FHE-Rollups: Scaling Confidential Smart Contracts on Ethereum and Beyond

Guy Zyskind
guyz@mit.edu
MIT
Fhenix

Yonatan Erez
yonatan@fhenix.io
Fhenix

Tom Langer
tom@fhenix.io
Fhenix

Itzik Grossman
itzik@fhenix.io
Fhenix

Lior Bondarevsky
lior@fhenix.io
Fhenix

ABSTRACT

Blockchains ensure that all transactions, including those that execute deterministic programs known as *smart contracts*, are processed correctly and without interruption. However, blockchains inherently provide no *confidentiality* – all transaction data, including inputs sent to smart contracts, are public. This has led to a rise of *confidential smart contract* blockchains. These blockchains utilize privacy-preserving techniques to add privacy to smart contracts, but they usually rely on Trusted Execution Environments (TEEs) (e.g., [14, 24]) that are susceptible to side-channel attacks and other security concerns ([7, 13, 33] to name a few).

More recently, several works have focused on achieving confidentiality using Fully Homomorphic Encryption (FHE) (e.g., [1, 30]). While this approach is promising, these works limit scalability as they require all nodes in the network to execute FHE computations and reach consensus over the encrypted state, which is prohibitive.

Instead, in this work and inspired by the recent move towards layer-2 solutions, we present the first rollup-based FHE architecture. We argue that while for plaintext computation rollups are a needed solution, in the context of FHE, where the computational overhead is orders of magnitude higher, they are a necessity.

In our design, we take an optimistic rollup approach, allowing us to avoid the orders of magnitude penalty incurred by state-of-the-art verifiable FHE techniques [34]. In fact, our framework can be seen as a cryptoeconomic solution to solve the same problem of verifiability in FHE.

We implement a proof-of-concept of our solution, and in the process, we show how we can build FHE rollups without making any changes to existing layer-1s like Ethereum, even if they do not support FHE operations inherently. We further implement three smart-contracts that are only possible if data remains confidential, and show that their performance is practical.

1 INTRODUCTION

Blockchains ensure correct execution and censorship resistance in a trust-minimized way (i.e., without trusting any centralized operator), at the expense of data confidentiality. Inherently, all data on-chain is public, because all nodes need to see the data to reach consensus. Despite popular belief, this extreme level of transparency is a by-product and not a goal of the system, and it greatly limits the type of use-cases we can build, since we cannot build any application that needs to utilize sensitive data.

In recent years, researchers and practitioners have employed several privacy-preserving technologies to solve the problem of confidentiality on the blockchain, in a body of work that became known as *confidential smart contracts* (e.g., [2, 14, 29, 30, 38, 39]). Of all of these techniques, Fully Homomorphic Encryption (FHE) is perhaps the most ambitious, as it allows to directly compute over encrypted data without decrypting it.

FHE has improved by leaps and bounds since Gentry presented the first construct almost a decade and a half ago [18]. Still, FHE requires significant computational overhead compared to computing in plaintext, making it impractical for execution at the layer-1 (L1) level where every node is required to replicate the entire computation, which is the approach that state-of-the-art FHE-based confidential smart contracts frameworks are taking [1, 30].

Inspired by the recent movement towards layer-2 solutions in the Ethereum ecosystem [22, 26, 27], we present the first architecture of an FHE-based rollup. We argue that while for plaintext computation rollups are a needed solution, in the context of FHE, where the computational overhead is orders of magnitude higher, they are a necessity.

In a rollup architecture, smart contract execution (the heavy-duty part of validating blocks) is separated from verifying the execution and reaching consensus. This ensures that only a single node (or a small number of nodes) is actually doing the computational heavy lifting, without impairing security. Furthermore, this node can be vertically (and horizontally) scaled as needed, including utilizing more expensive specialized hardware (GPUs, ASICs). The latter is common with zero-knowledge (zk) based rollups [1], which like FHE also leverages computationally-intensive cryptography, and can be leveraged in much the same way for FHE computations.

However, in our design, we take an optimistic rollup approach as opposed to a zk-rollup approach, allowing us to avoid the orders of magnitude penalty incurred by state-of-the-art verifiable FHE techniques [34]. In fact, our framework can be seen as a cryptoeconomic solution to solve the same problem of verifiability in FHE.

1.1 Our contributions

In this paper, we make the following main contributions:

- We introduce the first layer-2 *confidential smart contracts* platform, enabling greater efficiency and scalability.

[1] e.g., https://www.ingonyama.com, https://www.risczero.com

- We demonstrate through a proof-of-concept implementation, that an optimistic FHE rollup can be built on top of Ethereum, without making any changes to the base layer. While our work extends beyond Ethereum and EVM chains, showing that this is possible on Ethereum *today* implies that the most used smart-contract ecosystem can be augmented with confidential smart contracts.
- We implement and benchmark three types of confidential smart contracts in Solidity: (i) a confidential ERC-20 contract; (ii) a sealed-bid auction contract; (iii) and a private voting contract. All of these examples can only operate in a blockchain with confidentiality. We also demonstrate empirically that our solution is concretely efficient and practical.
- Outside the context of blockchains, our solution can be seen as a more efficient (cryptoeconomic) solution to the problem of verifiable FHE.

1.2 Design Goals and Security Model

1.2.1 Design Goals. Our system is built with the following objectives in mind:

- *Correctness and Availability.* A smart contract is executed correctly and with guaranteed output.
- *Input Confidentiality.* Nothing is learned about users' inputs during the execution of a smart contract. Since computations are reactive (i.e., stateful), the state can be considered as another private input.
- *Selective Output Confidentiality.* Smart contract outputs can be re-encrypted and selectively shared with the querying user, or with another with proper permissions. No one else needs to learn anything about the output. Outputs can also be made public.
- *Efficiency.* Execution efficiency is proportional to the computation complexity of the underlying FHE execution. This captures the rollup's efficiency property, which denotes that there is no need for consensus or replicating the computation.

In terms of confidentiality, we note that we do not try to hide the following: (i) identity of the user initiating a transaction (e.g., for executing a smart contract); (ii) The smart contract being called, and the method being called. In other words, there is no *circuit privacy*.

1.2.2 Security Model. We assume our Rollup is built on-top of a layer-1 that provides the usual properties of a blockchain (i.e., correctness and availability).

For our Threshold Services Network (see Section 4), we assume nodes share pairwise secure channels and a broadcast channel (for the latter, the Layer-1 can be used directly). We also assume an honest majority between the nodes, as required by the underlying decryption protocol we use. We note that our architecture generalizes to threshold decryption protocols that support a dishonest majority, and that this assumption is not a hard requirement.

Finally, like other optimistic rollups, we assume at least a single honest validator (also known as a verifier) exists.

1.3 Related Work

Our work builds upon the existing body of research and development of *confidential smart contract* platforms. Unlike all other works, to the best of our knowledge we are the first to describe a solution that operates fully and natively as a layer-2. More specifically, other confidential smart contract platforms usually differ by the kinds of privacy-preserving technologies they use:

- **Trusted Execution Environment (TEEs) Based**. Currently, the only confidential smart contracts networks in production are using TEEs (or secure enclaves) [14, 24, 29, 37]. These networks simulate secure computation by allowing users to encrypt their transactions with keys held inside of a secure enclave. Transactions then get decrypted and executed inside of the enclave, which ensures confidentiality as long as we can trust the security of the TEE. While TEEs are by far the most efficient solution, they are susceptible to side-channel attacks and other vulnerabilities (e.g., [7, 13, 33]).
- **Secure Multiparty Computation (MPC) Based**. Since our work relies on Threshold FHE, we share a similar threat model with these works (e.g., [2–4, 19, 38, 39]). However, for the purpose of presentation, we separate these from FHE-based solutions, as they often rely on linear secret-sharing [5, 20] and garbled circuits techniques [36]. The main drawback in these techniques compared to our work is that MPC protocols need to communicate data proportional to the circuit size in order to evaluate it. In the case of secret-sharing-based MPC, all parties also need to sequentially communicate with all other parties any time they evaluate a multiplication gate. In the context of public blockchains, this is impractical, as the latency is quite high and the bandwidth is limited. Furthermore, as these systems require multiple interacting nodes for every contract execution, they are not amenable to a rollup architecture such as the one we are proposing.
- **Zero-knowledge (ZK)-based**. Different works have considered confidential smart contracts using different ZK schemes. However, since ZK techniques are more suitable for verifiable computation (e.g., [25]), their utility for confidential smart contracts is limited. To overcome this, Hawk [23] suggested having a data-manager – an off-chain party that is tasked with collecting inputs from different clients and is trusted with seeing everyone's data. Alternatively, other platforms impose limitations on developers [8, 9, 21, 35].
- **FHE-based**. In the last couple of years, as a result of significant FHE performance improvements, HE and FHE based solutions have started to emerge [1, 17, 30–32]. These platforms are closest to our work, but none of them adopt a rollup architecture, which limits their scalability in practice. Some of these works adopt a Threshold FHE structure as we do (e.g., [1]), whereas others such as [31] allow only limited functionality, and without a shared state.

2 PRELIMINARIES

2.1 Layer-2 Rollups

Rollups are a scaling solution designed to alleviate the congestion on the primary layer-1 chain, in particular Ethereum. With Ethereum's

growing user base, the need for scaling solutions has become paramount. The primary goal of scalability is to enhance transaction speed and throughput without compromising on decentralization or security. Rollups execute transactions outside the base layer, posting back state-updates alongside proofs of correct execution. Ultimately, the layer-1 reaches consensus on these state updates, but it does so without re-executing the transactions on the base layer. In other words, transactions on the rollup are secured by layer-1's inherent security. There are two main types of rollups, which differ by how proofs are created and verified:

- **Optimistic Rollups**. These assume transactions are valid unless challenged. They move computation off-chain but post transaction data to the layer-1 (or another data availability network), allowing anyone to re-run the transactions off-chain and verify for themselves that the execution is correct. If any verifier detects malicious behavior, they can submit a fraud-proof on-chain, in which case the layer-1 acts as the final arbiter. For this reason, Optimistic Rollups require a dispute period (often of a few days) [22].
- **ZK Rollups**. These rollups similarly execute contracts off-chain and submit validity proofs back when sending a state-update. Validity proofs are constructed using advanced cryptographic techniques known as (succinct) ZK Proofs. They can be efficiently verified on-chain directly, without posting the full transaction data or having a dispute period (e.g., [6]).

Optimistic and ZK Rollups have inherently different trade-offs. Optimistic rollups suffer from a dispute-period delay, making finality longer. They also require posting the transactions themselves on-chain, which negates some of the scalability benefits [2].

On the other hand, ZK Rollups require significant computation power (and time) to produce a proof, especially the closer you try to get to native EVM [10]. A related downside is that these rollups are much more complicated to build, resulting in large amounts of code. The likelihood of critical vulnerabilities with zkEVMs is therefore much higher, at least until they have been battle tested enough over time.

2.1.1 Optimistic Rollups in More Detail. Optimistic rollups bundle multiple off-chain transactions and submit them to the L1 chain, reducing costs for users. They are termed "optimistic" because they assume transactions are valid unless proven otherwise. If a transaction is challenged, a fraud proof is computed. If proven fraudulent, penalties are applied. Today, optimistic rollups operate atop Ethereum, managed by Ethereum-based smart contracts. They process transactions off-chain but post data batches to an on-chain rollup contract. Ethereum ensures the correctness of rollup computations and handles data availability, making rollups more secure than standalone off-chain solutions or side-chains. They also create an inherent economic-security alignment between the two layers, as the layer-2 receives security while paying for incurred fees at the layer-1 level.

[2]This is meant to be mitigated to an extent with EIP-4844 and Danksharding.

From an architectural perspective, optimistic rollups consist of the following:

- *Transaction Execution*. Users send transactions to operators or validators, who aggregate and compress them for the layer-1.
- *Submitting to the layer-1*. Operators bundle transactions and send them to the layer-1 using calldata.
- *State Commitments*. The rollup's state is represented as a Merkle tree. Operators submit old and new state roots, ensuring the chain's integrity.
- *Fraud Proofs and Disputes*. These allow anyone to challenge a transaction's validity. If a challenge is valid (arbitrated by the layer-1), the fraudulent party is penalized.

Fraud Proofs play a vital role in optimistic rollups, and they are at the root of how we ensure that a rollup publishes a correct state update. Even in the face of malicious nodes trying to delay or tamper with transactions, the chain's integrity is preserved as long as there's a single honest node that observes state updates and checks that they are correct. In the context of rollups built on Ethereum, because the actual data is posted onto Ethereum, anyone in the world can act as a verifier. Once an honest verifier detects an incorrect state update (e.g., by including a tampered-with transaction), it can submit a dispute, which initiates the fraud proof game: a multi-round interactive protocol. Here, the asserter (the node that produced the state update) and challenger (the verifier who issued a dispute) follow a protocol overseen by a layer-1 verifier contract to ascertain the honest party. The protocol proceeds recursively, each time dividing the computation into two equal parts. The challenger chooses one part to challenge each time. This process, termed the *bisection protocol*, persists until only one computational step is in question. Once the interactive protocol narrows down to a single instruction, it is the layer-1 contract's turn to resolve the dispute by evaluating the instruction result as well as both the asserter's and the challenger's claims and their respective results to determine which one is correct.

2.2 Fully Homomorphic Encryption (FHE)

Fully Homomorphic Encryption (FHE) enables computations on ciphertexts that, when decrypted, match the results of those operations as if they were performed on the plaintext directly.

In practice, ever since the original scheme by Gentry [18], several FHE schemes have been developed, which are based on the original Learning With Errors (LWE) hardness problem, or related algebraic constructs such as its ring variant [28]. Our implementation utilizes the Torus FHE (TFHE) [15] scheme, but we note that our construct itself does not really matter for the purpose of constructing an FHE-rollup, we describe FHE more generally below, in a black-box manner.

A generic FHE scheme can be denoted by the tuple of algorithms $\mathsf{FHE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$, as follows:

- $\mathsf{Gen}(1^\kappa)$. Given a security parameter 1^κ, the algorithm outputs a pair of keys (pk, sk) where pk is the public encryption key and sk is the secret decryption key. Define domains $\mathcal{P}$ for plaintexts, $\mathcal{R}$ for randomness, and C for ciphertexts, as well as the set of permissible functions $\mathcal{F}$.

- $\mathsf{Enc}(pk, m; r)$. Given the public key pk, a message $m \in \mathcal{P}$ and randomness $r \in \mathcal{R}$, the encryption algorithm produces a ciphertext $c \in C$ such that

$$c = \mathsf{Enc}_{pk}(m; r).$$

- $\mathsf{Dec}(sk, c)$. With the secret key sk and a ciphertext c, the decryption algorithm retrieves the original plaintext message m:

$$m = \mathsf{Dec}_{sk}(c).$$

- $\mathsf{Eval}(pk, f, \{c_i\}_{i=1}^n)$. Given the public key pk, a function $f \in \mathcal{F}$, and a set of ciphertexts $\{c_i\}_{i=1}^n$, this algorithm produces a ciphertext $c' \in C$ such that:

$$\mathsf{Dec}_{sk}(c') = f(\{\mathsf{Dec}_{sk}(c_i)\}_{i=1}^n).$$

  This implies the function f is executed over encrypted data, and its result is encrypted as c'.

3 SYSTEM OVERVIEW

Our platform is built with modularity in mind. It includes the quintessential components of a rollup, alongside new and specific components needed to support (Threshold) FHE. In this section, we briefly describe the different components, as they are illustrated in Figure 1.

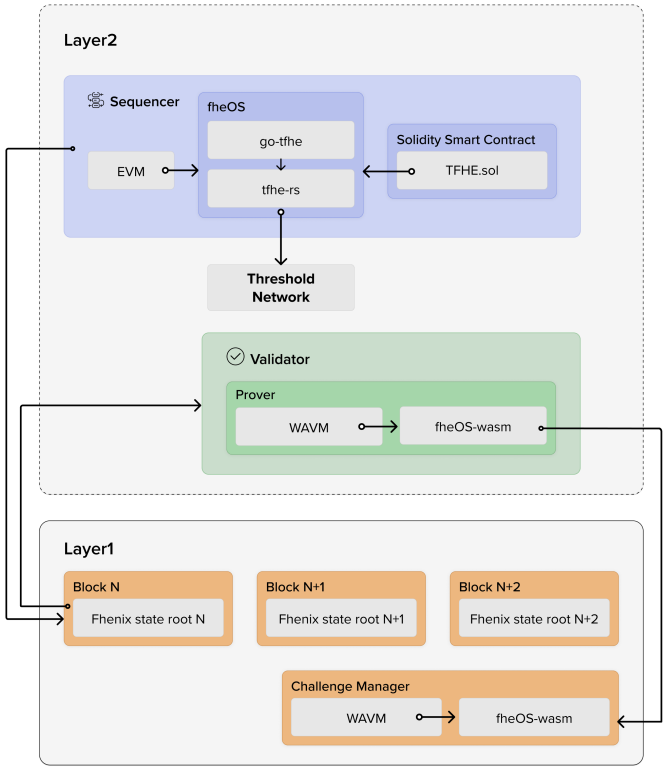

Figure 1: Overview of an FHE Rollup architecture

- **Settlement and Data Availability (DA).** These components are served by the layer-1. In our implementation both will leverage Ethereum directly, but we note that any other layer-1 would work, with the understanding that we inherit its security.
- **Sequencer.** Just like with other layer-2 architectures (e.g., [22]), transaction ordering is done by an entity known as the sequencer. In our design, the same entity is also in charge of transaction (and smart contract) execution, but we note that this two roles could be separated. The sequencer is in-charge of submitting layer-2 state updates periodically. A state update is the result of a batched execution of potentially many transactions.
- **Validators.** These are other nodes in the layer-2 network which observe state-updates submitted by the sequencer. If any validator observes an incorrect state-update (i.e., the sequencer cheated), they can trigger a dispute to the layer-1. In this case, the parties engage in an interactive protocol between the sequencer, challenging validator (the *challenger*) and the layer-1 which acts as an arbitrator (as described in [11, 12, 22]). At the end of the protocol, there is consensus on whether the sequencer or the challenger cheated. To further prevent cheating, it is common to impose financial penalties on the cheating party. The fraud-proof mechanism is further described in Section 5.
- **Threshold Services Network.** Our platform uses Threshold FHE under the hood, which implies users can encrypt their transactions using a single public FHE key. The secret key, however, is shared across a network of nodes we denote as the *threshold services network* (TSN). The TSN is is also in charge of any decryption or re-encryption operations that need to happen from time to time. We discuss this in more detail in Section 4.

4 THRESHOLD SERVICES NETWORK (TSN)

A key component in our design is the Threshold Services Network (TSN). This network is separate from the layer-1 or other rollup components and plays several key roles. In particular, we assume that in a setup phase the TSN generates the network's Threshold FHE key-pair (sk, pk) (using a distributed key-generation protocol), and that pk is published onto the rollup's first block, making it available to all users. In contrast, the secret key has to remain private, and is secret-shared across the TSN participants. There are several threshold decryption protocols to choose from, and our current implementation utilizes [16], which uses Shamir secret-sharing to split sk into n shares, out of which $t+1$ shares are needed to reconstruct.

4.1 Threshold Decryption and Re-encryption

Occasionally, the network will need to decrypt certain results, or re-encrypt them to a designated user. The TSN is in-charge of any such request coming from the rollup.

To illustrate why this is needed, consider the following two examples. First, imagine a private voting contract deployed on the rollup. When users vote for a certain candidate, they encrypt their votes, and the contract tallies these votes (all encrypted). At

a certain point in time, a functionality in the contract should be able to decrypt the tally and announce the winner. This request is routed to the TSN, which decrypts it and returns the result to be stored in the contract's state. This flow is illustrated in Figure 2.

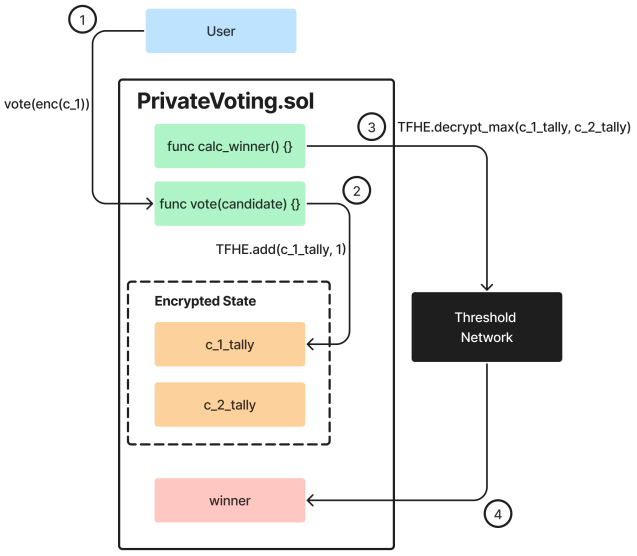

Figure 2: Smart contract request to decrypt FHE-encrypted data on the rollup

A similar example is that of a user who owns a NFT with private metadata only they can see. Such metadata is stored in the contract's state under the network's key. When a user tries to access that information, the contract should threshold re-encrypt it so only the designated user would be able to decrypt and get access to the underlying data.

4.2 Security

It is important to note that the underlying confidentiality guarantees of the entire system are closely related to the trust assumptions of the TSN. Anything that relates to keeping the threshold decryption key safe, correctly decrypting/re-encrypting ciphertexts, etc., is under the responsibility of the TSN.

Currently, the state-of-the-art protocols by [16] require the TSN to have at most $t < \frac{n}{3}$ malicious corruptions for a fast and robust protocol, or $t < \frac{n}{2}$ if we are willing to settle for security with abort.

5 FRAUD PROOFS

The key to Optimistic Rollups lies in their fraud proof mechanism. But how do we fit that mechanism, in particular Ethereum's EVM, to work with smart contracts that execute FHE circuits over encrypted data?

First, observe that FHE, unlike other encrypted computation techniques such as MPC, natively allows anyone to verify that a computation was done correctly, without breaking the privacy guarantees. This is because an adversary holding the encrypted inputs and outputs learns nothing about the underlying encrypted data (if this were not the case, then the encryption scheme would not be semantically secure).

This makes verifying FHE computations compatible with the idea of Optimistic Rollups, at least in theory. As mentioned earlier, Optimistic Rollups are based on posting the full transaction data on the layer-1 (or some other data availability service), alongside the output. In this case, both are encrypted. Just like with plaintext data, any off-chain validator could take the encrypted transaction data, re-execute the transactions, and make sure that they receive the same encrypted output. If this is not the case, then an honest validator could submit a dispute and start the arbitration process with the layer-1.

However, the whole fraud proving mechanism is rooted on the layer-1's ability to determine unequivocally whether the layer-2 node that posted the state update or the disputing verifier (i.e., the *challenger*) is cheating. To do this, the layer-1 needs to be able to run a single computational step of the underlying computation. Since our solution relies on the security of the layer-1, we wish to use Ethereum in our implementation as it is the most secure smart-contract platform. But this introduces a new challenge: How can Ethereum, or any other layer-1, validate execution on FHE primitives without inherent support for FHE operations?

To overcome this challenge, we utilize Arbitrum's Nitro fraud prover [3], which has an Ethereum contract on-chain that can verify the correctness of a single WebAssembly (WASM) opcode. This is sufficient, as we can now compile the underlying FHE libraries (which are written in Rust) to WebAssembly as well, and avoid requiring changes to the layer-1 itself.

Addressing performance concerns, it is rational to assume that if FHE computations are inherently intensive, simulating them in a WASM runtime atop EVM might incur significant performance penalties. While this is a valid concern, it is important to remember that initiating these computations is only mandatory in a dispute scenario, and real-time speed is not an absolute necessity given a sufficient dispute window. Considering that the standard practice allocates approximately seven days for it, we estimate that this is more than sufficient time to settle any such disputes. However, we did not empirically validate this hypothesis, and we note that doing so in the future is important.

We defer additional implementation details regarding the fraud prover to the appendix (Section A.3).

6 EVALUATION

We implement a proof-of-concept of our layer-2 FHE rollup system [4]. Our architecture is based on the Arbitrum stack [22], and uses Ethereum as the layer-1. For FHE capabilities, we use the *tfhe-rs* [5] library, which is written in Rust. We then implement a Solidity wrapper on top, such that we can write standard Solidity smart contracts. Figure 4 is an example of such a contract for private voting.

[3] https://docs.arbitrum.io/inside-arbitrum-nitro
[4] https://github.com/FhenixProtocol/tfhewasm, https://github.com/FhenixProtocol/go-tfhe
[5] https://github.com/zama-ai/tfhe-rs

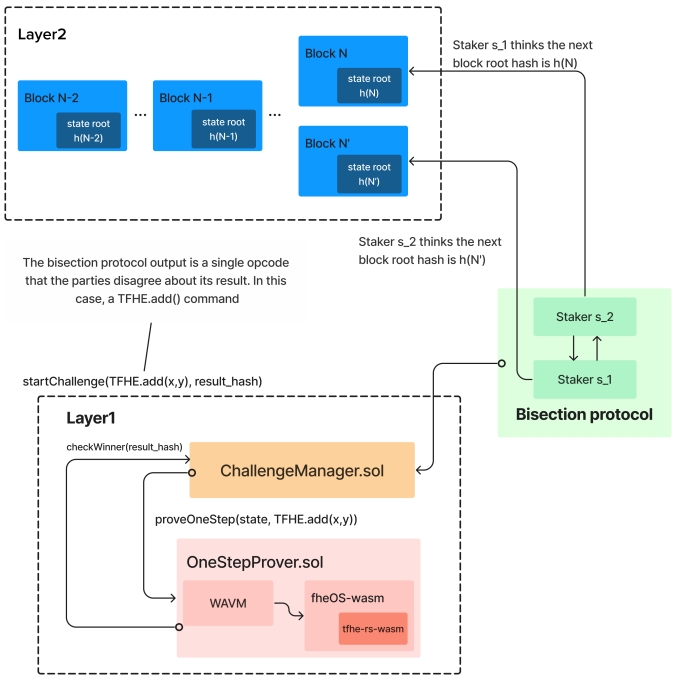

Figure 3: Fraud proof engine for FHE instructions

As a first step, we benchmarked elementary FHE addition and multiplication gates under different VMs, and for (plaintext) integers between 8-256 bits. As can be seen in Table 1, the performance gap between the VMs increases with the bit length. This is expected due to the underlying TFHE scheme [15]. However, on a per dollar basis, choosing a stronger VM does not justify the cost.

We also implemented three confidential Solidity smart contracts of real-world use cases. These include a confidential ERC-20 token contract (allowing for private transactions), a private voting contract, and a sealed-bid auction contract. For all contracts, we used 32-bit encrypted integers. The results are summarized in Table 2, and they illustrate that our solution is practical and concretely efficient.

7 CONCLUSION

In this paper, we presented the first proposed construct of an FHE layer-2 rollup, a novel solution that adds confidentiality to existing blockchains. Our approach leverages the power of FHE to enable encrypted EVM computations, revolutionizing the way transactions are executed and confidential data is handled on-chain. Unlike recent works [1, 17, 30], our approach uses a layer-2 rollup structure to avoid replicating the cost of FHE computations across all nodes, which leads to a much more efficient and practical solution.

While our implementation focuses on Ethereum and the EVM, our construct is generic and can also be of independent value as a system that enables verifiable FHE [34].

```
pragma solidity >=0.8.19 <0.9.0;

import "contracts/FHE.sol";
import "contracts/access/Permissioned.sol";

contract Voting is Permissioned {
    uint8 internal constant MAX_OPTIONS = 4;

    // Pre-compute these to prevent unnecessary gas usage for the users
    euint32 internal _u32Sixteen = FHE.asEuint32(16);
    euint8[MAX_OPTIONS] internal _encOptions = [FHE.asEuint8(0),
    ↪  FHE.asEuint8(1), FHE.asEuint8(2), FHE.asEuint8(3)];

    string public proposal;
    string[] public options;
    uint public voteEndTime;
    euint16[MAX_OPTIONS] internal _tally; // Since every vote is worth
    ↪  1, I assume we can use a 16-bit integer

    euint8 internal _winningOption;
    euint16 internal _winningTally;

    mapping(address => euint8) internal _votes;

    function vote(inEuint8 memory voteBytes) public {
        require(block.timestamp < voteEndTime, "voting is over!");
        require(!FHE.isInitialized(_votes[msg.sender]), "already
        ↪  voted!");
        euint8 encryptedVote = FHE.asEuint8(voteBytes); // Cast bytes
        ↪  into an encrypted type

        ebool voteValid = _requireValid(encryptedVote);

        _votes[msg.sender] = encryptedVote;
        _addToTally(encryptedVote, voteValid /* , _one */);
    }

    function finalize() public {
        require(voteEndTime < block.timestamp, "voting is still in
        ↪  progress!");

        _winningOption = _encOptions[0];
        _winningTally = _tally[0];
        for (uint8 i = 1; i < options.length; i++) {
            euint16 newWinningTally = FHE.max(_winningTally, _tally[i]);
            _winningOption =
            ↪  FHE.select(newWinningTally.gt(_winningTally),
            ↪  _encOptions[i], _winningOption);
            _winningTally = newWinningTally;
        }
    }

    function winning() public view returns (uint8, uint16) {
        require(voteEndTime < block.timestamp, "voting is still in
        ↪  progress!");
        return (FHE.decrypt(_winningOption),
        ↪  FHE.decrypt(_winningTally));
    }

    function _requireValid(euint8 encryptedVote) internal view returns
    ↪  (ebool) {
        // Make sure that: (0 <= vote <= options.length)
        return encryptedVote.lte(FHE.asEuint8(MAX_OPTIONS - 1));
        //FHE.req(isValid);
    }

    function _addToTally(euint8 option, ebool voteValid /* , euint16
    ↪  amount */) internal {
        for (uint8 i = 0; i < options.length; i++) {
            // euint16 amountOrZero =
            ↪  FHE.select(option.eq(_encOptions[i]), _one, _zero);
            ebool amountOrZero =
            ↪  option.eq(_encOptions[i]).and(voteValid); // `eq()`
            ↪  result is known to be enc(0) or enc(1)
            _tally[i] = _tally[i] + amountOrZero.toU16(); // `eq()`
            ↪  result is known to be enc(0) or enc(1)
        }
    }
}
```

Figure 4: Excerpt of a Voting Contract.

Table 1: FHE addition and multiplication using different VMs and for 8-256bit plaintext integers.

System			Addition (ms)						Multiplication (ms)					
VM	**Cores**	**Price/yr**	**8**	**16**	**32**	**64**	**128**	**256**	**8**	**16**	**32**	**64**	**128**	**256**
I9-13900K	32	$1,000	47.527	71.752	122.07	202.26	440.57	975.29	99.034	207.35	589.84	2084.2	7715.5	28946
R7i (x32)	32	$18,230	49	77.98	121.4	159.05	343.57	770.72	102.49	205.44	493.72	1587.2	5861.6	21777
hpc7a.96xlarge	96	$62,208	63.201	83.203	102.75	120.27	145.52	192.26	119.14	164.41	229.74	412.67	1059.4	3446.3
m6id.metal	128	$52,790	70.5	100	132	186	249	334	144	216	333	832	2500	8850

Table 2: Benchmarking different Solidity Contracts using FHE

Contract Name	Transaction	Time (ms)
FHERC20	transfer()	832
Voting	vote()	620
Auction	bid()	3500

Additionally, we also laid out the architecture of our proposed FHE rollup system, including its components and layers, and have proposed and implemented a proof-of-concept of a fraud-proof solution that requires no changes whatsoever to Ethereum. Our proof-of-concept results illustrate that our system is practical for everyday use.

REFERENCES

[1] Zama AI. 2023. FHEVM Whitepaper. https://github.com/zama-ai/fhevm/blob/main/fhevm-whitepaper.pdf. Accessed: 27-10-2023.

[2] Aritra Banerjee, Michael Clear, and Hitesh Tewari. 2021. zkhawk: Practical private smart contracts from mpc-based hawk. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 245–248.

[3] Carsten Baum, James Hsin-yu Chiang, Bernardo David, and Tore Kasper Frederiksen. 2022. Eagle: Efficient Privacy Preserving Smart Contracts. *Cryptology ePrint Archive* (2022).

[4] Carsten Baum, Bernardo David, and Rafael Dowsley. 2020. Insured MPC: Efficient secure computation with financial penalties. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 404–420.

[5] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM, 1–10.

[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable zero knowledge with no trusted setup. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 701–732.

[7] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. {ÆPIC} Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*. 3917–3934.

[8] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 947–964.

[9] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. 2020. Zether: Towards privacy in a smart contract world. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers*. Springer, 423–443.

[10] Vitalik Buterin. 2022. zkEVM and zkRollup. https://vitalik.ca/general/2022/08/04/zkevm.html. Accessed: dd-mm-yyyy.

[11] Ran Canetti, Ben Riva, and Guy N Rothblum. 2011. Practical delegation of computation using multiple servers. In *Proceedings of the 18th ACM conference on Computer and communications security*. 445–454.

[12] Ran Canetti, Ben Riva, and Guy N Rothblum. 2013. Refereed delegation of computation. *Information and Computation* 226 (2013), 16–36.

[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[14] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 185–200.

[15] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[16] Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P Smart, Samuel Tap, and Michael Walter. 2023. Noah's Ark: Efficient Threshold-FHE Using Noise Flooding. *Cryptology ePrint Archive* (2023).

[17] Wei Dai. 2022. Pesca: A privacy-enhancing smart-contract architecture. *Cryptology ePrint Archive* (2022).

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

[19] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. 2021. YOSO: You Only Speak Once: Secure MPC with Stateless Ephemeral Roles. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II*. Springer, 64–93.

[20] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play any mental game. *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (1987), 218–229.

[21] Aleo Systems Inc. 2022. Aleo: A Zero-Knowledge Operating System. https://aleo.org/. Accessed: dd-mm-yyyy.

[22] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 1353–1370. https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner Accessed: dd-mm-yyyy.

[23] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 839–858.

[24] Rujia Li, Qin Wang, Qi Wang, David Galindo, and Mark Ryan. 2022. SoK: TEE-assisted confidential smart contract. *arXiv preprint arXiv:2203.08548* (2022).

[25] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 238–252.

[26] Optimism PBC. 2022. Optimism: Optimistic Ethereum. https://optimism.io/. Accessed: 27-10-2023.

[27] Joseph Poon and Vitalik Buterin. 2017. Plasma: Scalable Autonomous Smart Contracts. https://plasma.io/plasma.pdf. Accessed: dd-mm-yyyy.

[28] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)* 56, 6 (2009), 1–40.

[29] SCRT. 2021. The Secret Network Graypaper. https://scrt.network/graypaper.

[30] Ravital Solomon and Ghada Almashaqbeh. 2021. smartfhe: Privacy-preserving smart contracts from fully homomorphic encryption. *Cryptology ePrint Archive* (2021).

[31] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. 2022. Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 179–197.

[32] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. 2019. zkay: Specifying and enforcing data privacy in smart contracts. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1759–1776.

[33] Stephan Van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX fails in practice. https://sgaxe.com/files/SGAxe.pdf

[34] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. 2023. Verifiable fully homomorphic encryption. *arXiv preprint arXiv:2301.07041* (2023).

[35] Zachary J Williamson. 2018. The aztec protocol. *URL: https://github.com/AztecProtocol/AZTEC* (2018).

[36] Andrew C Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 162–167.
[37] Hang Yin, Shunfan Zhou, and Jun Jiang. 2019. Phala network: A confidential smart contract network based on polkadot.
[38] Guy Zyskind, Oz Nathan, et al. 2015. Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE Security and Privacy Workshops*. IEEE, 180–184.
[39] Guy Zyskind, Oz Nathan, and Alex Pentland. 2015. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471* (2015).

A IMPLEMENTATION DETAILS

A.1 fheOS

The core FHE logic sits in our fheOS library. It is an encrypted computation library of pre-compiles for common encrypted opcodes, such as comparing two numbers and doing arithmetic operations like addition and multiplication. It gives Smart Contracts running on the network the ability to use FHE primitives within the contract. That means that dApps running on (or using) the rollup will be able to integrate encrypted data in their smart contract logic. Developers will have the choice to decide what will be encrypted and what will remain plaintext. fheOS is an EVM-friendly wrapper around FHE libraries, similar to fhEVM [1]. For implementation purposes and modularity, we opted to design our own version.

The fheOS library is the core engine of the rollup node; smart contracts utilizing encryption features will call fheOS precompiles for common FHE operations, and fheOS itself will be responsible for communication and authentication between the rollup and the Threshold Services Network (TSN, see Section 4) for decryption and re-encryption requests while proving that the decryption request is legitimate.

The fheOS library is designed to be injected as an extension into any existing flavor of EVM, meaning that it is fully EVM-compatible, thus keeping all the existing EVM functionality in place while boosting developers' ability to explore new use cases.

A.2 go-tfhe

As we sought to extend the capabilities of Ethereum's *go-ethereum (geth)*, their predominant client written in Go, we encountered a challenge: the core FHE mathematical operations library, tfhe-rs, is written in Rust, necessitating communication via a foreign function interface (FFI). To address this, we developed *go-fhe*, which encompasses:

(1) **Go-based API**. This component provides all essential interfaces for executing FHE mathematical operations using Go. It serves as the primary point of interaction for blockchain developers familiar with Go.
(2) **Rust Wrapper for TFHE.rs**. We created a specialized wrapper that adapts the TFHE.rs library functions for blockchain applications.
(3) **FFI Integration between Go API and the Rust Wrapper**. To bridge the two languages, our interface employs cgo and extern "C" mechanisms. This facilitates bindings between Golang and Rust, aligning function calls, types, and naming conventions.

In general, go-tfhe acts as a modular extensionable lightweight bridge between tfhe-rs and the blockchain application.

A.3 Fraud Prover Implementation Details

During the course of developing the proof-of-concept FHE fraud prover, we had to overcome several hurdles, which we note here for completeness.

First, To run *go-tfhe* as a part of the fraud proof mechanism, it was necessary to adapt the code for execution in WebAssembly (*WASM*). Given our reliance on FHE code written in Rust, while much of blockchain code is traditionally in Golang, we faced challenges in native compilation to a unified WASM module - the default bindings between Rust and Golang use *cgo*, which is not compatible with WASM. To address this, we crafted bindings in WASM to bridge between Golang, using the mainline Golang compiler's external function directives, and Rust, creating a dedicated Rust file as the primary WASM build target, eventually linking both using *wasm-merge* for a cohesive *.wasm* output.

Furthermore, we had to modify *tfhe-rs* as well. At the time of this writing, *tfhe-rs* supports compilation and execution of WASM in browsers only. In our case – smart contracts running atop of a blockchain – the interfaces available differ from those of browser contexts. Notably, some API calls, such as accessing operating-system capabilities or multithreading, are not accessible. For *TFHE.rs* to be compatible with a smart contract context, two primary modifications were made:

(1) **Disabling Multithreading**. Given that most smart contracts do not accommodate multithreading or concurrency, the code was adjusted to operate in a single-threaded and deterministic manner.
(2) **Custom Random Number Provider Integration**. The initial version of *tfhe.rs* depended on predefined random number providers or *seeders*. The only available seeders were contingent on the operating system's ability to generate random numbers. We introduced a custom seeder managed by the library user. This seeder receives an input seed and passes it into a ChaCha20-based seeded Pseudorandom Number Generator (*PRNG*) provider. This modification negates the need for non-deterministic random numbers in the FHE implementation, facilitating external validation by replicating computations using consistent inputs from blockchain data and state.

We note that based on our benchmarks of *tfhe-rs* running in this context show an order of magntitude performance degradation when using wasmer/cranelift and 8x when using wasmer/llvm for add operations (tested on i9-13900K, 128 GB RAM, wasmer 4.0). We note that the Arbitrum fraud proof engine makes use of software floating points which should impact performance further, but we did not complete benchmark tests as of writing this paper.