of Plaintext Knowledge (ZKPoK).

While a centralized or HSM-based KMS can be used, we rely instead on a decentralized KMS leveraging a novel threshold MPC protocol developed at Zama. This ensures no single entity can access the FHE keys or decrypt ciphertexts, while offering public verifiability of the requests. Zama's TKMS is described in detail in [Zam24].

2.3 Gateway

A Gateway is an offchain relayer that abstracts away most of the interactions between the KMS and fhEVM, enabling dApp developers to focus on their application logic rather than having to implement complex transaction flows. While Zama provides a high-performance cloud Gateway, anyone can spin their own in a trustless manner.

The Gateway exposes APIs for decryptions, reencryptions, ZKPoK verification, ciphertext storage, and more. It is also responsible for updating the list of L1 validators on the KMS and the list of KMS nodes on the L1, which are needed to verify the legitimacy of the decryption requests and outputs.

2.4 Coprocessor

Coprocessors are offchain services that observe the L1 and perform some associated tasks. In the case of the fhEVM, the coprocessor is responsible for two things: storing ciphertexts and executing FHE operations.

Just like blockchains, coprocessors can be deployed under various security assumptions: as a trusted service, as an optimistic rollup, as an L1 with consensus, as an Actively Validated Service (AVS), etc. Furthermore, the execution of FHE operations can be dissociated from the ciphertext storage, which can be a separate Data Availability layer with its own security assumptions.

Our design prevents silent attacks by ensuring fhEVM coprocessors are publicly verifiable: anyone can recompute ciphertexts stored by the coprocessor and see the decryption requests made to the KMS.

2.5 Client SDK

The fhEVM Client SDK offers developers convenient tools to easily encrypt data, generate ZKPoKs, interact with the Gateway and submit FHE transactions onchain. It is typically integrated into the application's frontend, and uses WASM for optimal performance.

2.6 FHE keys

Our protocol relies on a global FHE key under which all inputs and private state are encrypted. This is an important design decision since it enables composability between contracts and multi-user applications.

This global network FHE key is generated by the KMS, and comprises a public encryption key used by users to encrypt their inputs, a public evaluation key, used by FHE nodes to perform FHE computations, and a private key, used by the KMS to decrypt ciphertexts.

FHE keys can be updated as needed, without interrupting the KMS and fhEVM operations.

2.7 Certified ciphertexts

In order to use an encrypted input when calling a confidential smart contract, users are required to submit a *certified ciphertext*, which consists of the input encrypted using the global public encryption key, and an associated valid Zero-Knowledge Proof of plaintext Knowledge (ZKPoK) [Lib24].

The ZKPoK ensures that the ciphertext is well-formed and that the user knows the underlying plaintext message. This ensures that no-one can take existing ciphertexts from the onchain state and pass them as inputs to smart contracts

to decrypt them. The ZKPoK scheme used in the fhEVM requires a global *common reference string* (CRS), which is securely generated by the KMS.

ZKPoKs can either be verified on L1 directly, or on a third party validation service (such as the KMS), which returns a signed attestation that is verified on L1.

2.8 FHE computations

All FHE operations in the fhEVM use the TFHE-rs library [Zam22], which implements Zama's variant of the TFHE scheme [Joy21].

Smart contracts can perform FHE computations by interacting with the TFHE Solidity library described in Section 3. This library performs a *symbolic* execution of the computations to be made on ciphertexts by using "handles" instead of actual ciphertexts. The ciphertexts themselves are stored in a ciphertext database, which can either be onchain or offchain. Once the block containing the symbolic execution is finalized, the actual execution of the FHE computations can be done, either onchain by the L1 itself, or offchain by a coprocessor.

This "lazy" execution model allows a high transaction throughput on the L1, as no FHE computation is done during transactions. Users won't be able to tell the difference as they won't know that a ciphertext hasn't been computed until they try to decrypt it.

2.9 Access Control

In order to prevent malicious contracts or users from decrypting arbitrary ciphertexts, we leverage an onchain Access Control List (ACL) containing a list of ciphertext handles and addresses allowed to access them. This enables smart contracts to programmatically define rules for who is allowed to access which ciphertexts.

2.10 Decryption and re-encryption

Ciphertexts sometimes need to be decrypted or reencrypted under a different key. For example, the result of a confidential vote is public and thus can be decrypted, while token balances should only be viewable by their owners thus should instead be reencrypted under their keys.

Decryptions can be triggered by smart contracts using a "blockchain oracle"-like architecture between the fhEVM chain and the KMS. This is done via the Gateway contract, passing it the ciphertext to be decrypted and a callback function to put the result back onchain. This request will then be picked up by the offchain Gateway relayer, which will fetch the data, send the request to the KMS, and call the callback function.

For reencryptions, everything happens offchain, as no value needs to be put back onchain. The application (on behalf of the user) simply calls the Gateway, giving it the handle of the ciphertext to decrypt, and the user's public key to reencrypt the value under. The Gateway then calls the KMS, who sends back the reencrypted value, which is then be decrypted locally using the user's private key. At no point does the Gateway or KMS see the actual plaintext, as it is never decrypted.

It is important to note that ciphertexts can only be decrypted or reencrypted by addresses that are allowed to access them in the onchain ACL. The Gateway merely acts as a trustless relayer, and must provide a proof to the KMS that the address-ciphertext pair is indeed in the onchain ACL.

3 SOLIDITY LIBRARY

One of our main design goals with the fhEVM is to enable developers to easily build confidential applications, without learning cryptography or changing their existing workflow.

Solidity in particular has become the de-facto language for EVM smart contract development, benefiting from a rich, mature tooling ecosystem, including compilers, debuggers, IDEs, and libraries. This is why we chose to write our FHE library (called `TFHE`) in Solidity, offering a simple developer API for FHE computation and access control.

3.1 Encrypted types and operators

The `TFHE` library supports various encrypted data types:

- encrypted booleans: `ebool`
- encrypted unsigned integers: `euint4`, `euint8`, `euint16`, `euint32`, `euint64`, `euint128` and `euint256`
- encrypted signed integers: `eint4`, `eint8`, `eint16`, `eint32`, `eint64`, `eint128` and `eint256`
- encrypted bytes: `ebytes64`, `ebytes128`, `ebytes256`
- encrypted addresses: `eaddress`

```
function compute(
    euint64 x,
    euint64 y,
    euint64 z
) public returns (euint64) {
  return TFHE.mul(TFHE.add(x, y), z);
}
```

Listing 3.1: Example of computing on encrypted integers `x`, `y`, and `z`.

They are implemented as Solidity user defined value types and can be used for variables, parameters, and values in mappings and arrays. Note that they are not actual ciphertexts, but rather `uint256` ciphertext handles pointing to the actual ciphertexts (which are stored in a special database, either onchain for the fhEVM-native, or offchain for the fhEVM-coprocessor).

Contrary to most FHE solutions that only support additions and multiplications, we support all the usual operators for each family of encrypted types:

- `ebool`: logical operators (`and`, `or`, `xor`, `not`)
- `euintX` and `eintX`: bitwise logical operators (`and`, `or`, `xor`, `not`), arithmetic operators (`add`, `sub`, `mul`, `div`, `rem`, `neg`, `abs`, `sign`), comparison operators (`le`, `lt`, `ge`, `gt`, `eq`, `ne`, `min`, `max`), bit shifts and rotations (`shl`, `shr`, `rotl`, `rotr`), ternary operator `select`, and encrypted pseudo-random number generation `randEuintX`
- `ebytesX`: equality operators (`eq`, `ne`), ternary operator `select`
- `eaddress`: equality operators (`eq`, `ne`), ternary operator `select`

Example Listing 3.1 shows how to perform an addition and multiplication on encrypted 64 bit values.

When calling an FHE operation on the L1, the `TFHE` library does not actually execute the FHE computation. Instead, it does a *symbolic* execution, taking handles of ciphertexts as parameters, and producing a new handle pointing to the (future) output ciphertexts (c.f. Listing 3.2). The actual FHE execution is done separately by mapping each of these operations to their equivalent in TFHE-rs [Zam22], using the handles to read and write ciphertexts to and from the ciphertext database.

```
enum Operators {
  fheAdd
}

function add(
    euint64 x,
    euint64 y
) public returns (euint64) {
  return euint64(
    keccak256(
      abi.encodePacked(2, fheAdd, x, y)
    )
  );
}
```

Listing 3.2: Example of symbolic execution. 2 is used as a domain separator for ciphertext handles resulting from FHE computations.

3.2 Encrypted inputs

Smart contract functions can take two types of encrypted inputs: either handles to existing ciphertexts (e.g. an `euint64`), or certified ciphertexts that have never been seen before.

When passing a handle as input, the `TFHE` library will check that the contract is allowed to access the ciphertext before performing any FHE computation, reverting the transaction otherwise.

In the case of new, certified ciphertexts, the inputs must be accompanied by a valid ZKPoK that proves that the ciphertext is well formed, and that the user knows the plaintext. To save space and avoid computing multiple ZKPoKs, all encrypted inputs are packed into a single ciphertext. Smart contract then takes as parameters a byte array containing both the packed ciphertexts and ZKPoK, and a list of `einput` representing the index of each individual input in the packed ciphertext.

To use a certified ciphertext, the contract needs to cast them to a valid encrypted type by calling the `TFHE.asXXX(einput, calldata)` method (e.g. `TFHE.asEuint64(amount, inputData)`). This casting operator will verify the attestation, extract the required ciphertext from the packed data, store the ciphertext in the database and return a handle to that ciphertext. An example of this is shown in Listing 3.3.

3.3 Branching on encrypted values

Since comparing ciphertexts yields an encrypted boolean value, it is not possible to use it as part of an if-else or require statement. Instead, our library offers a multiplexer

```
function myfunction(
    einput param1,
    einput param2,
    bytes calldata inputData
){
    euint64 handle1 = TFHE.asEuint64(param1,
        inputData);
    ebool handle2 = TFHE.asEbool(param2, inputData);
    ...
}
```

Listing 3.3: Example of converting ciphertext-related input `amountCt` to an encrypted integer handle `amount`.

```
function myfunction(
    euint64 param
) public {
  ebool encryptedCondition = TFHE.lte(..., ...);
  euint64 paramOrZero = TFHE.select(
    encryptedCondition,
    param,              // if true
    TFHE.asEuint64(0)   // if false
  );
  ...
}
```

Listing 3.4: Example of using the select operator to nullify a value if an encrypted condition is false.

operator `TFHE.select` which allows selecting a value based on an encrypted boolean. This can be used in turn to emulate branching by nullifying the effect of an operation, as illustrated in Listing 3.4. Note that in these cases, transactions will always go through onchain, even though nothing will happen in practice (e.g. no tokens will actually be transferred).

3.4 Access Control

All ciphertexts are encrypted under the same public key, allowing for multi-user applications and composability between smart contracts. Deciding which address can access which ciphertext is done programmatically using an onchain Access Control List (ACL). When an address is granted access to a ciphertext, it can compute on it, decrypt/reencrypt it, and grant access to other addresses. As such, developers and users should be careful which contracts they interact with: just as they should not give token spending rights to malicious contracts, they should not give ciphertext access rights to malicious contracts.

Updating and reading the ACL is done using the following functions (as illustrated in Listing 3.5):

- `TFHE.allow(handle, address)` stores a permission permanently onchain, allowing an address to access a ciphertext at anytime. This is used for example when a user needs to decrypt or reencrypt a ciphertext at a later date, or when a contract needs to store the ciphertext in its state and reuse it later;
- `TFHE.allowTransient(handle, address)` stores the permission temporarily in transient storage, allowing an address to access a ciphertext only for the duration of a transaction. This is used for example when calling a contract from within an other contract and passing it a handle as parameter. It is also used internally by the `TFHE` library when a contract calls operators such as `TFHE.add(...)`, `TFHE.asEuint64(...)` and others;
- `TFHE.isAllowed(handle, address)` returns true if the address specified is allowed to access a ciphertext;
- `TFHE.isSenderAllowed(handle)` is a shorthand that returns true if the `msg.sender` is allowed to access the specified ciphertext. It is equivalent to calling `TFHE.isAllowed(handle, msg.sender)`, and should systematically be used by contracts to check that an address calling a function that takes handles as parameters is actually allowed to access those handles.

Note that if a contract tries to call `TFHE.allow(...)` or `TFHE.allowTransient(...)` without itself having access to the ciphertext, the transaction will revert.

3.5 Decryption

The fhEVM itself doesn't have the private decryption key, which is instead managed by the KMS. As such, requesting a decryption of a ciphertext is done via an oracle call to a Gateway service, which then forwards the request to the KMS and puts the result back onchain by triggering a callback function. An example is shown in Listing 3.6.

3.6 Reencryption

Contrary to decryption, reencryption doesn't require a transaction and is an entirely offchain process. The advantage of reencrypting a value is that no-one sees the plaintext, not even the KMS, as its is directly reencrypted into the user-provided public key.

To request a reencryption, the client application needs to call the Gateway service via a web API, which will then

```
function transfer(
    address from,
    address to,
    euint64 amount
) public {

    // make sure the caller can access 'amount'
    require(TFHE.isSenderAllowed(amount), ...);

    // Set amount to 0 if funds are insufficient
    euint64 txAmount = TFHE.select(
        TFHE.lte(amount, balances[from]),
        amount,
        TFHE.asEuint64(0)
    );

    // Do the transfer
    balances[from] = TFHE.sub(
        balances[from], txAmount);
    balances[to] = TFHE.add(
        balances[to], txAmount);

    // Allow users to access their updated balances
    TFHE.allow(balances[from], from);
    TFHE.allow(balances[to], to);

    // Allow this contract to access balances
    // This is needed for future computations
    TFHE.allow(balances[from], address(this));
    TFHE.allow(balances[to], address(this));

    // Use 'txAmount' in contract ABC
    TFHE.allowTransient(txAmount, address(ABC));
    ABC.someFunction(txAmount);
}
```

Listing 3.5: Example ERC20 transfer function.

fetch the necessary ciphertext handles from the L1 via view functions, forward the request to the KMS, and send back the result to the client application who can then decrypt it. A javascript example using our Client SDK is shown in Listing 3.7.

4 IMPLEMENTATION DETAILS

The following section describes implementation details of the fhEVM. When the generic term fhEVM is used instead of the specific term fhEVM-native or fhEVM-coprocessor, it means the implementation is identical for both versions.

Here is an example fhEVM-native encrypted ERC20 token transfer, which illustrates both the transaction and reencryption flows:

- **Transferring tokens**:
    1. User asks an ERC20 Application to transfer some tokens.
    2. The Application encrypts the amount, generates the associated ZKPoK and asks the KMS to verify it. The KMS then sends back an attestation of validity.
    3. The applications submits the ERC20 transfer transaction to the L1, passing it the encrypted amount and ZKPoK attestation as parameters.
    4. Upon receiving the transaction, the L1 EVM verifies the ZKPoK attestation, stores the input ciphertext onchain and does a symbolic FHE evaluation of the ERC20 transfer function.
    5. Once the L1 block containing the transaction is finalized, validators run the actual FHE computations and put back the resulting ciphertexts onchain.

```

function decryptValue(
    euint64 encryptedValue
) public {

    // Make sure the caller can access '
        encryptedValue'
    require(TFHE.isSenderAllowed(encryptedValue),
        ...);

    // Convert the handle into an equivalent uint256
    uint256[] memory cts = new uint256[](1);
    cts[0] = Gateway.toUint256(encryptedValue);

    // Send request to the gateway
    Gateway.requestDecryption(
        cts,
        this.myCustomCallback.selector,
        0, block.timestamp + 100, false
    );
}

function myCustomCallback(
    uint256 requestID,
    uint64 decryptedValue
) public onlyGateway returns (uint64) {
    ...
    return decryptedValue;
 }
```

Listing 3.6: Example decryption of an encrypted 64 bit value.

- **Viewing balances**:
    1. User asks the Application to view their balance.
    2. The application generates a public-private reencryption key-pair, asks the User to sign it using their EVM wallet and makes a request to the Gateway for reencryption of the balance ciphertext.

```
// Generate the keys used for the reencryption
const { publicKey, privateKey } =
    instance.generateKeypair();

// Create an EIP712 object for the user to sign.
const eip712 = instance.createEIP712(
    publicKey, CONTRACT_ADDRESS
);

// Request the user's signature on the public key
const signature = await window.ethereum.request({
    method: 'eth_signTypedData_v4', [
        USER_ADDRESS,
        JSON.stringify(eip712)
    ]
});

// Get the ciphertext to reencrypt
const encryptedERC20 = new Contract(
    CONTRACT_ADDRESS, abi, signer
).connect(provider);
const encryptedBalance =
    encryptedERC20.balanceOf(USER_ADDRESS);

// This function will call the gateway and decrypt
// the received value with the provided private key
const userBalance = instance.reencrypt(
    encryptedBalance,
    privateKey,
    publicKey,
    signature,
    CONTRACT_ADDRESS,
    USER_ADDRESS
);
```

Listing 3.7: Example reencryption of an encrypted ERC20 balance.

3. The Gateway then uses an L1 full node to retrieve the balance ciphertext and generate a merkle proof that the onchain ACL contains an address-ciphertext entry containing the User's address and balance ciphertext.
4. Next, the Gateway makes a reencryption request to the KMS, including the ciphertext to decrypt, the merkle proof and the User's public key.
5. The KMS verifies the proofs and signatures, reencrypts the balance under the User's key, and sends back the result to the Gateway which sends it back to the Application.
6. The Application finally decrypts the balance, verifies the KMS signature and sends back the plaintext balance to the User.

The end-to-end flow of transaction for both the fhEVM-native and fhEVM-coprocessor are sensibly the same, as shown in Fig. 4.1 and Fig. 4.2 respectively.

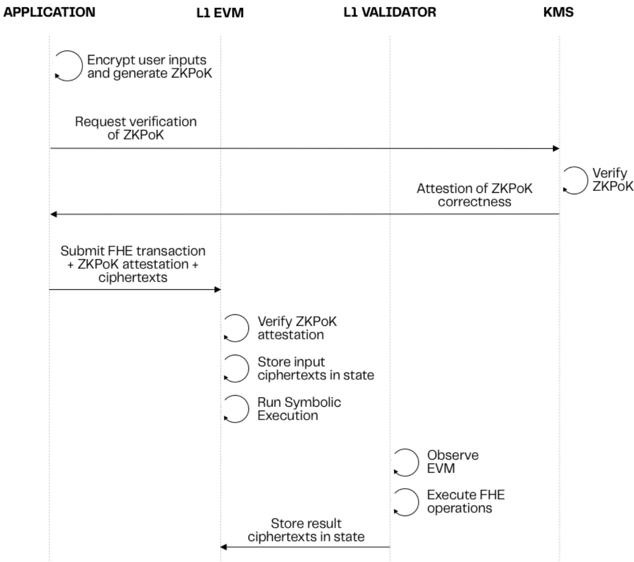

Figure 4.1: fhEVM-native transaction flow.

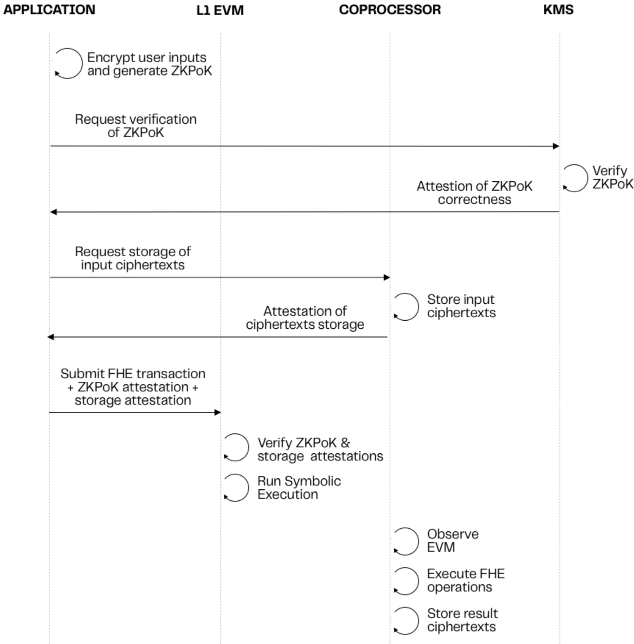

Figure 4.2: fhEVM-coprocessor transaction flow.

4.1 TFHE scheme

While many FHE schemes have been proposed over time, three in particular stood out and are in the process of being standardized by the ISO: BGV [BGV11], CKKS [CKKS16] and TFHE [Joy21]. BGV and CKKS have the advantage of enabling fast batched additions and multiplications, but have to approximate non-linear operations such as comparisons. Furthermore, they are often used in "leveled" mode, only allowing a limited number of consecutive operations after which the result must be decrypted. While useful for applications such as machine learning where the computation depth is bounded and the model can absorb approximations error, it cannot effectively be used in blockchain applications where compute depth is unbounded (you can transfer tokens as many times as you want!), and where approximations cannot be tolerated (approximate comparisons for example implies that checking a user has enough funds will sometimes pass even though it shouldn't).

The TFHE [Joy21] scheme on the other hand enables exact, unbounded computation through the use of a breakthrough operator called *programmable bootstrapping (PBS)*. In a nutshell, the PBS operator evaluates a table lookup homomorphically, while keeping the ciphertext noise low. This allows for exact, unlimited and arbitrary homomorphic univariate functions, which in turn can be used to create multivariate functions such as addition, multiplication, division, comparison, min-max, etc.

In our implementation, we make use of the TFHE scheme as a blackbox by leveraging Zama's TFHE-rs [Zam22] library, which currently offers the most complete feature set and the fastest performance on both CPU and GPU. Key generation, encryption, decryption, evaluation and ZKPoKs are all made available off the shelf via this library. We use the `PARAM_MESSAGE_2_CARRY_2_KS_PBS_TUNIFORM_2M64` parameter set, which gives 132 bits of AES-equivalent security (as per the LWE-Estimator [APS15]) and a probability of error of 2^{-64}, while being compatible with the threshold key generation and decryption done by the KMS. Benchmarks per operation are show in Table 4.2, where each operand is a ciphertext, aside from division where the divisor is a plaintext, and shift/rotations where the amount to shift/rotate is a plaintext. Timings are computed using an AMD EPYC 9R14 server with 192 cores for CPU, and a NVIDIA 2xH100 server for GPU.

Table 4.1: Timings in milliseconds of unsigned integer operations in TFHE-rs.

Operation	`euint8`	`euint16`	`euint32`	`euint64`
CPU (ms)				
Add / Sub	59	60	82	96
Mul	93	144	211	366
Div*	138	197	268	420
Eq / Neq	35	36	56	72
Lt / Gt	53	74	95	100
Min / Max	94	114	138	159
Shift* / Rot*	19	20	21	22
And / Or	19	20	20	22
Select	28	29	32	33
Rand	13	14	14	15
GPU (ms)				
Add / Sub	13	17	22	29
Mul	24	36	67	164
Div*	32	46	177	185
Eq / Neq	8	11	12	16
Lt / Gt	14	17	21	27
Min / Max	24	28	34	41
Shift* / Rot*	3	3	4	5
And / Or	3	3	4	5
select	10	11	12	14
Rand	4	4	5	6

* by a plaintext

In the following application benchmark, we measure the latency and throughput of ERC20 transfers, using 64 bit encrypted balances and transfer amounts. This uses the same configuration as the integer benchmark previously, with throughput scaling linearly with the number of servers.

Table 4.2: ERC20 transfers benchmark.

Hardware	Latency (ms)	Throughput (tx/s)
CPU	325	12
GPU	99	37

Table 4.3: Uncompressed public FHE keys sizes. pk is the public encryption key, ksk is the keyswitching key, bsk is the bootstrapping key and csk is the compression key.

pk	ksk	bsk	csk	CRS
16 kB	73 MB	58 MB	134 MB	81 MB

4.2 Key generation

The KMS is responsible for generating several private and public material (sizes are shown in Table 4.3):

- an FHE private decryption key, which is used to decrypt and reencrypt ciphertexts from the fhEVM
- a set of FHE public keys for keyswitching, bootstrapping and ciphertext compression.
- a public Common Reference String (CRS), used for the ZKPoKs client side.

In order to generate keys, the fhEVM needs to register with the KMS by uploading a smart contract with the logic to verify merkle inclusion proofs of the ACL and submitting a key generation transaction. The fhEVM nodes, client applications and others can then download them and start encrypting and computing on encrypted values. Similarly, keys can be rotated by requesting new keys to be generated, which will also produce keyswitching keys that the fhEVM can use to convert existing ciphertexts encrypted under the previous key into ciphertexts encrypted under the new key. This means that even in the case of a key breach, only previous ciphertexts are at risk, and not future ones, allowing to mitigate the impact of such potential attacks.

Before key generation happens, all FHE-related transactions are reverted because no valid input ciphertexts can be submitted to the systems without the existence of a public key and a CRS. In general, CRS generation and FHE key generation are mostly independent protocols and do not need to happened at the same time. In our context however, one is not useful without the other and as such both can happen at the same time.

4.3 FHE computation and ciphertext storage

Our design uses a novel *symbolic* execution model, which enables asynchronous, parallel FHE execution by separating the EVM smart contract execution from the FHE computation.

On the smart contract side, all operations are performed using *handles*, which are `uint256` values that either point to a transient ciphertext stored in memory, or to a persisted one stored in state (calling `SSTORE` on a handle will persist the corresponding ciphertext to state). FHE operators in the onchain TFHE library take handles as inputs, and produce new handles as outputs, without executing the actual FHE operation. This allows the L1 to maintain a high throughput, as this symbolic execution is very fast and consumes little gas. See Listing 3.2 for an example implementation in Solidity.

Once the block containing the symbolic execution is finalized, the actual FHE execution can take place. This is done by running a special purpose FHE runtime, which does the following (see algorithm Algorithm 1):

1. extract the list of FHE operations to perform and their associated input/output handles, and place them in a queue. Additionally, keep a list of handles that have been persisted to state.
2. for each FHE operation in the queue, use the handles to fetch the input ciphertexts from the state or transient storage. If such ciphertexts don't exist yet, push the operation to the back of the queue.
3. execute in parallel FHE operations for which inputs are available, save output ciphertexts in transient storage or in state if the handle was itself saved to state, and remove the operation from the queue.
4. repeat steps 2 and 3 until the queue is empty.

While simple, this algorithm allows parallel execution of FHE operations, yielding far better throughput than if they were executed sequentially at transaction time. We estimate that using GPUs, we could achieve 20-30 FHE transactions per second, covering a large majority of current EVM use cases. Nonetheless, to avoid having too much lag between the L1 symbolic execution and the FHE execution, the onchain TFHE library can implement a maximum "FHE gas" per block, thereby guaranteeing a maximum latency between the time a user submits a transaction to L1 and the time they can decrypt the result.

Algorithm 1 Evaluation of FHE Operations

```
1: queue ← extractFheOps(blockId)
2: stateHandles ← extractSstoreHandles(blockId)
3: temp ← new List()
4: while not queue.isEmpty() do
5:     (fheOp, inHandles, outHandles) ← queue.pop()
6:     if state.contains(inHandles) then
7:         inCts ← state.get(inHandles)
8:     else
9:         if temp.contains(inHandles) then
10:            inCts ← temp.get(inHandles)
11:        else
12:            inCts ← new List()
13:        end if
14:    end if
15:    if inCts.isEmpty() then
16:        queue.append(fheOp, inHandles, outHandles)
17:    else
18:        outCts ← execute(fheOp, inCts)
19:        if stateHandles.contains(outHandles) then
20:            state.append(outHandles, outCts)
21:        else
22:            temp.append(outHandles, outCts)
23:        end if
24:    end if
25: end while
```

Depending on the flavor of fhEVM being used, the implementation of the FHE runtime and storage can differ:

- **fhEVM-native**: Validators read their own state to determine which FHE operations to execute, reading and writing ciphertexts from a special purpose key-value storage contract onchain. Since the FHE computation happens outside the EVM however, there needs to be a mechanism by which validators can put back the results onchain and reach consensus on them. To solve this, we use special lifecycle methods that allow batch insertion of ciphertexts into the onchain ciphertext storage, as well as a special-purpose FHE scheduler that predicts how long it will take to execute FHE operations in a block. Validators then use this predicted execution time to insert ciphertexts at a given block height, allowing them to reach consensus. Since ciphertexts can only be decrypted after they have been inserted in state, the faster validators can execute FHE transactions, the better the user experience.

- **fhEVM-coprocessor**: An offchain cloud service runs a full node, extract the FHE operations to execute, and read/write ciphertexts from a local database or an external data availability layer such as IPFS. Contrary to the fhEVM-native implementation, coprocessors do not need to reach consensus, and so do not use a scheduler to commit to inserting ciphertexts at a given block height. Despite being offchain, the fact that input and output ciphertexts are publicly accessible means that anyone can verify the integrity of the coprocessor by recomputing the FHE operations and comparing the results. For additional security, coprocessors can run inside trusted enclaves such as AWS Nitro, which can provide attestations that the ciphertexts were produced using a specific software version.

An additional important consideration is the size of ciphertexts being stored and included in transactions. When encrypted naively, FHE ciphertexts can end up being several orders of magnitude larger than plaintext data (e.g. an `euint64` would take 526,720 B, a 65,000x expansion). To solve this issue, we leverage the TFHE-rs compact public key encryption to pack up to 4096 bits of inputs into a single 8 kB ciphertext, as well as post-computation compression to pack up to 512 bits of message into a single 2.2 kB ciphertext (sizes for various data types are shown in Table 4.4).

Note that since compressed ciphertexts are fixed-sized containers, the more bits of messages are packed, the smaller the expansion factor; thus, encrypting a single bit will always have the worse expansion factor. For reference, using ciphertext compression would allow to encrypt and store all of Ethereum's state on a single server with 500 TB of disk space.

Table 4.4: Sizes for various FHE data types.

Type	Message (bits)	Compressed Size (kB)
Inputs	1 - 4096	8.8
`ebool`	1	1.8
`euintX` / `eintX`	4 - 128	1.8
`eaddress`	120	1.8
`ebytes64`	512	2.2
`ebytes128`	1024	4.4
`ebytes256`	2048	8.8

4.4 Encrypted inputs and ZKPoKs

When a user wants to use a new ciphertext as an encrypted input to a smart contract, they need to include both the ciphertexts and a ZKPoK proving that they know the corresponding plaintexts and that the ciphertexts are well formed.

In TFHE-rs, ZKPoKs are implemented following the scheme from [Lib24], which allows proving the plaintext knowledge and correctness of the TFHE public key encryption. There are two mode of operation for the ZKPoKs: one with a faster prover and slower verifier, and another with a slower prover and faster verifier. In our protocol, we use the faster prover variant, as client-side hardware is often limited in comparison to validator hardware. Benchmarks are shown in Table 4.5, with the proving done in WASM in a Chromium browser running on a 16-core MacBook Pro M2, while the verification is done on an AMD EPYC 9R14 server with 192 cores.

As verifying a ZKPoK can take over a hundred milliseconds, verifying them sequentially for each transaction in the EVM would be both too expensive and too slow, even with dedicate precompiles. Since transactions need to be reverted on incorrect ZKPoKs, they have to be verified before an FHE computation can happen, acting as a bottleneck that would limit the maximal transaction throughput to 8 transactions per second independently of the FHE computation time itself. As such, we opted to implement the ZKPoK verification on the KMS itself, which acts as a trusted third-party verification service, moving the latency to the client-side rather than the EVM side. To verify a ZKPoK, the client application submits a transaction to the KMS chain, which verifies the ZKPoK and returns a signed attestation of its validity. This attestation is then included in the fhEVM transaction itself, and verified by smart contracts.

Table 4.5: Timings and sizes of ZKPoKs for 10 euint64.

Proving (browser)	Verification (server)	Proof Size
1.3 s	130 ms	950 B

To save space, all input ciphertexts are packed into a single one which can hold up to 4096 bits of message. As such, only one ZKPoK is required for the packed inputs, saving both on size and proving/verification time. The ZKPoK must have been generated using the user's address u and contract address a as auxiliary information to ensure that the input ciphertext can only be used in a transaction signed by u to a function in contract a.

To accept encrypted inputs, Solidity functions must include the following parameters (see Listing 3.3 for an example):

- a set of `einput`, representing the index of ciphertexts to extract from the packed input ciphertext
- a `calldatabytes` containing the ZKPoK attestation and ciphertext-related data.

The contract can then extract each encrypted input by calling `TFHE.asEuintXXX(einput, calldata)`, which will automatically verify the proofs, return a handle and allow the current contract access to the ciphertext for further computation.

While the developer API is identical, there is a difference between fhEVM-native and fhEVM-coprocessor in the content of the `calldatabytes`:

- **fhEVM-native**: the `calldatabytes` contains the ciphertext bytes themselves, which are stored in the onchain ciphertext storage. Handles are then derived deterministically from the ciphertext bytes and used in further computations.
- **fhEVM-coprocessor**: since ciphertexts are stored offchain, the user must first send the ciphertexts to the coprocessor along with the ZKPoK attestation. The coprocessor will verify the attestation, and if valid store the ciphertexts in its database, returning signed handles for each ciphertext. This list of signed handle is what will be included as the `calldatabytes` parameter, and parsed onchain when calling `TFHE.asEuintXXX(einput, calldata)`.

4.5 Access control

Access Control Lists (ACL) for ciphertexts are maintained by an onchain smart contract whose purpose is to keep track of which specific address can access which ciphertext. A smart contract is allowed to access a ciphertext if it is:

- given access by a contract or user that itself already has access (e.g. `TFHE.allow(ct, address)`);
- a certified ciphertext passed an an input to the contract;

```
contract ACL {

  type Handle is uint256;

  mapping(Handle => bool) public
       allowedForDecryption;
  mapping(Handle => mapping(address => bool)) public
        allowedAddresses;

  function isAllowed(
    Handle handle,
    address account
  ) public view returns (bool) {
    return allowedAddresses[handle][account];
  }

  // Allow use of `handle' for address `account'.
  function allow(
    Handle handle,
    address account
  ) external {
    if(msg.sender != address(coprocessor)){
      require(isAllowed(handle, msg.sender));
    }

    allowedAddresses[handle][account] = true;
  }

  // Mark `handle' as decryptable.
  function allowForDecryption(
    Handle memory handle
  ) external {
    require(isAllowed(handle, msg.sender));
    allowedForDecryption[handle] = true;
  }
}
```

Listing 4.1: Simplified access control list (ACL) contract.

- a trivial encryption (e.g. `TFHE.asEuint64(0)`);
- the output of an FHE operation on inputs the contract was allowed to access (e.g. `TFHE.add(ct1, ct2)`);
- the output of a parameter-less FHE operation (e.g. `TFHE.randEuint16()`);

The ACL smart contract mainly consists of a key-value store mapping pairs of handle and address to a boolean value indicating whether or not access is authorized. An example implementation is shown in Listing 4.1.

4.6 Decryption

If a smart contract wants to decrypt a ciphertext and put the plaintext back onchain, it can trigger an oracle-like asynchronous decryption, making use of a Gateway linking the blockchain system and the KMS. The decryption flow is shown in Figure 4.3.

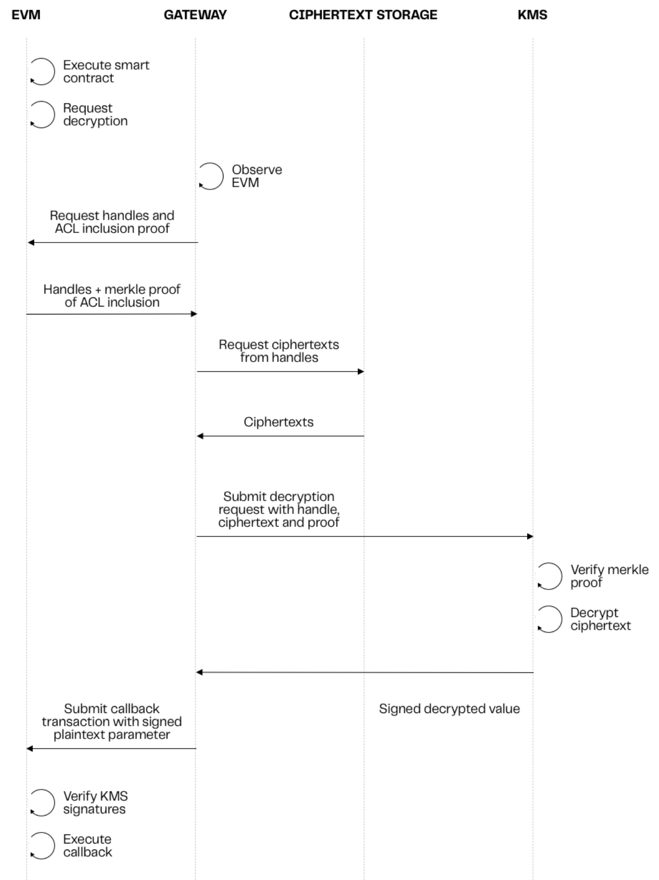

Figure 4.3: Decryption flow.

The Gateway is divided in two components: a Gateway smart contract (c.f. Listing 4.2) which lives on the blockchain and an offchain relayer called the Gateway service.

To trigger decryption of a ciphertext, smart contracts first need to call `TFHE.allowDecrypt(handle)` to mark the ciphertext as decryptable in the ACL. It can then call `Gateway.requestDecryption(...)` on the Gateway smart contract, passing it the ciphertext handle, callback and other parameters. Upon receiving a request, the Gateway contract emits an event containing the address of the calling contract, a callback function selector and the handle of the ciphertext.

The Gateway offchain relayer picks up this event, and does the following actions:

1. Fetch a Merkle proof from the L1 showing that the handle is marked as decryptable in the ACL;
2. Fetch the ciphertext corresponding to the handle from

the ciphertext storage;

3. Submit a decryption request containing the ciphertext bytes and the inclusion proof to the KMS;

4. Retrieve the decrypted value from the KMS;

5. Make a transaction to the L1 to trigger the callback, passing it the decrypted value as parameter.

Before executing the callback, the Gateway contract checks that the decrypted values were signed by the KMS, preventing illegitimate values to be put back onchain.

4.7 Reencryption

Users who want to decrypt a ciphertext without revealing the plaintext to the KMS or other observers can provide a public key under which the KMS should reencrypt the ciphertext. Reencryption is a fully offchain process, as shown in Fig. 4.4.

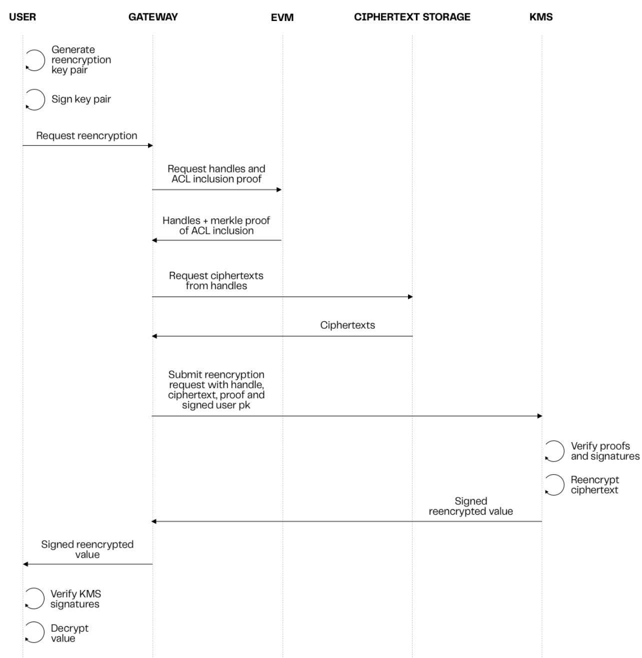

Figure 4.4: Reencryption flow.

Similarly to the decryption scenario, giving an address the ability to reencrypt a ciphertext is done by having the smart contract that owns it call `TFHE.allow(handle, address)` (c.f. Listings 3.4 and 4.1). Since this mutates the onchain ACL, it has to be done during a transaction, typically when the ciphertext is produced (e.g. when an ERC20 transfer occurs).

```
contract Gateway {
  // handle of a ciphertext
  type Ciphertext is uint256;

  struct DecryptionRequest {
    Ciphertext ct;
    address contractCaller;
    bytes4 callbackSelector;
  }

  mapping(uint256 => DecryptionRequest) internal
      decryptionRequests;

  uint256 public counter;

  function requestDecryption(
    Ciphertext ct,
    bytes4 callbackSelector,
  ) external returns (uint256){
    DecryptionRequest storage decReq =
        decryptionRequests[counter];

    decReq.ct = ct;
    decReq.contractCaller = msg.sender;
    decReq.callbackSelector = callbackSelector;

    emit EventDecryptionRequest(
      counter,
        ct,
        msg.sender,
        callbackSelector
      );

      counter++;
      return counter-1;
  }

  function fulfillRequest(
    bytes memory plaintext,
    bytes memory signatures
  ) external {
    DecryptionRequest decReq = decryptionRequests[
        requestID];

    (decryptionReq.contractCaller).call(
      abi.encodePacked(
        abi.encodeWithSelector(
          decReq.callbackSelector,
          requestID
          ),
        plaintext
      );
    );
  }
}
```

Listing 4.2: Simplified gateway contract.

When a user (or application) wants to reencrypt a ciphertext, they call the offchain Gateway service, providing it with the following items:

- The handle h_c of the ciphertext c to reencrypt;
- A public encryption key pk_u to reencrypt c under;
- The address a of the contract that owns h_c;

- A signed EIP-712 [BLE21] formatted object containing pk_u and a.

The Gateway then fetches the following complementary elements:

- The actual ciphertext bytes from the ciphertext storage;
- A Merkle proof of inclusion π_1 from the ACL contract storage that shows that a owns h_c;
- A Merkle proof of inclusion π_2 from the ACL contract storage that shows that h_c is marked as reencryptable by the requesting address.

The Gateway then requests a reencryption from the KMS, which verifies the proofs, executes the reencryption protocol. Upon completion, the Gateway forwards the KMS-signed reencrypted ciphertext to the user who can finally decrypt it using the private key they previously generated.

The reason two Merkle proofs are required for reencryption is because in practice, it's an application rather than a user that generates the keypair under which c will be reencrypted. Hence, in order to prevent a malicious application from requesting reencryptions with this keypair for *all* ciphertext owned by the user on the blockchain, we restrict the set of ciphertexts reencryptable with this keypair to the set of ciphertexts owned by the user and the contract a by using two merkle proofs and having the user sign an EIP-712 object containing the address of the contract a. This implies that whenever the user wants to obtain a reencryption of a ciphertext owned by itself and another smart contract a', it has to generate another keypair (sk'_u, pk'_u) (presumably with the help of another application) and sign another EIP-712 object containing pk'_u and a'.

Note that this design allows for programmable privacy by letting smart contract developers choose precisely who has access to read which encrypted values. The reencryption mechanism also readily extends to smart contract wallets, but we omit the details here for brevity.

5 FUTURE WORK

5.1 Coprocessor security

Coprocessors can be secured the same way as any other decentralized protocol:

- **Cryptoeconomics**: Coprocessors are publicly verifiable, as all encrypted inputs and outputs are accessible to anyone, allowing to recompute and compare results. Incentives can be put in place such that the coprocessor operator would be financially penalized by cheating.
- **ZK-FHE**: Using SNARKs allows coprocessors to prove the correctness of the FHE computation, which can then be verified by the L1 or KMS. TFHE bootstrapping has only been shown to be provable recently [TW24], taking 20 minon a 192 core CPU server.
- **Fraud proofs**: Rather than proving every FHE operation in ZK, which would take considerable time and computation effort, we can instead use fraud proofs in a similar way to optimistic rollups [KGC+18, ZEL+23], using ZK or other mechanism to prove that a subset of the FHE computation was done incorrectly.
- **Consensus**: Since FHE computations are fully deterministic, they can be replicated and verified by consensus. Whether coprocessors are implemented as L1 themselves or as Actively Validated Services (AVS), the idea is that multiple independent parties recompute the same task and compare results.

5.2 Hardware acceleration

Scaling fhEVM transactions to 1,000+ transactions per second or more cannot be done on CPU and GPU using current FHE schemes. There are several attempts are creating FHE accelerators using FPGA [BDTV22] and ASIC [Cor, Cry, Opt], with promising results.

REFERENCES

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015. URL: http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml.

[BAZB20] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In Joseph Bonneau and Nadia Heninger, editors, *FC 2020: 24th International Conference on Financial Cryptography and Data Security*, volume 12059 of *Lecture Notes in Computer Science*, pages 423–443, Kota Kinabalu, Malaysia, February 10–14, 2020. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-51280-4_23.

[BCDF23] Carsten Baum, James Hsin-yu Chiang, Bernardo David, and Tore Kasper Frederiksen. Eagle: Efficient privacy

preserving smart contracts. In Foteini Baldimtsi and Christian Cachin, editors, *FC 2023: 27th International Conference on Financial Cryptography and Data Security*, Lecture Notes in Computer Science, Bol, Croatia, May 1–5, 2023. Springer, Heidelberg, Germany. To appear. Preprint available as Cryptology ePrint Archive, Report 2022/1435 at https://ia.cr/2022/1435.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press. doi:10.1109/SP.2014.36.

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, pages 947–964, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press. doi:10.1109/SP40000.2020.00050.

[BCT21] Aritra Banerjee, Michael Clear, and Hitesh Tewari. zkHawk: Practical private smart contracts from MPC-based Hawk. In *3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 245–248, Paris, France, September 27–30, 2021. IEEE Computer Society. Extended version available as Cryptology ePrint Archive, Report 2021/501 at https://ia.cr/2021/501. doi:10.1109/BRAINS52497.2021.9569822.

[BDTV22] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. FPT: a fixed-point accelerator for torus fully homomorphic encryption. Cryptology ePrint Archive, Paper 2022/1635, 2022. https://eprint.iacr.org/2022/1635. URL: https://eprint.iacr.org/2022/1635.

[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. URL: https://eprint.iacr.org/2011/277.

[BLE21] Remco Bloemen, Leonid Logvinov, and Jacob Evans. EIP-712: Typed structured data hashing and signing. Ethereum Improvement Proposals, no. 712, 2021. URL: https://eips.ethereum.org/EIPS/eip-712.

[BSBQ21] Ferenc Béres, István András Seres, András A. Benczúr, and Mikerah Quintyne-Collins. Blockchain is watching you: Profiling and deanonymizing ethereum users. In *IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS 2021)*, pages 69–78, Online Event, August 23–26, 2021. IEEE Computer Society. doi:10.1109/DAPPS52256.2021.00013.

[BT22] Aritra Banerjee and Hitesh Tewari. Multiverse of HawkNess: A universally-composable MPC-based Hawk variant. *Cryptography*, 6(3):39, 2022. doi:10.3390/cryptography6030039.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[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. URL: https://eprint.iacr.org/2016/421.

[Cor] Cornami. Cornami fhe accelerator. URL: https://cornami.com.

[Cry] Fabric Cryptography. Introducing the vpu, an fhe and zk accelerator. URL: https://www.fabriccryptography.com/.

[CZK+19] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200, Stockholm, Sweden, June 17–19, 2019. IEEE Computer Society. doi:10.1109/EuroSP.2019.00023.

[Dai22] Wei Dai. PESCA: A privacy-enhancing smart-contract architecture. Cryptology ePrint Archive, Report 2022/1119, 2022. https://eprint.iacr.org/2022/1119.

[DDN+16] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In Jens Grossklags and Bart Preneel, editors, *FC 2016: 20th International Conference on Financial Cryptography and Data Security*, volume 9603 of *Lecture Notes in Computer Science*, pages 169–187, Christ Church, Barbados, February 22–26, 2016. Springer, Heidelberg, Germany. doi:10.1007/978-3-662-54970-4_10.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[JLLJ+23] Nerla Jean-Louis, Yunqi Li, Yan Ji, Harjasleen Malvai, Thomas Yurek, Sylvain Bellemare, and Andrew Miller. SGXonerated: Finding (and partially fixing) privacy flaws in TEE-based smart contract platforms without breaking the TEE. Cryptology ePrint Archive, Report 2023/378, 2023. https://eprint.iacr.org/2023/378.

[Joy21] Marc Joye. Guide to fully homomorphic encryption over the [discretized] torus. Cryptology ePrint Archive, Paper 2021/1402, 2021. https://eprint.iacr.org/2021/1402. URL: https://eprint.iacr.org/2021/1402.

[KGC+18] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1353–1370, Baltimore, MD, August 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner.

[KGM19] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *ISOC Network and Distributed System Security Symposium – NDSS 2019*, San Diego, CA, USA, February 24–27, 2019. The Internet Society. doi:10.14722/ndss.2019.23060.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative

execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press. doi:10.1109/SP.2019.00002.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press. doi:10.1109/SP.2016.55.

[Lib24] Benoît Libert. Vector commitments with proofs of smallness: Short range proofs and more. In Qiang Tang and Vanessa Teague, editors, *Public-Key Cryptography - PKC 2024 - 27th IACR International Conference on Practice and Theory of Public-Key Cryptography, Sydney, NSW, Australia, April 15-17, 2024, Proceedings, Part II*, volume 14602 of *Lecture Notes in Computer Science*, pages 36–67. Springer, 2024. doi:10.1007/978-3-031-57722-2_2.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018: 27th USENIX Security Symposium*, pages 973–990, Baltimore, MD, USA, August 15–17, 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[Mon23] Monero. Monero research lab (MRL), 2023. URL: https://www.getmonero.org/resources/research-lab/.

[Oas23] Oasis. Oasis network technology: Bringing privacy to Web3, 2023. URL: https://oasisprotocol.org/technology#overview.

[Opt] Optalysis. Optalysis fhe accelerator. URL: https://optalysys.com/.

[Par23] Partisia. Partisia blockchain (PBC), 2023. URL: https://partisiablockchain.com/resources.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany. doi:10.1007/3-540-46766-1_9.

[Pha23] Phala. Blockchain infrastructure. Phala Network Docs, 2023. URL: https://docs.phala.network/developers/advanced-topics/blockchain-infrastructure#the-architecture.

[SBG+19] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1759–1776, London, UK, November 11–15, 2019. ACM Press. doi:10.1145/3319535.3363222.

[SCR23] SCRT. Secret network overview: Private smart contracts on the blockchain, 2023. URL: https://scrt.network/about/about-secret-network/.

[Sil17] Silur. Homomorphically encrypted storage. Draft EIP, 2017. URL: https://github.com/Silur/EIPs/blob/4943fed23f82582f906ee46a113fe0b115836635/EIPS/eip-fhe.md.

[SWA23] Ravital Solomon, Rick Weber, and Ghada Almashaqbeh. smartFHE: Privacy-preserving smart contracts from fully homomorphic encryption. In *2023 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 309–331, Delft, Netherlands, July 3–7, 2023. IEEE Computer Society. doi:10.1109/EuroSP57164.2023.00027.

[TKK+22] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. SpecHammer: Combining spectre and rowhammer for new speculative attacks. In *2022 IEEE Symposium on Security and Privacy*, pages 681–698, San Francisco, CA, USA, May 22–26, 2022. IEEE Computer Society Press. doi:10.1109/SP46214.2022.9833802.

[TW24] Louis Tremblay Thibault and Michael Walter. Towards verifiable FHE in practice: Proving correct execution of TFHE's bootstrapping using plonky2. Cryptology ePrint Archive, Paper 2024/451, 2024. https://eprint.iacr.org/2024/451. URL: https://eprint.iacr.org/2024/451.

[vMK+21] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on intel CPUs via cache evictions. In *2021 IEEE Symposium on Security and Privacy*, pages 339–354, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press. doi:10.1109/SP40001.2021.00064.

[VMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018: 27th USENIX Security Symposium*, pages 991–1008, Baltimore, MD, USA, August 15–17, 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[vSSY+22] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SoK: SGX.Fail: How stuff get eXposed. https://sgx.fail, 2022.

[XCZ+23] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. VeriZexe: Decentralized private computation with universal setup. In Joe Calandrino and Carmela Troncoso, editors, *USENIX Security 2023: 32nd USENIX Security Symposium*, pages 4445–4462, Anaheim, CA, USA, August 9–11, 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/xiong.

[YXC+18] Rui Yuan, Yu-Bin Xia, Hai-Bo Chen, Bin-Yu Zang, and Jan Xie. ShadowEth: Private smart contract on public blockchain. *Journal of Computer Science and Technology*, 33:542–556, 2018. doi:10.1007/s11390-018-1839-y.

[Zam22] Zama. TFHE-rs: A pure rust implementation of

the TFHE scheme for boolean and integer arithmetics over encrypted data, 2022. URL: https://github.com/zama-ai/tfhe-rs.

[Zam24] Zama. Kms whitepaper. https://github.com/zama-ai, 2024. Accessed: 2024-04-23.

[ZEL+23] Guy Zyskind, Yonatan Erez, Tom Langer, Itzik Grossman, and Lior Bondarevsky. Fhe-rollups: Scaling confidential smart contracts on ethereum and beyond, 2023. URL: https://www.fhenix.io/wp-content/uploads/2024/04/FHE_Rollups_Paper_Final-20241404.pdf.