Fhevm

A cross-chain protocol for confidential smart contracts*

Zama, Paris, France

ABSTRACT

Zama's fhevm is a cross-chain protocol that enables confidential smart contracts on any L1 and L2 using Fully Homomorphic Encryption (FHE). It brings end-to-end encryption to onchain applications, guaranteeing confidentiality and privacy while remaining fully composable, verifiable, and permissionless.

Fhevm uses coprocessors to run heavy FHE computations, threshold MPC to secure decryption keys, and a set of smart contracts which can be hosted on a rollup or L1 to orchestrate the different components.

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 both widely distributed and publicly visible. This lack of confidentiality holds even when transactions are partially 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 confidentiality challenge for general-purpose (Turing-complete) blockchains, such as Ethereum, while following four key principles:

- There must be no negative impact on the security of the underlying blockchain.
- The protocol must deliver exact and correct results, with everything being publicly verifiable.
- Developers must be able to write confidential smart contracts using existing languages and tools, such as Solidity, in a way that feels natural.
- Confidential smart contracts must be fully composable with each other, and support data from multiple users without compromising confidentiality.

In this document we introduce the Zama Fhevm protocol that enables confidential smart contracts on top of any blockchain or rollup, without changes to the underlying chain. It combines fully homomorphic encryption (FHE) for the confidential computation, threshold multi-party computation (MPC) protocols for FHE key generation and decryption, and Zero-Knowledge Proofs of Knowledge (ZKPoK) to ensure the correctness and integrity of encrypted inputs. It combines these technologies in a way that is non-intrusive for the blockchain by relying on several components:

- A Coprocessor, which executes the heavy FHE computations and verifies ZKPoKs.
- A Threshold Key Management Service (KMS), which securely generates keys and manages decryptions using MPC protocols.
- A Gateway, in the form of a set of contracts, which acts as an orchestrator.
- A Solidity library and set of contracts, which make it easy for developers to use encrypted data in their contracts, without changing their compilation stack.

Note that while this document focuses on the Ethereum Virtual Machine (EVM), the fhevm protocol can be deployed

*Version 3.1 (June 30, 2025).

on non-EVM ecosystems as well, such as Solana, Cosmos, Ton and others.

1.1 Applications

The fhevm protocol adds a layer of confidentiality to blockchain transactions and state, enabling new applications that have strong data security requirements. It not only secures data in transit, it also secures data in use: It is end-to-end encryption for blockchain. Some example use-cases are given below.

1.1.1 Confidential Token Transfers. 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 transaction data to competitors.

The fhevm protocol solves this by enabling 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 tokens are composable, as one can convert a public 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).

1.1.2 Confidential Swaps. Auto-Market Makers (AMMs) such as Uniswap are a fundamental primitive in decentralized finance (DeFi) which are used to swap hundreds of billions of dollars worth of assets onchain per day. A consequence of the transaction data being public is front-running and the resulting problem of Maximal Extractable Value (MEV). This front running enables network validators to extract value from end users by inserting other (related) transactions around the end-user's transactions. While this is beneficial to some network participants, it is typically detrimental to the end user.

The fhevm protocol solves this by enabling swaps to be done using confidential 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.

1.1.3 Token Launchpads. Launching tokens onchain is currently done either through direct listing on a Decentralized Exchange (DEX), auctions with public bids, or more viral mechanisms such as Pump.fun's bonding curve token launch. While these methods provide various levels of gamification and price discovery, they typically suffer from bidder bias due to the public nature of the process, as well as from bots sniping the tokens by front-running legitimate bidders.

Using the fhevm protocol allows anyone to launch tokens using sealed-bid auctions and order books, thereby opening a new category of gamified, price-discovery optimized launchpads that are also resistant to bots.

1.1.4 Decentralized AI. Artificial intelligence is one of the most revolutionary technology that promises to change everything both onchain and offchain. Running AI onchain however requires confidentiality, as training data, model weights and user prompts must be kept secret at all times, for privacy, IP or legal reasons.

Our fhevm protocol solves this, by enabling AI models to run encrypted end-to-end onchain, on any decentralized infrastructure.

1.1.5 Confidential Voting. Blockchains are a great way to run elections, to manage DAO governance, or to participate in prediction markets, as they allow one to have real-time, tamper-proof vote counts. Without confidentiality however, votes are visible to anyone, leading to a host of issues such as influencing votes, bribery, biased predictions, or social pressure.

Our fhevm protocol solves this by enabling 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 is public, but the individual votes remain confidential.

1.1.6 Decentralized Identities (DIDs). Decentralized identifiers are a novel type of identifier that enables verifiable and self-sovereign digital identities for individuals and organizations.

With fhevm, smart contracts are able 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 can publish the encrypted birth date of a consenting user to a smart contract.

Subsequently, authorized parties can query the smart contract to gain information about the user's age (e.g., whether they have the age of majority) when necessary. The same applies to making KYC/compliance claims when accessing financial products onchain, such as proving eligibility in onchain elections.

1.1.7 Network States. Blockchains have the potential to radically change how countries and network states operate by automating public infrastructure with smart contracts. Everything from identity, currency, elections, company registry, taxes and more could be done onchain. This however requires strong confidentiality guarantees, which the fhevm provides off the shelf.

1.2 Related Work

There are multiple ways to achieve smart contract computations on private inputs. Although some of the solutions listed below can use a combination of 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 fully homomorphic encryption (FHE).

1.2.1 Zero-Knowledge (ZK) Proofs. At their core, ZK proofs tackle the privacy challenge by keeping only committed data and the associated proofs of correct computation onchain. However, the data must be known by the prover in plaintext, in order for the prover to be able compute on it and produce the proof. This means a plaintext copy of the data must be kept somewhere. This can work well when only data from a single party is required for a computation, but raises the issue of what to do for applications requiring data from multiple parties. Examples of the use of ZK proofs are widespread in blockchain, however most applications are for rollups in which the zero-knowledge property is not required (and thus often not enforced). Applications which really utilize the zero-knowledge property are less common, and here we highlight a few:

- In the ZCash [BCG+14] and Monero [Mon23] systems, anonymity is provided to both the sender and receiver of transactions while keeping the amount of exchanged coins shielded using Pedersen commitments [Ped92]. The associated mixer is then proved to have been performed correctly via the use of ZK proofs. Note also that these protocols do not support arbitrary computations.
- The Zexe [BCG+20] and VeriZexe [XCZ+23] systems 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.
- The Hawk [KMS+16] system uses ZK proofs where the inputs to the smart contracts are revealed to a trusted manager that does the computation. TEEs or MPC is suggested as a way to realize the trusted manager without going to details. Note that a custom compiler must be used for contracts, adding to the developer burden.

Using fhevm protocol one could obtain the same outcome, but now the computation happens directly on the encrypted values, and can be done publicly. This means that mixing values from multiple users does not require revealing all of these in plaintext to a trusted party.

However, ZK proofs are unmatched when it comes to the property of public verifiability. When combined with FHE, ZK proofs allow for verifiable, confidential computations on shared state, which can be used to improve our protocol (see Section 5.1 for details).

1.2.2 Trusted Execution Environments (TEEs). Blockchain systems based on TEEs only store encrypted data onchain, and perform computations by decrypting the data inside a secure enclave that holds the decryption keys [CZK+19, KGM19, Oas23, Pha23, SCR23, YXC+18]. 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 [BCC04, CD16].

The enclave approach has been shown many times to be vulnerable to several side-channel attacks [JLLJ+24, KHF+19, LSG+18, vSMK+21, VMW+18, vSY+24, TKK+22]. Such attacks are performed by measuring power consumption, electromagnetic radiation,

measuring timing differences on executions, or measuring memory access patterns. Such side-channel attacks make it particularly challenging for decentralized protocols, where anyone can run a validator or full-node and thus have full access to the enclave (either root access and/or physical access).

However, TEEs are useful as a defense-in-depth solution, in particular where no cryptographic alternative exists. They should be seen as a complementary solution to enhance the security of cryptographic protocols using ZK, MPC or FHE. In our protocol, we support the use of TEEs in order to provide an additional layer of security when handling sensitive material such as secret shares used by the MPC parties.

1.2.3 Multi-Party Computation (MPC). Protocols based on MPC are used extensively in the blockchain space to secure cryptographic keys for signing of transactions, both at the user level and at the exchange level. There are many so-called MPC-wallet providers which enable users to split their cryptographic key between multiple locations, with the key never being reconstituted in order to provide the signing functionality. The signing functionality is then provided via an MPC protocol.

There are a number of other uses of MPC in blockchain, where the MPC protocol is used to replace a single entity which otherwise would need to be trusted with secret data. For example:

- 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 online to participate.
- Eagle [ByCDF23] 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 online all the time, they need to do one round of interaction with the MPC engine to provide inputs [DDN+16].
- gcEVM [Sod] uses garbled circuits-based MPC between the blockchain nodes to compute on encrypted contract state. This makes not only the underlying technology different from ours, but also that fact that we move the heavy computations offchain, and that our solution can be used with existing blockchains without changing their software.

MPC's performance mostly depends on networking IO. As such, it offers good performance for batched confidential computations, where data only needs to be shared with other parties once, but quickly deteriorates when sequential operations need to be performed.

Moreover, the secret shared material in any MPC-based systems is strongly tied to the set of parties for security reasons, and changing this set requires executing a threshold protocol involving all the secret shared material. In systems where the blockchain state is secret shared, this could hence be a significant operation since it can get very large.

On the other hand, in the fhevm protocol, the blockchain state is kept as FHE ciphertexts and only the decryption key is secret-shared. Thus, the blockchain state does not need to be refreshed when the set of parties change, and securely resharing the decryption key to a new set of parties is much easier.

One area where MPC outperforms other solutions however is threshold key management, as examplified by the MPC-wallets mentioned above. By splitting private keys into multiple pieces, it becomes possible to decentralize key management and not have to rely on a single entity. Our protocol makes use of MPC in order to facilitate decentralized key management of the underlying FHE secret keys.

1.2.4 Fully Homomorphic Encryption (FHE). Solutions based on *partially* homomorphic encryption have previously been proposed, but inherently limit the type of operations that can be performed, and therefore only support a certain class of smart contracts and applications. For example:

- Zether [BAZB20] (which was the inspiration for Solana's recent confidential token extension) 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, and does not offer composability with DeFi applications.
- On the other hand, Zkay [SBG+19] explained how to execute Ethereum smart contracts privately using FHE,

but it uses a trusted third party that holds the decryption key; thus creating a single point of trust (or failure).

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

- The PESCA [Dai22] protocol uses a similar setup as ours, making use of a global FHE public key, which everyone uses to encrypt their balances, and a threshold FHE protocol to help decrypt outputs. However, the approach is very different since our aim is to do the FHE computations offchain and allow non-intrusive integration into existing blockchains, with a user and developer experience that is close to that of non-confidential smart contracts.

In general, the main advantages of FHE over other technologies are that it enables computing on the encrypted data directly, making it immune to side-channel attacks when operating over ciphertexts, and does not require sharing data between multiple parties or a single trusted entity. On the downside, FHE does require additional computational power. However, hardware, implementation and algorithmic improvements have enabled FHE performance to improve exponentially in the last few years. For example, a single H100 GPU is already enough to nearly match Ethereum's average throughput, while reaching Solana-level throughput (thousands of transactions per second) will soon be possible thanks to dedicated hardware acceleration (ASICs). Table 1.1 presents the advantages and disadvantages of each approach.

	TEE	MPC	ZK	FHE
Security	limited	high	high	high
Composable	yes	yes	no	yes
Verifiable	yes	no	yes	yes
Performance (CPU)	high	medium	medium	medium
Performance (ASIC)	-	-	high	high

Table 1.1: Comparison of confidential computing technologies.

2 KEY CONCEPTS

This section gives a high-level overview of the key concepts behind the fhevm protocol, with subsequent sections giving more details. Figure 2.1 shows an architectural overview and Table 2.1 presents a summary of trust assumptions.

The fhevm protocol is a cross-chain protocol that enables confidential smart contracts on any blockchain or rollup, without requiring any changes to the underlying chain. It uses a global FHE key under which all inputs and private contract states are encrypted, which allows for composability between contracts and users, and interoperability between chains.

Interacting with the protocol is seamless for developers. They write their confidential smart contracts in Solidity using the fhevm Solidity library described in Section 3, and standard toolchains. The library does a *symbolic* execution of the FHE operations, using *handles* instead of actual ciphertexts, while *coprocessors* run the actual execution of the heavier FHE operations on ciphertexts offchain. This asynchronous, parallel execution model allows for both a high confidential transaction throughput without slowing down the host chain itself.

As for users, they interact with confidential smart contract using their existing wallet, with the encryption and decryption done by interacting with the fhevm *Gateway contracts*. This interaction is typically done in the frontend app, such that encryption and decryption is invisible to end users.

2.1 Host Chains

The fhevm protocol can be deployed on any existing blockchain or rollup, which we simply refer to as *host chain* in this document. There is a small set of *trusted host contracts* deployed on the host chains, that application smart contracts use to interact with the protocol. The use of these smart

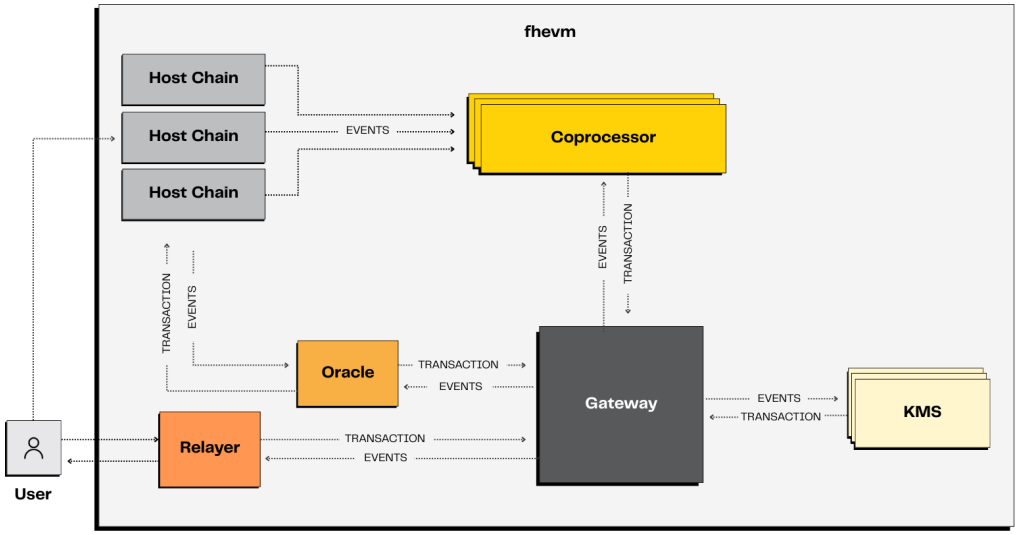

Figure 2.1: Architecture of the fhevm protocol.

Component	Trust assumptions
Coprocessors	At least $\frac{1}{2}$ are honest, but publicly verifiable and slashable.
KMS	At least $\frac{2}{3}$ are honest.
Oracle	None, since all messages are signed.
Relayer	None, since all messages are signed.

Table 2.1: Summary of security assumptions for components.

contracts is typically done behind the scenes by our Solidity library. For instance, we make use of a trusted Access Control List (ACL) contract on the host chain, to allow application smart contracts to define the access control rules for their encrypted state.

2.2 Gateway

The Gateway contracts enable users and applications to validate encrypted inputs, bridge ciphertexts across different host chains, and decrypt ciphertexts. With the help of the coprocessors, the Gateway contracts keep a copy of the ACLs from all connected host chains, allowing them to autonomously make a decision on whether, for instance, a decryption request should be granted. The Gateway contracts are also responsible for enforcing consensus among the coprocessors and orchestrating the KMS nodes.

We make no assumptions about where the Gateway contracts are deployed, including whether on an L1 or rollup, as long as this deployment can be considered trusted[1]. We use *Gateway* to denote any such a deployment of the Gateway contracts.

2.3 Coprocessor

The coprocessor is an offchain component that performs the following tasks by listening to events from the host chains and the Gateway:

1. Verify encrypted inputs from users.
2. Run the actual FHE computations, and store the resulting ciphertexts.
3. Replicate the ACL from the host chains to the Gateway.

The fhevm protocol uses a set of coprocessors, operated by different parties. They each commit their results, including computed ciphertexts, to the Gateway, which in turn ensures that a consensus is reached. Moreover, all tasks performed by the coprocessors are publicly verifiable.

Note that the only place where the delay between symbolic and actual execution is noticeable, is when users or smart contracts ask for a value to be decrypted. However,

[1]The Zama deployment uses an Arbitrum Orbit rollup, where events emitted from the Gateway contracts are independently verified by the operators by having each of them run a validator. The rollup is locked down to prevent arbitrary third-party contracts to be deployed on it.

the coprocessors can be vertically and horizontally scaled to make this gap insignificant.

2.4 Key Management Service (KMS)

The fhevm protocol relies on a key management service (KMS) for generating FHE keys, decrypting ciphertexts, and generating common reference strings (CRS) for the ZKPoKs. The use of the KMS is orchestrated by the Gateway, which enables public verifiability of all requests.

While it is possible to instantiate the KMS in different ways, the one described here uses MPC for distributed key generation and threshold decryption, which ensures that no single party can access the FHE secret keys or decrypt ciphertexts. The KMS will initially be run by 13 highly reputable organizations, with the goal of eventually scale to hundreds of nodes.

2.5 Oracle

Although not part of the fhevm protocol itself, one or more decryption oracles are also typically deployed on each host chain, allowing application smart contracts to request decryptions. The offchain oracle then submits request to the Gateway, and pushes the resulting plaintext back to the application smart contract on the host chain. Since results from the KMS are signed, they can be verified on the host chain to remove trust in the oracle.

Note that while any oracle can be used, Zama provides a hosted oracle that can be used without additional infrastructure.

2.6 Relayer

Similarly to the oracle, one or more relayers are typically deployed, although they are not part of the fhevm protocol itself. These relayers make it easier for users to interact with the Gateway, including requesting decryptions and providing encrypted inputs. As for oracles, since results from the KMS and coprocessors are signed, they can be verified on the host chain to remove trust in the relayer.

Note that while any relayer can be used, Zama provides a hosted relayer that can be used without additional infrastructure.

2.7 Operators

The operators of the fhevm protocol consist of coprocessors and KMS parties. These are responsible for the day-to-day maintenance of the deployment, including adding new operators.

3 SOLIDITY LIBRARY

One of our main design goals for the fhevm protocol is to enable developers to easily build confidential applications, without learning cryptography or changing their existing workflow. For this reason we provide the `FHE` Solidity library[2], which integrates nicely with the rich and mature Solidity ecosystem, including compilers, debuggers, IDEs, and other libraries.

Our library also offers a developer-friendly abstraction of the trusted host contracts, minimizing direct interaction with these.

3.1 Encrypted Types and Operations

The `FHE` library supports various encrypted data types:

- Encrypted booleans: `ebool`.
- Encrypted unsigned integers, from 8 to 256 bits in steps of 8: `euint8`, `euint16`, `euint32`, ..., `euint256`.
- Encrypted signed integers, from 8 to 256 bits in steps of 8: `eint8`, `eint16`, `eint32`, ..., `eint256`.
- Encrypted bytes, from 1 to 32 bytes in steps of 1: `ebytes1`, `ebytes2`, ..., `ebytes32`; and additional larger types: `ebytes64`, `ebytes128`, and `ebytes256`.
- Encrypted addresses: `eaddress`.

Each of these types are implemented as Solidity user-defined value types and, hence, can be used for variables, parameters, and values in mappings and arrays. Note that encrypted values are not actual ciphertexts, but rather *handles* represented by `bytes32` values that point to ciphertexts stored by the coprocessors, see Section 4.5.

Contrary to most FHE solutions that only support additions and multiplications, the fhevm supports all the usual operators for each family of encrypted types. For example:

- `ebool`: logical operators (`and`, `or`, `xor`, `not`).

[2] Available at https://github.com/zama-ai/fhevm.

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

Listing 1: Example computation on encrypted integers `x`, `y`, and `z`.

- `euint` and `eint`: 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`), and bit shifts and rotations (`shl`, `shr`, `rotl`, `rotr`).
- `ebytes`: bitwise logical operators (`and`, `or`, and `xor`, `not`), bit shifts and rotations (`shl`, `shr`, `rotl`, `rotr`) and equality operators (`eq`, `ne`)
- `eaddress`: equality operators (`eq`, `ne`).

As an example of usage, consider Listing 1, which shows how to perform addition and multiplication on encrypted 64-bit values.

3.2 Branching on Encrypted Values

Since comparing encrypted values yields an encrypted Boolean value, it is not possible to use it as part of an "if-else" or "require" statement directly. Instead, our library offers a `select` operation, which allows selecting one of two encrypted values based on an encrypted Boolean[3]. This can for instance be used to emulate branching, by nullifying the effect of operations in the non-taken branch. A simple example of this is shown in Listing 2, where `param` is effectively replaced by zero in case the condition is false. Note that in the example, transactions will always go through, but the underlying plaintext values are unchanged.

3.3 Encrypted Inputs

Smart contracts can directly use handles as inputs when calling functions on other smart contracts. However, a few extra steps must be taken when users want to provide new encrypted values or bridge handles that exist on another host chain. Both of these cases involve first interacting with the Gateway to obtain what we call *external values* together with an *attestation* that can be passed to the smart contract. The external values are essentially handles that must be verified using the attestation before being used, while the attestation takes the form of a list of signatures from the coprocessors. Section 4.7 and Section 4.8 discuss the motivation behind this approach, and give details on how the user obtains the list of values, while we here focus on how they are used.

[3]To be precise, the output value of the `select` operation shares the same underlying plaintext value as one of the two inputs values, but the value is *not* identical to any of them. In fact, it is impossible, even given all associated ciphertexts, to deduce which input was selected.

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

Listing 2: Example of using the select operator to nullify a value if an encrypted condition is false.

```
function myfunction(
    externalEuint64 param1,
    externalEbool param2,
    bytes calldata attestation
){
    euint64 handle1 = FHE.fromExternal(param1,
        attestation);
    ebool handle2 = FHE.fromExternal(param2,
        attestation);
    ...
}
```

Listing 3: Example of converting ciphertext-related input `amountCt` to an encrypted integer handle `amount`.

These values are defined by the `external` types. When receiving these values, the only meaningful thing the contract can do with each external value, is to extract an encrypted value from it by calling the `fromExternal` method. An example of this is shown in Listing 3, where a Solidity function takes an `externalEuint64` and `externalEbool` as input, together with an attestation for them, and extracts an `euint64` and `ebool`, respectively. Calling `fromExternal` performs type checking, and verifies the attestation against the list of known coprocessors, the user address, and the contract address. If successful, it returns an encrypted value that the contract is granted (transient) access to use.

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

    // Make sure the caller can access input
    require(FHE.isSenderAllowed(amount));

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

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

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

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

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

Listing 4: Example confidential token transfer function.

3.4 Pseudo-Random Encrypted Values

Smart contracts can generate pseudo-random encrypted numbers using the `randEuint` and `randEuintBounded` operations. The values are uniform in their domain, and may be used like any other encrypted number. The process is deterministic on the coprocessors, with entropy derived from the secret FHE key.

3.5 Access Control

Deciding which address can access which ciphertext is done programmatically by calling the trusted host contract that maintains an access control list (ACL). This list keeps track of who can compute with a handle, decrypt it, and grant access to it 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.

Interacting with the ACL is done using the following library functions (as illustrated in Listing 4):

- `allow`(`handle`, `address`) grants `address` persistent permission to use `handle`. This is used for example when a contract needs to store the handle in its state and use it for computation in a later transaction, or to enable a user to decrypt or reencrypt the value. It emits an `Allowed` event to the coprocessors.

- `allowTransient`(`handle`, `address`) allows `address` to access `handle` for the duration of the transaction only. This is used for example when a contract wants to call another contract and pass it a handle as argument. It is also used internally by the `FHE` library to grant access to results when a contract calls operators such as `add`, `asEuint64` and others.

- `allowForDecryption`(`handle`) persistently marks a handle as publicly decryptable by anyone. It emits an `AllowedForDecryption` event to the coprocessors.

- `isAllowed`(`handle`, `address`) returns true if `address` is allowed either persistent or temporary access to `handle`.

- `isSenderAllowed`(`handle`) is a shorthand that returns true if the `msg.sender` is allowed to access `handle`. It is equivalent to calling `isAllowed`(`handle`, `msg.sender`), and should systematically be used by callee contracts to check that caller contracts are actually allowed to access any handles passed as arguments.

If a contract calls `allow` or `allowTransient` without itself having access to the handle, then the trusted host contract reverts the transaction. Besides maintaining the ACL, the trusted host contract also emits events that trigger the coprocessors to update the copy of the ACL kept by the Gateway contracts. See Section 4.9 for more details.

3.6 Public Decryption

The private decryption key it not stored anywhere on the host chain, but is instead managed by the KMS via the Gateway. As such, decryption operations in smart contracts are an asynchronous process that uses oracles to request a decryption from the Gateway and put the result back on-chain. The results are signed by the KMS and can be verified on the

host chain, so no additional trust assumptions are needed for this process.

The fhevm library supports invoking a decryption oracle through the `decrypt` interface function, which accepts a list of handles and a callback selector. This call generates a unique request ID and sends it to an on-chain oracle contract which implements the `IDecryptionOracle` interface. The library is agnostic to the specific oracle used, and developers only need to specify the oracle contract address using `setDecryptionOracle`. To ensure the integrity of the response, the callback function can verify the authenticity of the plaintext using `checkDecryptionSignatures`, which validates the signatures to confirm that the data came from the KMS.

An example of what the use of a decryption oracle may look like is shown in Listing 5, with further details given in Section 4.10. Note that the decryption process supports batch decryption of a list of handles for efficiency reasons.

```
function constructor() {
    FHE.setDecryptionOracle(0x9223...);
}

function requestUint8() public {
    bytes32[] memory cts = new bytes32[](2);
    cts[0] = FHE.toBytes32(encryptedUint8);
    cts[1] = FHE.toBytes32(secondEncryptedUint8);
    FHE.requestDecryption(
        cts,
        this.callbackUint8.selector);
}

function callbackUint8(
    uint256 requestID,
    uint8 decryptedInput,
    uint8 decryptedInput2,
    bytes[] memory signatures
) public {
    // Verify KMS signatures
    FHE.checkSignatures(requestID, signatures);

    // ... use the decrypted values ...
}
```

Listing 5: Example decryption of an encrypted 64 bit value.

3.7 User Decryption

Some applications only need to display decrypted values to users, without putting the result back on-chain (e.g. displaying encrypted token balances in wallets). While users could directly send decryption requests to the Gateway, in practice it is the application running in, for instance their browser, that will do so on their behalf, by asking them to sign the request with their wallet.

To avoid requiring a user to approve a signature every time an application wants to for instance display their account balance, we introduce the notion of an *authentication token* that can be used by the application to decrypt *any* handle that both the user *and* the application has access to. More specifically, an application that is represented on the host chain by a contract address a, asks the user with address u, to sign an EIP712 object that embeds both u and a (but no handles). This way, by inspecting the EIP712 object during signing, the user limits the scope of the token to handles that belong to the application, and prevents it from decrypting any of their handles belonging to other applications. To accommodate applications that consist of multiple contracts, in reality the token embeds a list of contract addresses.

Furthermore, for some values, users will typically want to ensure that the decrypted value can only be read by them, and not the general public. For this reason, a (classical, non-FHE) public key can be embedded in the token, and the KMS will encrypt the result under this key before sending it back, ensuring no-one else can see it, not even the Gateway. More concretely, each KMS party will encrypt its *share* of the decrypted value under the public key, and as such not even the KMS parties will learn anything.

A convenient client-server Javascript library[4] is provided for this, that handles the generation of the user's public key, the signature from the user, the interaction with the Gateway contracts, and the decryption of the returned result using the user's private key. An example use of the library is shown in Listing 6 and more details about the process are given in Section 4.11.

4 IMPLEMENTATION DETAILS

This section details the implementation of the fhevm protocol.

4.1 TFHE Scheme

While many FHE schemes have been proposed over time, a few in particular stood out and are in the process of being standardized by the ISO: BGV [BGV12], BFV [Bra12, FV12], CKKS [CKKS17] and TFHE [CGGI20, CLOT21, Joy22]. The schemes BGV, BFV and CKKS have the ad-

[4] Available at https://github.com/zama-ai/fhevm.

```
// 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);

// Request the Gateway to decrypt the value and
// encrypt the result under the reencryption key
const userBalance = instance.userDecrypt(
    encryptedBalance,
    privateKey,
    publicKey,
    signature,
    CONTRACT_ADDRESS,
    USER_ADDRESS
);
```

Listing 6: Example reencryption of an encrypted ERC20 balance.

vantage 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. These schemes are useful for applications such as machine learning, where the computation depth is bounded and the model can absorb approximation errors. However, they 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, imply that checking whether a user has sufficient funds may sometimes succeed even when it should not).

The TFHE scheme [CGGI20, CLOT21, Joy22] on the other hand, enables exact and unbounded computation through the use of an operator called a *programmable bootstrapping (PBS)*. In a nutshell, the PBS operator evaluates a table lookup homomorphically, while resetting the ciphertext noise to a nominal level. This allows for the homomorphic evaluation of exact, unlimited and arbitrary 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 library [Zam22, Zam25], which currently offers the most complete feature set and the fastest performance on both CPU and GPU. Encryption, evaluation and ZKPoKs are all made available off the shelf via this library. We use a parameter set that is cryptographically secure, based on the latest attack vectors and security standards, while being compatible with the threshold key generation and decryption done by the KMS. Benchmarks[5] per operation are shown in Table 4.1, where each operand is a ciphertext, except for division where the divisor is a plaintext, and shift/rotations where the amount to shift/rotate is also a plaintext. Timings are computed using one AMD EPYC 9R14 server with 192 cores for CPU, and two NVIDIA H100s for GPU.

4.2 Key Management Service

The key management service (KMS) is managed by the Gateway, and employs a set of n distinct MPC parties[6] that hold secret shares of the FHE decryption keys and execute the MPC protocols. The Gateway provide the public interface of the KMS, through which all KMS operations are launched, including key generation and decryption. The parties on the other hand are not publicly reachable, but instead run a connector that receives events from the Gateway, and signs and submits results back to them, and one or more cores that perform the actual MPC execution by interacting with cores of the other parties. See Figure 4.1 for a high level overview.

When new public material, such as encryption keys, is generated by the cores, then each core persists this in the party's public database, which ensures availability even if some parties disappear in the future. The cores of the different parties communicate with each other over secure authenticated channels when executing the MPC protocols, in particular gRPC over mTLS.

The combined use of MPC and smart contracts allows the

[5] See also https://docs.zama.ai/tfhe-rs/get-started/benchmarks for the most recent numbers.

[6] Zama's deployment uses $n = 13$.

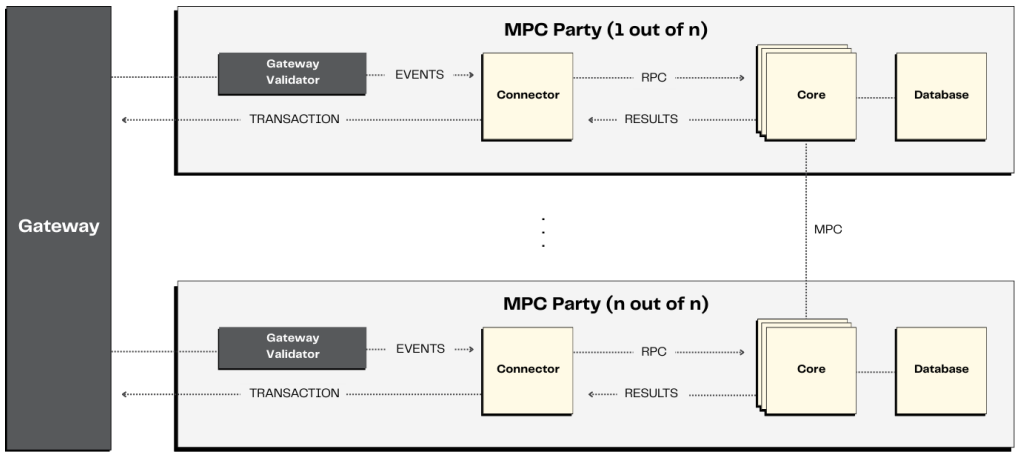

Figure 4.1: Architecture of the KMS.

Operation	`euint8`	`euint16`	`euint32`	`euint64`
		CPU (ms)		
Add / Sub	54	60	82	109
Mul	98	141	213	400
Div*	139	202	280	456
Eq / Neq	36	56	57	81
Lt / Gt	37	55	77	99
Min / Max	77	98	121	148
Shift* / Rot*	20	20	21	23
And / Or	19	20	21	23
Select	30	32	33	36
		GPU (ms)		
Add / Sub	10	10	14	20
Mul	20	28	60	166
Div*	23	36	68	171
Eq / Neq	7	10	11	15
Lt / Gt	10	13	17	22
Min / Max	16	20	25	33
Shift* / Rot*	3	3	4	5
And / Or	3	3	4	4
select	6	7	8	11

* by a plaintext

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

system to offer:

- *Decentralization.* No secret material is ever stored in a central location, and it remains secure as long as $t < \frac{n}{3}$ parties are not colluding. With our choice of $n = 13$, this means that the KMS can tolerate collusions of size up to and including $t = 4$.
- *Resilience to outages.* Only a subset of the MPC nodes are required to be online to ensure continuous operation, because of the fact that we use robust MPC protocols designed with *guaranteed output delivery*.
- *An incorruptable and public audit log.* All operations and results are recorded as blockchain transactions.

We note that it is important for security that each party can independently validate correct execution of the Gateway contracts, because a malicious Gateway that issues faulty events to the connectors could not only decrypt ciphertexts that should not be decrypted, but might even extract the secret FHE key through *selective failure attacks*, where failure or not when decrypting a maliciously constructed ciphertext depends on the bits of the secret key [Sma23].

Finally, we leverage enclaves to enhance security by making it very challenging for corrupt parties to even learn their own share of the secret keys. This is important since parties might otherwise be incentivized to, for instance, sell their share, thereby potentially aiding a buyer in obtaining enough shares to reconstruct a secret key. Since these are shares of a decryption key (and not, for instance, a signature key),

this type of collusion is hard to detect, and hence we aim to prevent it from even happening through the use of enclaves[7].

4.2.1 MPC protocols. The MPC protocols are designed to ensure completion of execution even if some of the parties are offline or misbehaving. That is, a *robust*, *maliciously* secure MPC protocol with *guaranteed output delivery*. This in turn implies that the protocols require a strong *honest majority* with a corruption threshold $t < \frac{n}{3}$. The protocols are *information theoretically* secure, although in practice we optimize the overall protocol by leveraging pseudorandom functions and random oracles. For more details on our open source[8] protocols we refer the reader to our research papers [DDE+23, BCD+25].

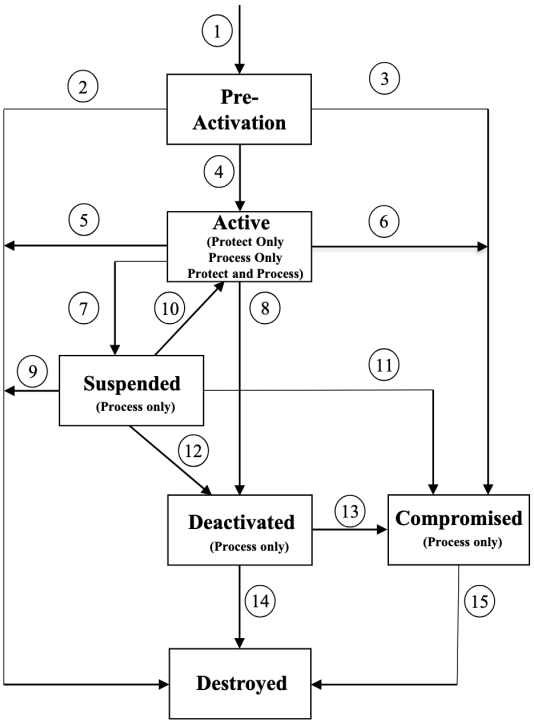

Figure 4.2: NIST key lifecycle states [Bar20].

4.2.2 Key lifecycles. The Gateway works with a single *conceptual key* that is used by all connected host chains. Even so, things are not as simple as just constructing a single FHE key once, since the key needs to be rotated at certain intervals for security, and the shares of it must likewise occasionally be refreshed, including if changing the set of KMS parties. To keep things simple, we here focus on key rotation, and note that each conceptual key in fact hence consists of multiple *concrete keys*.

To handle this, the KMS follows the NIST standard [Bar20] which describes 6 states that a concrete key can be in, and the permitted state changes as shown in Fig. 4.2. More specifically, the states and possible actions are as follows:

Pre-activation: A newly generated concrete key will start in this state.

Active: The state which indicates that a concrete key is the currently active one. Only one concrete key can be in this state. All operations are allowed in this state.

Suspended: A concrete key moves from Active into Suspended when it gets replaced by another concrete key becoming Active. While a concrete key is Suspended, it can still be used for providing inputs, FHE computation, and decryptions for a limited time, but no key rotation from it can be launched, nor is it possible to change the state of keys dependent on a Suspended key. Similarly to the Active state, we only allow one concrete key to be Suspended at any point in time. This is a very temporary state that is only used to ensure no disruption in operations.

Deactivated: Once a concrete key is Deactivated it cannot become Active again, but it will still exist in the system. Only decryption is allowed. It should be considered the typical final state for a key.

Compromised: If a concrete key has been misused it will end up in this state and cannot be Active again. Only decryption is still allowed.

Destroyed: If a concrete key is Deactivated or Compromised, then it may be Destroyed at a certain point in time, meaning that only meta data about it remains while its sensitive data is deleted. That is, decryption may never be possible again, even with updates of the system logic.

[7] In Zama's deployment we use AWS Nitro enclaves, in conjunction with the AWS KMS and policies that only allow sensitive material to be accessed from inside an enclave.

[8] Available at https://github.com/zama-ai/threshold-fhe.

Since FHE allows the generation of key switching keys (KSKs), it is possible to continue to use ciphertexts encrypted under a retired concrete key in composition with newer concrete keys for the same logical key. More specifically, a KSK can convert a ciphertext encrypted under one key to a ciphertext encrypted under another key. For this reason, we specify the following logic of the KMS when it comes to lifecycles.

- The concrete key that is being key switched to, must be Active or Suspended.
- The concrete key that is being key switched from must not be Pre-activated, Compromised or Destroyed.

4.2.3 Backup. The MPC protocols automatically affords resilience to data loss, by allowing $n - t$ honest parties to still be able to use a given secret shared key in case t parties lose their key shares. The parties can then fully recover by doing a key refresh or key rotation.

However, in case more than t parties lose their data at the same time, the MPC protocols offer no recovery on their own. While this might seem like a contrived situation, it could for example happen due to a bad software update or a sophisticated attack, since the keys shares will necessarily always have to be used by an online system. This would be devastating since all encrypted state is effectively then lost.

For this reason, the KMS also offers an offline custodial backup solution. The MPC parties do not need to interact with the custodians whenever they generate an FHE key, and interaction is only required during an initial setup step and recovery. The solution ensures that a quorum of custodians must agree to execute a recovery procedure (by order of the operators). The quorum threshold can be chosen arbitrarily i.e. it does *not* need to be less than $\frac{m}{3}$ for m custodians, like the threshold for the MPC nodes. Even if the custodians are malicious, they will *not* be able to recover any private keys unless at least the quorum of them collude. Furthermore the procedure allows to recover towards a different (yet equally sized) set of MPC parties.

The backup procedure can briefly be described as follows. During a setup phase, each custodian constructs a *backup key pair* for (classical) asymmetric encryption and signing, and posts the public part to the Gateway. Then, for every FHE key generation, each party takes their share of the secret FHE key, and shares it further into m shares; one for each custodian. They encrypt each of these shares under the backup key of the appropriate custodian, and post the encrypted shares to the Gateway. If the FHE key shares ever need to be recovered, each member of the set of MPC parties that is being recovered to, generate a *recovery key pair* for (classical) asymmetric encryption and disclose the public parts to the custodians. The custodians in turn decrypt the shares they can for each of the MPC parties, and signcrypt them under the recovery keys. These are returned to the MPC parties who can reconstruct the original share as long as they receive a quorum of the custodians' responses.

4.3 Gateway contracts

We next outline the different Gateway contracts which administers the fhevm protocol. These contracts are activated in two ways: either by an oracle or relayer following an event emitted on the host chain (as is the case for decryption), or directly by the operators and token holders (as is the case for key generation).

`GatewayConfig:` Manages everything in relation to the setup, meta data, and addresses of operators participating in a deployment of the fhevm protocol, including KMS parties, coprocessors, their public storage locations, and registered host chains. In other words, it takes care of basically any system administration.

`MultichainACL:` Manages the copy of ACLs from host chains, keeping tracks of all ciphertext handles and the addresses that are allowed to use and decrypt them. It is updated by the coprocessors.

`CiphertextCommits` Stores a mapping between handles and digests of the ciphertexts. It is populated by the coprocessors.

`Bridging:` Manages the bridging process of moving handles from one host chain to another, in conjunction with the coprocessors.

`KeyManagement:` Manages everything in relation to key states and operations. It emits events to the KMS connectors.

`InputVerification:` Manages the process of ensuring that encrypted inputs are correctly constructed and known

by the user providing it. The actual proof is verified by the coprocessors.

`Decryption:` Facilitates the decryption procedure, i.e. ensuring that decryption requests are valid and then emitting events to the KMS connectors.

4.4 Coprocessor

Continuing the description from Section 2.3, the coprocessor is an off-chain component with the architecture shown in Figure 4.3. Specifically, by running a full node for each host chain, it receives events to perform computations and to propagate ACL updates. The events are added to a database acting as a task queue, from where one or more workers pick up the tasks once they are ready to be executed. Once processed, results are added back to the database, including all computed ciphertexts. In some tasks the results are finally forwarded to the Gateway, where a consensus is enforced among the coprocessors.

Each coprocessor also maintains a public storage used for ciphertexts[9], with a location registered by the Gateway. This is used for anyone to download ciphertexts as needed, including the KMS.

4.5 Handles and Symbolic FHE Execution

As illustrated in Listing 7, when smart contracts call an FHE operation on the host chain, the fhevm Solidity library together with the trusted host contract do not execute the actual FHE computation. Instead, a *symbolic* execution takes place, taking handles represented as `bytes32` values as inputs, producing new handles as output, and emitting an event containing all handles to notify the coprocessors to perform the actual execution on the associated ciphertexts that they have stored in their database. The output handles are computed by hashing together the input handles, the operation type, and the result type, which allows the host chain to maintain high throughput, since evaluating the hash function is very fast and consumes little gas. Note also that this means that handles are unique, deterministic, and state independent.

To avoid having too much lag between the symbolic execution on the host chain and the actual execution on the coprocessor, the trusted host contract also implements a maximum "FHE gas" per transaction. This limits the amount of FHE computation that can be done in a transaction, and in turn guarantees a maximum latency between the time an encrypted value is used in a transaction on the host chain, and the time that the ciphertexts associated with results depending on the encrypted value, can potentially be decrypted.

[9] The Zama deployment uses an AWS S3 bucket.

```
enum Operators {
  fheAdd
  ...
}

type euint64 is bytes32;
type ...

library FHE {
  function add(
    euint64 x,
    euint64 y
  ) public returns (euint64) {

    // Produce output handle
    euint64 output = euint64(
      keccak256(abi.encodePacked(fheAdd, x, y, ...))
    );

    // Emit event to notify coprocessors
    emit FheOpEvent(
      address(this),
      fheAdd,
      x,
      y,
      output
    );

     return output
  }

  ...
}
```

Listing 7: Example of symbolic execution.

4.6 Actual FHE Execution

As explained in Section 4.5, on the smart contract side, all operations are performed symbolically using handles instead of actual ciphertexts, and events are emitted for the coprocessors to do the actual computation. By running a full node for the host chain, each coprocessor receives these events and adds a corresponding task to its work queue. When all input ciphertexts for a given task are ready, a coprocessor worker pulls the task from the queue, reads the ciphertexts from the coprocessor database associated with the input handles, performs the actual FHE computation using TFHE-rs, and writes the resulting ciphertext back to the database under the output handle. While simple, this algorithm allows parallel and horizontally scaled execution of FHE operations,

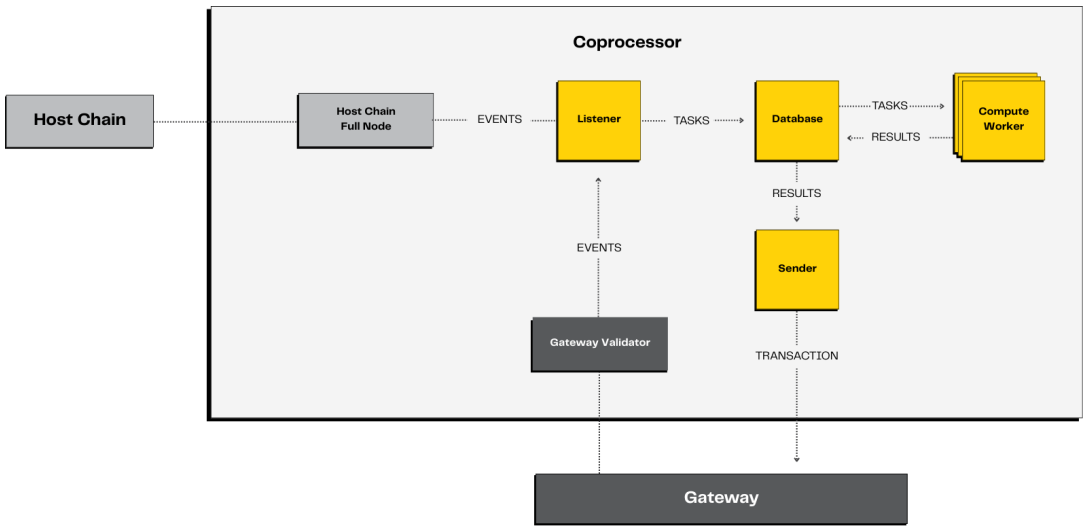

Figure 4.3: Architecture of the coprocessor.

yielding far better throughput than if they were executed sequentially at transaction time.

A downside to this approach, is that the host chain only reaches consensus on the symbolic computation, and not the actual ciphertexts. This means that a coprocessor could potentially deviate and produce faulty ciphertexts either by mistake or maliciously. They could be motivated to do so for instance to save on compute, to decrypt a target ciphertext by substituting it for a decryptable result, or potentially even extract the secret key through selective failure attacks. This is mitigated by having the coprocessors make a public commitment to ciphertexts as part of processing `Allowed` events. Concretely, as explained in Section 4.9, they here send transactions to the Gateway with both the handle and the digest of the associated ciphertext. This makes the coprocessors publicly verifiable, since anyone can recompute the FHE operations and compare digests.

4.7 Inputting Encrypted Values

For security reasons, when a user wants to encrypt a plaintext value and use it as an encrypted value on a host chain, they need to provide both the ciphertext and a Zero-Knowledge Proof-of-Knowledge (ZKPoK) proving that they know the corresponding plaintext and that the ciphertext is well formed. They also need to specify the intended user address u and receiving contract address c on the host chain, which, together with the host chain id, are embedded into the ZKPoK as auxiliary information. To save space, multiple plaintext values are packed into a single 16-20 kB ciphertext, which can hold up to 2048 bits of plaintext data. The packed ciphertext contains type information about each value packed into it. Moreover, only one ZKPoK is required for the packed inputs, saving both on size and proving/verification time.

It is the coprocessors who verify the proof, and in return provide an attestation in the form of a signature that can be verified on the host chain. This process is orchestrated by the Gateway as follows:

1. User encrypts their plaintext values and creates a ZKPoK with the auxiliary information.
2. The packed ciphertext, ZKPoK, and auxiliary information are sent to the Gateway. This emits a `VerifyProofRequest` event that triggers all coprocessors.
3. Each coprocessor verifies the ZKPoK and then does the following for each value:
   (a) Extracts an unpacked ciphertext for the value.
   (b) Deterministically derives a handle for the unpacked ciphertext by applying a hash function.
   (c) Stores the unpacked ciphertext in its database under the handle.

4. Each coprocessor then signs its list of handles, with u and c embedded into the signature, and sends the list and the signature back to the Gateway. It check that the signature is valid and that it is comes from a registered coprocessor.

5. When the Gateway has received identical handles from a majority of the coprocessors, an event is emitted to the user with the list of handles and the list of signatures.

On the user side, each handle is treated as an `external` value and the list of signatures is treated as an attestation. These may now be sent as inputs on the host chain. Note that the intended user and contract address are not checked by the coprocessors; they are simply embedded into the signatures. Checking these is done on the host chain when the actual user and contract address is known. See Section 3.3 for further details.

In TFHE-rs, ZKPoKs are implemented following the scheme from [Lib24], which enables, simultaneously, both a proof of plaintext knowledge and a proof of correctness of the TFHE public key encryption. Benchmarks are shown in Table 4.2, 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.

Proving (browser)	Verification (server)	Proof Size
1.3 s	130 ms	1.8 kB

Table 4.2: Timings and sizes of ZKPoKs for 10 euint64.

Type	Message (bits)	Compressed Size (kB)
`ebool`	1	1.8
`euintX` / `eintX`	8 - 256	1.8
`eaddress`	160	1.8
`ebytes64`	512	2.2
`ebytes128`	1024	4.4
`ebytes256`	2048	8.8

Table 4.3: Sizes for various FHE data types.

4.8 Bridging Encrypted Values

For security reasons, a handle on one host chain is not immediately usable on another host chain. In particular, it first needs to be certified that the user had access to the handle on the source host chain in order to grant access on the destination host chain.

The process for this is similar to the one for inputting encrypted values (see Section 4.7) and is likewise orchestrated by the Gateway. We here assume that the same Gateway is used for both host chains. The process is then as follows:

1. User sends a list of handles from the source host chain to the Gateway, along with an intended user address u and contract address c on the destination host chain.

2. The Gateway verifies that the user is allowed access to each handle on the source host chain using its copy of the source chain's ACL. If successful, then it emits an event that triggers the coprocessors.

3. Each coprocessor then deterministically derives a new handle for the destination host chain from each handle in the list.

4. Each coprocessor then signs its list of handles, with u and c embedded into the signature, and sends the list and the signature back to the Gateway. These check that the signature is valid and that it is comes from a registered coprocessor.

5. When identical handles has been received from a majority of the coprocessors, an event is emitted to the user with the list of handles and the list of signatures.

Like for inputting encrypted values, on the user side, each handle is treated as an `external` value and the list of signatures is treated as an attestation, which can be sent as inputs on the host chain. See Section 3.3 for further details.

4.9 Allowed and AllowedForDecryption Events

When a handle is allowed on a host chain, the trusted host contract not only updates its own ACL, it also emits an `Allowed` or `AllowedForDecryption` event with the handle h, the caller contract address c, and the address a being allowed. Each coprocessor receives this event and does the following:

- Prepares the ciphertext associated with h for potential decryption, and stores both the original ciphertext and the prepared ciphertext it in its public storage under their digests d_{reg} and d_{sns}, respectively. It then sends h, d_{reg}, and d_{sns} to the Gateway as a commitment.

- Sends h, c, and a to the Gateway to update its copy of the host chain's ACL.

The Gateway in both cases waits until a majority consensus has been reached on the values, and then update state accordingly. In the case of commitments to ciphertext, the Gateway simply stores the mapping from h to d_{reg} and d_{sns}. In the case of ACL updates, the Gateway validates that c is allowed access to h, and if so, also allows a access to h. Note that both of these updates must happen before a decryption request can succeed, but their ordering is not important.

Ciphertexts are prepared by performing a *Switch-n-Squash* operation[10], which is a deterministic process that uses TFHE-rs to convert the ciphertexts into a larger plaintext domain, where there is a large gap between the message bits and the error bits. This gap is needed for security reasons by the KMS when performing a decryption, as detailed in Section 4.10 and Section 4.11.

The commitments to ciphertexts are used by the KMS connectors to verify that they have received the correct ciphertexts, when they download them during for example decryption requests. Furthermore, the commitments are used to ensure public verifiability of the coprocessors.

4.10 Public Decryption

As discussed in Section 3.6, smart contracts can request decryption of encrypted values via oracles, that pass a list of ciphertext handles hs, calling contract address a, and callback function f to the Gateway. This contract in turn validates a against the ACL on the Gateway, and if successful emits an event containing hs, a, and f to trigger the KMS connectors. Each of the connectors then does the following:

1. Downloads the prepared ciphertexts associated with handles in hs from a coprocessor; the location to download from can be fetched from the Gateway if not known already.

2. Checks that the digests of the downloaded ciphertexts matches the majority consensus reached by coprocessors when they committed to them.

3. Calls one of its associated cores to engage in the MPC public decryption protocol to decryption the ciphertexts.

4. Sends back a transaction to the Gateway when results have been obtained from the core, with either the plaintexts and a signature, or the error encountered.

On the Gateway side, once at least $t + 1$ results with valid signatures from registered parties have been sent back, an event with the plaintexts and list of signatures is emitted. This triggers the oracle to invoke $a.f(m)$ on the host chain. If desired, the receiving smart contract on the host chain can verify the signatures on the plaintext value generated by the KMS parties. This means that the oracle can be untrusted since it can at best decide to not fulfill the request, in which case another oracle can take over, and the contract can be sure that only legitimate values are put back on the host.

4.11 User Decryption

Users who want to decrypt the ciphertexts associated with a list of handles, without revealing the plaintexts to the KMS or any other observer, can provide a (classical, non-FHE) public key under which the KMS must re-encrypt the plaintexts. The process is similar to public decryptions, but it is instead the user who sends the decryption request to the Gateway either directly or through a relayer. Concretely, the following items are sent in a transaction to the Gateway:

- A (classical, non-FHE) public encryption key under which the results should be encrypted.

- The user address u.

- A list of handles hs. It is checked, for each handle h in hs, that it is allowed for u.

- A list of addresses as of contracts. It is checked, for each handle h in hs, that it is allowed for at least one address in as.

- An EIP-712 signed structure containing everything except the handles.

[10] See e.g. Section 3 of [DDE+23] for details.

After performing the checks mentioned above against the ACL on the Gateway, an event is emitted that triggers the KMS connectors, and in turn the cores to execute the MPC user decryption protocol. This protocol is similar to public decryptions, with the difference that each core instead sends back their *secret share of the plaintexts* encrypted under the public key, instead of the plaintexts themselves. This means that even the cores do not learn the plaintexts.

Like in public decryptions, each core also signs the results before sending to the Gateway. However, the consensus enforced by the Gateway here does not include the plaintexts, since the encrypted secret shares coming from the parties are not identical. Instead it simply verifies the signatures and waits until it has obtain at least $2t+1$ encrypted shares. Finally, the user decrypts the encrypted shares, and performs reconstruction on the shares to obtain the plaintexts.

5 FUTURE WORK

Here we list some of our objectives for improving our protocol.

5.1 Security

We aim for full public verifiability of the fhevm protocol. The only place where this is currently not the case is for the KMS, where a threshold assumption is needed to not only ensure secrecy of the private FHE keys, but also to guarantee correct results. We believe that the combined use of secret sharing and enclaves justifies this assumption, yet making the KMS public verifiable would further improve the guarantee of obtaining correct results.

Another objective is to improve the security around coprocessors. Currently, they can be incentivized to behave honestly through consensus and public verifiability, but the use of *verifiable FHE* (e.g., using ZK) would remove the risk of ciphertext substitution and selective failure attacks. Making verifiable FHE efficient is an ongoing work.

We are also investigating the efficiency of making the protocol more decentralized, both in terms of coprocessors and KMS nodes. With efficient verifiable FHE, anyone could submit computations and be rewarded, in a proof-of-work fashion. For the KMS we are considering both the option of letting a larger set of nodes participate in for instance decryptions, and the option of keeping a smaller set of nodes but rotating them among a larger set. This is known in the literature as committee-based MPC.

Finally, our goal is to make every component of our protocol post-quantum secure. While this is already the case for our FHE scheme and MPC protocols, we would also need the underlying blockchains (and their signature schemes) to be post-quantum secure. Likewise, we are currently working on a post-quantum version of our ZKPoK scheme used to verify inputs (note that the privacy of existing proofs would not be affected by a future quantum computer).

5.2 Performance

We are actively working on improving the performance of FHE through research and specialized hardware[11]. And likewise, we are actively working on making our current MPC protocols faster, as well as improving the relationship between security and performance; as one example, we are interested in having weaker robustness guarantees in exchange for either a higher security threshold or faster execution time.

With respect to the fhevm protocol itself, we are working on ways to avoid the infinite state growth that is currently caused by the ACL being append-only. We are also looking at ways to more efficiently support multiple host chains, by for instance having another layer of Gateways, with one Gateway for each host chain that handles most operations, yet all rollup to a single "Super-Gateway".

Beyond the above, we are also working on engineering optimizations. This includes making the system horizontally scalable and supporting larger precompiled computations in the coprocessor instead of only atomic operations.

5.3 Features

We are continuously adding new features to the Solidity library, such as operations and data types, and omit these here, as it will be an ongoing effort to ensure a smooth and rich developer experience.

In addition to that, we are also considering adding more fundamental features. This includes extending the KMS to support MPC threshold signatures.

[11] There are several attempts at creating FHE accelerators using FPGA [DVV23] and ASIC [Cor, Fab, Opt], with promising results.

5.4 Applicability

Finally, we are actively working on making it easy for anyone to add the fhevm protocol to their host chain, including using their own implementation of some of the components (e.g. Oracle and Relayer). We currently support EVM-based host chains, but nothing in the protocol depends on this, and it should be relatively straightforward to adapt it to non-EVM host chains.

REFERENCES

[Bar20] Elaine Barker. NIST SP 800-57 Part 1 Rev. 5 - Recommendation for Key Management: Part 1 – General. National Institute of Standards and Technology, May 2020. URL: https://csrc.nist.gov/pubs/sp/800/57/pt1/r4/final.

[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. Springer, Cham, February 2020. doi:10.1007/978-3-030-51280-4_23.

[BCC04] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick McDaniel, editors, *ACM CCS 2004: 11th Conference on Computer and Communications Security*, pages 132–145. ACM Press, October 2004. doi:10.1145/1030083.1030103.

[BCD+25] Carl Bootland, Kelong Cong, Daniel Demmler, Tore Kasper Frederiksen, Benoit Libert, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Samuel Tap, and Michael Walter. Threshold (fully) homomorphic encryption. Cryptology ePrint Archive, Paper 2025/699, 2025. URL: https://eprint.iacr.org/2025/699.

[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. IEEE Computer Society Press, May 2014. 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. IEEE Computer Society Press, May 2020. 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. IEEE Computer Society, September 2021. Extended version available as Cryptology ePrint Archive, Report 2021/501 at https://ia.cr/2021/501. doi:10.1109/BRAINS52497.2021.9569822.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325. Association for Computing Machinery, January 2012. doi:10.1145/2090236.2090262.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, Berlin, Heidelberg, August 2012. doi:10.1007/978-3-642-32009-5_50.

[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. IEEE Computer Society, August 2021. 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.

[ByCDF23] 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, Part I*, volume 13950 of *Lecture Notes in Computer Science*, pages 270–288. Springer, Cham, May 2023. doi:10.1007/978-3-031-47754-6_16.

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

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020. doi:10.1007/s00145-019-09319-x.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, Cham, December 2017. doi:10.1007/978-3-319-70694-8_15.

[CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 670–699. Springer, Cham, December 2021. doi:10.1007/978-3-030-92078-4_23.

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

[CZK+19] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. 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*, pages 185–200. IEEE Computer Society Press, June 2019. doi:10.1109/EuroSP.2019.00023.

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

[DDE+23] Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Samuel Tap, and Michael Walter. Noah's ark: Efficient threshold-FHE using noise flooding. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 35–46. ACM Press, November 2023. doi:10.1145/3605759.3625259.

[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. Springer, Berlin, Heidelberg, February 2016. doi:10.1007/978-3-662-54970-4_10.

[DVV23] Jan-Pieter D'Anvers, Michiel Van Beirendonck, and Ingrid Verbauwhede. Revisiting higher-order masked comparison for lattice-based cryptography: Algorithms and bit-sliced implementations. *IEEE Transactions on Computers*, 72(2):321–332, 2023. doi:10.1109/TC.2022.3197074.

[Fab] Fabric Cryptography. Introducing the VPU, an FHE and ZK accelerator. URL: https://www.fabriccryptography.com/.

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

[JLLJ+24] Nerla Jean-Louis, Yunqi Li, Yan Ji, Harjasleen Malvai, Thomas Yurek, Sylvain Bellemare, and Andrew Miller. SGXonerate:finding (and partially fixing) privacy flaws in TEE-based smart contract platforms without breaking the TEE. *Proceedings on Privacy Enhancing Technologies*, 2024(1):617–634, January 2024. doi:10.56553/popets-2024-0035.

[Joy22] Marc Joye. SoK: Fully homomorphic encryption over the [discretized] torus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):661–692, 2022. doi:10.46586/tches.v2022.i4.661-692.

[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*. The Internet Society, February 2019. 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. IEEE Computer Society Press, May 2019. 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. IEEE Computer Society Press, May 2016. 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, *PKC 2024: 27th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 14602 of *Lecture Notes in Computer Science*, pages 36–67. Springer, Cham, April 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. USENIX Association, August 2018.

[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/.

[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. Springer, Berlin, Heidelberg, August 1992. 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. ACM Press, November 2019. 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/.

[Sma23] Nigel P. Smart. Practical and efficient FHE-based MPC. In Elizabeth A. Quaglia, editor, *19th IMA International Conference on Cryptography and Coding*, volume 14421 of *Lecture Notes in Computer Science*, pages 263–283. Springer, Cham, December 2023. doi:10.1007/978-3-031-47818-5_14.

[Sod] SodaLabs. gcEVM v0.1. URL: https://www.sodalabs.xyz/.

[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*, pages 309–331. IEEE Computer Society Press, July 2023. 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. IEEE Computer Society Press, May 2022. doi:10.1109/SP46214.2022.9833802.

[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. USENIX Association, August 2018.

[vSMK+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. IEEE Computer Society Press, May 2021. doi:10.1109/SP40001.2021.00064.

[vSY+24] Stephan van Schaik, Alexander Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Daniel Genkin, Andrew Miller, Eyal Ronen, Yuval Yarom, and Christina Garman. SoK: SGX.fail: How stuff gets eXposed. In *2024 IEEE Symposium on Security and Privacy*, pages 4143–4162. IEEE Computer Society Press, May 2024. doi:10.1109/SP54263.2024.00260.

[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 Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023: 32nd USENIX Security Symposium*, pages 4445–4462. USENIX Association, August 2023.

[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.

[Zam25] Zama. TFHE-rs: A (practical) handbook, 2025. URL: https://github.com/zama-ai/tfhe-rs-handbook/.