fhEVM

Confidential EVM Smart Contracts using Fully Homomorphic Encryption*

Morten Dahl Clément Danjou Daniel Demmler Tore Frederiksen Petar Ivanov
Marc Joye Dragos Rotaru Nigel Smart Louis Tremblay Thibault

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 privacy as all on-chain data is widely distributed and publicly visible, even when hidden behind pseudonymous addresses.

Solving this challenge, while ensuring the strong security guarantees of existing blockchains, is an ongoing research area. Without a solution, the transparent nature of blockchains gets in the way of adaptation of certain, otherwise very useful applications, making the widespread use of blockchains depend on solving this privacy challenge.

Our aim is to solve this challenge for general-purpose (meaning Turing-complete) blockchains such as Ethereum. We believe that the only scalable approaches to smart-contract development do not necessitate developers to have prior expertise in cryptography. Therefore, we design our solution to provide a seamless and intuitive developer experience. Moreover, successful privacy-preserving solutions must support the usual blockchain use-cases and design patterns such as easily mixing data from different users, keeping encrypted data on-chain, and smart-contract composition.

Our solution presented here is based on *fully homomorphic encryption* (FHE) combined with threshold protocols. All data is on-chain and available to everyone with some of it being in an encrypted form. Due to the deterministic properties of homomorphic function evaluation, everyone can perform computations on the encrypted data and use typical consensus protocols. This also, intentionally, preserves one of the key benefits of transparency on blockchains: the computation carried out remains public, while only the data that is being computed on is hidden.

1.1 Our contributions

This document presents novel blockchain components and how they are used to build an Ethereum Virtual Machine (EVM) based blockchain supporting computation on encrypted values. The concrete contributions include:

- An fhEVM, which integrates Zama's open source FHE library TFHE-rs [Zam22], into a typical EVM, exposing homomorphic operations as precompiled contracts.
- A decryption mechanism [Sma23] based on distributed threshold protocols [DDE+23] that prevents misuse by malicious smart contracts, yet is flexible and non-intrusive to honest smart-contract developers.
- A Solidity library that makes it easy for smart-contract developers to use encrypted data in their contracts, without any changes to compilation tools.

Note that while the presentation given here is in terms of an EVM-based blockchain, our approach is general enough to support other blockchains, for instance based on WASM. It can also easily be made to use non-threshold based decryption mechanisms.

*Version 1.0.0 (September 21, 2023).

1.2 Applications

On-chain encrypted values enhance or even unveil many use cases of blockchain, with a few described below.

1.2.1 Encrypted ERC-20 tokens. The ERC-20 token standard for fungible tokens is an important standard for blockchains. However, by the public nature of blockchain systems, the individual balances of ERC-20 token holders are public and the mere pseudonymity offered may lead to privacy concerns for individuals [BSBQ21]. Since the fhEVM enables the use of encrypted values in smart contracts, it is possible to construct an encrypted version of the ERC-20 token standard. To do so, it suffices to change the data type of balances from integers to encrypted integers, and replace each operation with their respective FHE counterpart.

1.2.2 Blind auctions. A first-price sealed-bid auction allows bidders to privately submit bids so that no one learns their amount. When the bidding period is over, the highest bidder is identified and declared winner. Blind auctions encourage bidders to bid no more than what they think the auction prize is worth. Such auctions can be naturally implemented entirely on-chain using encrypted values, without needing a trusted entity to perform the computation on the plaintext bids, unlike solutions based on commitments or zero-knowledge proofs. Using an encrypted ERC-20 token, bidders simply transfer an encrypted amount to the blind auction smart contract, which then compares bids to determine and reveal the winner.

1.2.3 Privacy-enhanced DAOs. Decentralized autonomous organizations (DAOs) have become increasingly popular in the blockchain space as a way to create self-governing communities without a central authority. They rely on smart contracts to enforce rules and make decisions based on the votes from their members. In general, a DAO has a treasury which is managed by its members via proposals. Proposals are voted on by members and are executed only if the vote has passed. Voting systems can be implemented with an ERC-20 token contract where the amount of tokens held by a member reflects its voting power. On-chain encrypted values therefore enable DAO members to cast votes confidentially. Using an encrypted treasury, DAOs can also compete fairly in auctions, preventing others from knowing in advance the details of the treasury and as a result its bidding power.

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

1.3 Related work

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

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

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

Zexe [BCG+20] and VeriZexe [XCZ+23] allow arbitrary scripts to be evaluated within zero-cash style blockchains using zkSNARKs. The limitation is that since multiple smart contracts / parties cannot access encrypted state on-chain, the input data has to be known by at least one party to generate the zkSNARKs. Currently it is impossible to update encrypted states without revealing them using this methodology.

Hawk [KMS+16] uses ZK proofs where the inputs to the

smart contracts are revealed to a trusted manager that does the computation.

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

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

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

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

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

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

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

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

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

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

2 HIGH-LEVEL OVERVIEW

This section gives a high level overview of our approach, with further details in the subsequent sections.

We make little assumptions about the consensus layer beyond relying on it for providing agreed-upon blocks of transactions to execute and the public signature keys of all current validators. Tendermint is used in our concrete implementation. Importantly, we make no changes to the consensus protocols and directly inherit their security properties.

2.1 Global FHE key

Our solution relies on a global FHE key under which all inputs and private state are encrypted. This is an important design decision since it makes it easy to mix encrypted data from multiple users and across multiple smart contracts.

The encryption mechanism is asymmetric, with the public encryption and evaluation keys stored on-chain, and the private decryption key secret shared across the validators. A suite of threshold protocols is used when the decryption key is needed, keeping it secure at all times.[1] Specifically, no group of validators below a certain threshold can make any use of the private decryption key on their own. The key is generated during a setup phase by the initial validators using a threshold protocol, and securely re-shared when the validator set is changing.

2.2 Encrypted inputs

In order to provide an encrypted input to a transaction or view function, users are required to submit two values: The first value is the input encrypted using the global public FHE key. The second value is the associated valid zero-knowledge proof of plaintext knowledge (ZKPoK) [Sma23, § 2.4]. Together, these two components form a *certified ciphertext*.

The ZKPoK ensures that the ciphertext is well-formed, that the user knows the underlying plaintext message, and that the ciphertext cannot be used in another context.

As explained in more detail in Section 6, these requirements are satisfied via a signature-proof-of-knowledge scheme (a term introduced by Camenisch and Stadler [CS97] and formalized by Chase and Lysyanskaya [CL06]) where the user binds a statement to a proof that they know the message that has been encrypted. A global *common reference string* (CRS) is required for this scheme [Lib23], which is also generated by the initial validators using a threshold protocol during the setup phase and stored on-chain.

Before doing anything with the ciphertext, the receiving smart contract must ask the fhEVM to verify the ciphertext and the corresponding proof. If successful, the fhEVM stores the ciphertext on-chain and returns a handle to it that can be used for further processing. If not, execution is reverted. See Section 4 for more details.

2.3 Encrypted values and execution

Besides verification as described above, all operations on ciphertexts performed by smart contracts are done through handles. The fhEVM keeps a mapping between handles and ciphertexts in memory. The mapping is automatically updated when new ciphertexts are computed during execution. While this also has some efficiency benefits, the primary reason to use the mapping is that it provides an efficient way to ensure that smart contracts only operate on ciphertexts that have been honestly obtained. Intuitively, this prevents malicious actors from breaking the expected confidentiality guarantees, for instance by deploying a smart contract with a "stolen" ciphertext baked into it, and then having the contract ask for it to be decrypted. Informally, a ciphertext is honestly obtained by a smart contract if and only if it was:

- received as an input from a user who proved that they know what message it contains;
- sent from another smart contract that had honestly obtained it before;
- loaded from storage where it was honestly obtained when it was stored;
- computed as the output of a homomorphic operation on honestly obtained inputs.

Note that these rules allow for secure composition of (potentially malicious) smart contracts.

Precompiled contracts are used to move the computation of FHE operations from the EVM execution environment to native or hardware-accelerated execution environments. This allows for faster computation of these complex cryptographic operations. Each FHE operator is associated with

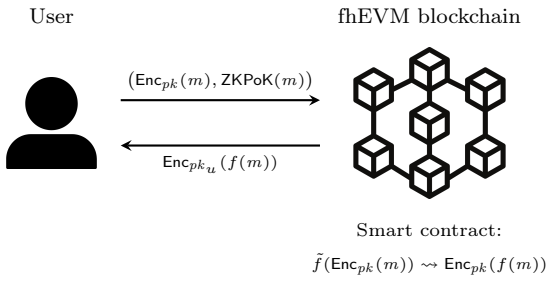

Figure 2.1: Encrypted inputs and outputs. The user sends value m encrypted with the blockchain's public FHE key pk together with a valid ZKPoK as input to the blockchain. The smart contract homomorphically computes on the encrypted value using FHE, denoted as $\tilde{f}(\cdot)$. To output an encrypted value our system conceptually re-encrypts $f(m)$ under the user's public key pk_u using distributed decryption.

[1] Note that we here only describe our key management solution based on secret sharing and threshold protocols. It is straightforward to use another solution, for instance a centralized solution based on trusted hardware.

a precompiled contract via a static address, giving smart-contract developers the ability to call these from their Solidity code without the need to change the Solidity language nor its compiler. See Section 3 for more details.

It is also worth noting that precompiled contracts have fully customizable gas prices, meaning that the price of FHE operations can be set according to crypto-economic factors.

2.4 Decryption and re-encryption

Anyone who wishes to recover the plaintext value of a ciphertext must do so by calling a function of a smart contract, which in turn must call either the decrypt or re-encrypt functionality exposed by the fhEVM. In other words, it is the public code of a smart contract that decides whether such a request is granted, and users are responsible for only sending their encrypted data to smart contracts they trust.

In the case of the decryption, a threshold protocol is used to return the plaintext to each validator which stores it on-chain. When this is undesirable, re-encryption can be used instead, which securely transforms a ciphertext encrypted under the global FHE key into another ciphertext encrypted under a user-provided classical public encryption key using a threshold protocol. The user's public encryption key may be ephemeral and can be signed by the user in order to prevent impersonation attacks.

Since re-encryption requests are completed using a threshold protocol, view functions cannot be executed by neither any full node nor any single validator. We mitigate this issue by introducing a network gateway that distributes view function calls to all validators, who in turn each execute the corresponding smart contract logic to determine whether they should rightfully participate in the threshold protocol. Details are given in Section 7.

2.5 Full nodes

Given that non-validating full nodes do not participate in the threshold decryption protocol, they do not have instantaneous access to results of decryptions. Such nodes need to have access to these to catch up to the head of the chain. They are therefore stored on-chain in such a way that full nodes can easily query validators to get the result of previous decryptions. This mechanism is explained in detail in Section 4.7.

3 SOLIDITY LIBRARY

One of our main design goals for the private smart contract framework is usability; i.e., allowing developers to accomplish their task with the least amount of effort. We consider two aspects of usability: one is utilizing the existing development toolset as much as possible, the other is to make it as easy as possible for developers to express smart contract logic working on encrypted data.

The Ethereum and Solidity toolsets are very rich, mature, and widely used by blockchain developers. They include compilers, debuggers, IDEs, and libraries. In order to add FHE support to these toolsets, we opt for a Solidity library that uses Yul inline assembly to call the fhEVM precompiled contracts. This hides implementation details under a high-level Solidity API so that FHE functionalities are just normal Solidity functions or overloaded operators.

3.1 Encrypted data types

Our Solidity library, `TFHE`, currently exposes encrypted integer types of three different bit-lengths: `euint8`, `euint16`, and `euint32`. It also exposes an encrypted boolean type `ebool`. These are Solidity user defined value types (wrapping a `uint256`) and may for instance be used as variables, parameters, and values in mappings and arrays.

3.2 Encrypted inputs

When a smart contract receives a certified ciphertext from a user, it must first verify its accompanying proof, thereby converting the input into a usable `euint` by calling `TFHE.asEuint`. If the verification is successful, the fhEVM returns an encrypted value that is ready to use by the smart contract. If not, execution is reverted.

An example of this is shown in Listing 3.1, where `amountCt` holds a certified ciphertext, the call to `asEuint32` asks the fhEVM to verify the underlying zero-knowledge proof and return an encrypted value for further usage.

3.3 Encrypted operations

The fhEVM offers various operations on the encrypted data types. These are implemented by calling a precompiled smart contract, which in turn calls out to the TFHE-rs library [Zam22]. See Listing 3.2 for an example using multiplication and addition.

```
function verify(
    bytes calldata amountCt
) public returns (euint32) {
  euint32 amount = TFHE.asEuint32(amountCt);
  return amount;
}
```

Listing 3.1: Example of converting a certified ciphertext `amountCt` to an encrypted integer `amount`.

```
function compute(
    euint32 x,
    euint32 y,
    euint32 z
) public returns (euint32) {
  return TFHE.mul(TFHE.add(x, y), z);
}
```

Listing 3.2: Example of computing on encrypted integers `x`, `y`, and `z`.

The fhEVM currently supports typical arithmetic operations, boolean operations, and comparisons.

3.4 Decryption

Smart contracts may decrypt values by calling `TFHE.decrypt`. This triggers the execution of a threshold protocol and consequently comes with a certain latency. The result is a plaintext value of the corresponding data type.

3.5 Require statements

Smart contracts may decrypt and check encrypted boolean values during execution. If the boolean predicate decrypts to "true" then execution continues, otherwise it is reverted. See Listing 3.3 for an example.

There is some latency associated with checking require conditions due to the threshold decryption that must be performed. To mitigate this, the fhEVM offers an optimistic require `TFHE.optReq`, which accumulates the boolean conditions instead of decrypting them immediately, and performs a single decryption only when needed. The downside is that execution is not reverted as soon as a condition is violated, potentially leading to a higher gas consumption. The accumulated conditions are decrypted at the end of executing a transaction or view function, and before any `TFHE.decrypt` operation. The reason for the latter is that there is an information leak associated with these operations, and it is desirable to minimize the amount of information leaked.

```
function transfer(
    address from,
    address to,
    euint32 amount
) public {
  ebool cond = TFHE.lte(amount, balances[from]);
  require(TFHE.decrypt(cond));

  balances[to] = TFHE.add(balances[to], amount);
  balances[from] = TFHE.sub(balances[from], amount);
}
```

Listing 3.3: Example of checking a condition on encrypted integers `amount` and `balances[from]` using decryption and a require statement.

```
function transfer(
    address from,
    address to,
    euint32 amount
) public {
  ebool cond = TFHE.lte(amount, balances[from]);

  euint32 oldTo = balances[to];
  euint32 newTo = TFHE.add(oldTo, amount);
  balances[to] = TFHE.cmux(cond, newTo, oldTo);

  euint32 oldFr = balances[from];
  euint32 newFr = TFHE.sub(oldFr, amount);
  balances[from] = TFHE.cmux(cond, newFr, oldFr);
}
```

Listing 3.4: Example of using conditional multiplexing to avoid information leakage introduced by decryptions and require statements.

Finally, the inherent leak of information produced by a smart contract decrypting encrypted values may be prohibiting for some applications. For these cases, the fhEVM offers a multiplexer operator `TFHE.cmux` as illustrated in Listing 3.4. This operation computes both the true value and the false value, and combines these using the control value. With this operation there is no leakage but more gas is spent computing both values.

3.6 Encrypted outputs

Our infrastructure allows for programmable privacy by letting smart contract developers choose precisely who has access to read which encrypted values. This is done by having the explicit `TFHE.reencrypt` operation that may be preceded by `TFHE.decrypt` statements and used within `if` statements. A simple example is shown in Listing 3.5.

Note that `TFHE.reencrypt` can *only* be used in view functions because there is no point in persisting its result to smart contract storage. The fact that no signature is re-

```
function balanceOf(
  bytes32 publicKey
) public view returns (bytes memory) {
  return TFHE.reencrypt(
    balances[msg.sender],
    publicKey
  );
}
```

Listing 3.5: Example of re-encrypting to a user-provided public key.

quired to execute view function calls presents a security issue, but it is up to the smart contract developer to implement a solution suitable for the given application. One solution is to ask the user to sign the public encryption key, for instance according to the EIP-712 standard [BLE21], linking it to the on-chain identity, and making sure that the sender of the public encryption key is who they claim to be. It is the responsibility of the developer to have the smart contract verify this signature to ensure authenticity of the provided user public key.

4 FHEVM IMPLEMENTATION

In this section we detail the inner-workings of the fhEVM. It is assumed that the reader is already familiar with common notions of the EVM [Woo21]. The fhEVM augments the EVM by adding several stateful precompiled smart contracts. Contrary to conventional smart contracts, these do not contain EVM bytecode. Instead, their behavior is entirely defined by functions baked into the EVM implementation and expressed in its programming language. Moreover, their gas price is determined at the contract level as opposed to at the bytecode level, making it possible to adjust the relative price between plaintext and encrypted operations. They are stateful because they are allowed to manipulate the EVM's memory and storage.

4.1 Encrypted values, ciphertexts, and handles

Encrypted values (or `euints`) are simple wrappers around handles h represented as `uint256` values and computed by applying a cryptographically secure hash function to the ciphertext c:

$$h \coloneqq \mathsf{Keccak256}(c) \ .$$

We will often write h_c to denote this computation.

Besides converting a ciphertext to a handle, smart contracts can only ask the fhEVM to perform operations on ciphertexts via their handle. In other words, smart contracts manipulate ciphertexts via handles only, and do not directly control their bytes.

Note that since handles are computed by applying a cryptographically secure hash function to ciphertexts, it is overwhelmingly unlikely that an arbitrary `uint256` value will be misinterpreted as a handle. Likewise, it is overwhelmingly unlikely that the handles of different ciphertexts will collide. This would imply that the standard Keccak256 hash function is insecure.

4.2 Privileged memory

We introduce privileged memory, a component required for tracking which ciphertexts were honestly obtained. This memory is implemented as a region of fhEVM memory, is only mutable by the fhEVM, and has the lifetime of the execution of a transaction or view function call.

Privileged memory holds a mapping μ of handles h_c to pairs $(c, \mathcal{S})$, where c is the associated ciphertext and $\mathcal{S}$ is a set of EVM call stack depths. During execution, this mapping records for which call stack depths a ciphertext was honestly obtained, and may therefore be used as input for homomorphic operations and decryptions. Formally, we say that a ciphertext c is *honestly obtained* if h_c is in the domain of μ and $csd \in \mathcal{S}$ for $(c, \mathcal{S}) = \mu(h_c)$ where csd is the current call stack depth.

Similarly, when we say that we *mark ciphertext c as honestly obtained*, we mean that μ is updated as follows:

$$\mu(h_c) \coloneqq \begin{cases} (c, \mathcal{S} \cup \{csd\}) & \text{if } \mu(h_c) = (c', \mathcal{S}) \\ (c, \{csd\}) & \text{if } \mu(h_c) = \emptyset \end{cases} .$$

where csd again is the current call stack depth. Note that $c = c'$ with overwhelming probability due to the use of a cryptographically secure hash function.

4.3 Ciphertext verification

The fhEVM exposes a precompiled contract to verify certified ciphertexts entering the system as inputs from users. Given a ciphertext c and a proof π, this precompile verifies the proof using the scheme described in Section 6, passing in `tx.origin`, `msg.sender` and `functionSelector` from

the EVM's execution context as the user address, the smart contract address and a selector of the called smart contract function, respectively.

`functionSelector` uniquely identifies a smart contract function and is defined as the first 4 high-order bytes of Keccak256(`functionSignature`) in big-endian, where `functionSignature` is defined to be the function name followed by a list of parameter types separated by a single comma and enclosed in parentheses. If verification passes, then ciphertext c is marked as honestly obtained. Otherwise, the function call is reverted.

For the above method to work reliably and to reduce complexity, ciphertext verification is only allowed at depth 1 of the fhEVM call stack (i.e. in a smart contract function called by an externally owned account). In that way, user libraries can generate proof π using `msg.sender` as the smart contract address they invoke and `functionSelector` as the function they directly call. Otherwise, they would need to know where in the call stack ciphertext verification is requested and, consequently, the corresponding `msg.sender` and `functionSelector` at this depth. That introduces complexity and dependencies between proof generation and the implementation of the called smart contract.

4.4 Privileged storage

We introduce smart contract privileged storage, a novel type of smart contract storage. Like regular smart contract storage, this is a unique per-contract storage location in the blockchain state. However, unlike regular smart contract storage, the privileged storage is only mutable by the fhEVM and not by the smart contract code itself. Having it per-contract allows the fhEVM to efficiently reclaim storage when the contract is deleted. The privileged storage is implemented by creating an associated contract and then using that contract's storage. This associated smart contract does not contain any EVM bytecode.

The fhEVM uses privileged storage to store an honestly obtained ciphertext c when a smart contract stores its handle h_c. This means that although storing h_c is cheap because it is simply a `uint256` value, the fhEVM also implicitly stores the larger associated ciphertext. To mitigate this storage cost, c is only stored once, even if h_c itself is stored multiple times, and a set $\mathcal{R}$ is used to keep track of the locations h_c is stored at.

Concretely, the `CREATE` and `CREATE2` opcodes are augmented to create an additional contract. Let α be the address of the account deploying the contract, then $a \xleftarrow{[96..255]} \mathsf{Keccak256}(\alpha, n)$ is the address of the regular contract, where n is the nonce of account α, and $a' \xleftarrow{[96..255]} \mathsf{Keccak256}(a, 0)$ is the address of the associated contract used for privileged storage (see Figure 4.1). Let σ_a denote the storage of the former and ρ_a denote storage of the latter. The `SELFDESTRUCT` opcode is augmented to also remove ρ_a in addition to σ_a, which ensures that ciphertexts are automatically removed from storage when a contract is destroyed.

Note that by the design of the address creation mechanism, it is infeasible to create a smart contract that owns the same storage space as any other contract. Therefore, we obtain from our construction a storage space for ciphertexts that is shielded from manipulation by any malicious smart contract.

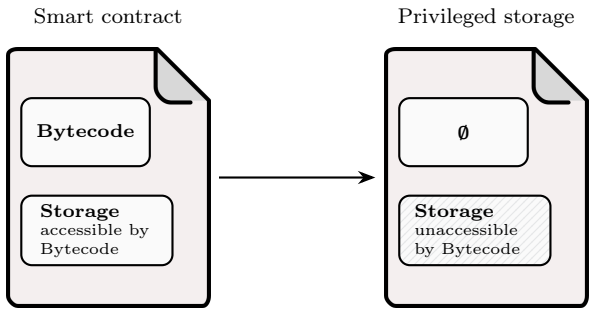

Figure 4.1: Privileged storage.

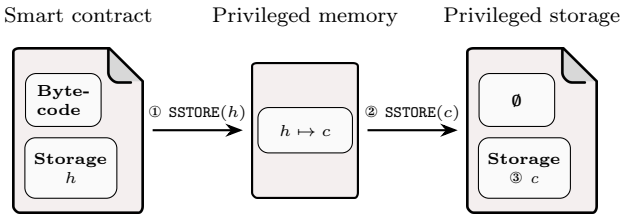

Figure 4.2: Interaction between a smart contract, privileged memory and privileged storage when calling the `SSTORE` opcode. The stack call depths are omitted.

The `SSTORE` opcode is also augmented so that when a contract asks to store handle h_c at location ℓ in σ_a, garbage collection is performed by looking up $(\mathcal{R}', c') \leftarrow \rho_a(h')$ for

$h' = \mathsf{Keccak256}(\sigma_a(\ell))$, and updating ρ_a as follows:

$$\rho_a(h') \coloneqq \begin{cases} \emptyset & \text{if } \mathcal{R}' = \{\ell\} \\ (\mathcal{R}' \setminus \{\ell\}, c') & \text{otherwise} \end{cases}.$$

Once garbage collection is done, if c is marked as honestly obtained then it is persisted to state by looking up $(c, \mathcal{S}) \leftarrow \mu(h_c)$ and updating ρ_a for $h = \mathsf{Keccak256}(h_c)$ as follows:

$$\rho_a(h) \coloneqq \begin{cases} (\mathcal{R} \cup \{\ell\}, c) & \text{if } \rho_a(h) = (\mathcal{R}, c) \\ (\{\ell\}, c) & \text{otherwise} \end{cases}.$$

The interaction between a smart contract and memory on an `SSTORE` opcode is illustrated in Figure 4.2. Note that the ciphertexts in $\mu(h_c)$ and in $\rho_a(h)$ are equal since h_c and h are computed from c and h_c, respectively, using a collision resistant hash function.

As shown above, lookups in ρ_a are done with keys that are derived by applying a preimage resistant hash function to a ciphertext handle. Doing so prevents malicious users from crafting handles that point to arbitrary positions in ρ_a and, in that way, forcing garbage collection of arbitrary ciphertext data.

The set $\mathcal{R}$ is used to prevent malicious users from invalidating already stored ciphertext handles by storing the bytes of the handle in a context where it is not honestly obtained and then overwriting it, forcing garbage collection. Since we don't update $\mathcal{R}$ if the handle is not honestly obtained, a subsequent garbage collection would not act on the malicious handle and, therefore, the invalidation of the original stored handle is prevented.

Finally, the `SLOAD` opcode is modified such that when a contract loads the value at location ℓ and $\rho_a(\mathsf{Keccak256}(\sigma_a(\ell))) = \rho_a(\mathsf{Keccak256}(h_c)) = (\mathcal{R}, c)$, c is marked as honestly obtained. This is shown in Figure 4.3.

4.5 Delegation

The `CALL` and `RETURN` opcodes are modified to automatically update μ when a contract passes a ciphertext handle to another.

When a `CALL` opcode is executed by a smart contract, every handle of ciphertext marked as honestly obtained is

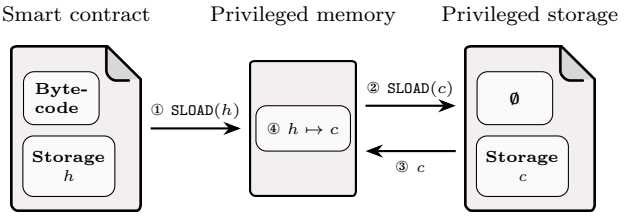

Figure 4.3: Interaction between a smart contract, privileged memory and privileged storage when calling the `SLOAD` opcode. The stack call depths are omitted.

extracted from the arguments. Let csd be the current call stack depth. For every extracted handle h, if $\mu(h) = (c, \mathcal{S})$ and $csd \in \mathcal{S}$, then $\mu(h) \coloneqq (c, \mathcal{S} \cup \{csd+1\})$, thereby marking c as honestly obtained for the callee contract.

Similarly, when a `RETURN` opcode is executed, every handle of honestly obtained ciphertexts is extracted from the arguments. Let csd be the current call stack depth. For every handle h found, if $\mu(h) = (c, \mathcal{S})$ and $csd \in \mathcal{S}$, then $\mu(h) \coloneqq (c, \mathcal{S} \cup \{csd - 1\})$. The intuition is that since the callee contract is explicitly passing the handle to the caller contract, the associated ciphertext should also be valid for the caller to use. Moreover, for all h' such that $\mu(h') = (c', \mathcal{S}')$, $\mu(h') \coloneqq (c', \mathcal{S}' \setminus \{csd\})$.

4.6 Operators

Precompiled contracts are added for arithmetic operators (`add`, `sub`, `mul`, `div`), logic operators (`and`, `or`, `xor`, `not`), comparison operators (`le`, `lt`, `ge`, `gt`, `eq`, `ne`, `min`, `max`), bit shift operators (`shl`, `shr`) and negation (`neg`). These all call the corresponding operation in the TFHE-rs library.

Let $h_1, \ldots, h_n$ be the inputs to the precompiled contract for any of these operations, let $(c_i, s_i) = \mu(h_i)$ be their associated ciphertext, and let csd be the current stack depth. If some h_i is not in the domain of μ, or if some s_i does not contain csd, then execution is reverted. Otherwise the ciphertext c returned by TFHE-rs after calling its function on $c_1, \ldots, c_n$ is marked as honestly obtained.

4.7 Decryption

When a decryption operation is encountered in smart-contract code, the behavior of validators and full nodes differ. Validators engage in a threshold decryption protocol with other validators in order to collectively decrypt the ci-

phertext as described in Section 7.4. However, full nodes cannot do this since they do not have a share of the secret key. To mitigate this, when executing transactions, validators store the resulting plaintext value on-chain after each decryption, and full nodes use these values when for instance performing block sync in order to reach the same state as validators. Full nodes may obtain these from a validator node or from another full node that is closer to the head of the chain.

Concretely, a contract address a_d is fixed and its storage is used to hold a mapping δ from handle h to plaintext value m. Note that this storage is never reclaimed since full nodes may want to catch up at any point in the future.

The fhEVM also exposes a precompiled contract for the decryption functionality. Its behavior depends on whether the node is currently a validator.

If it is a validator, then it triggers participation in an execution of the threshold decryption protocol. The result is a plaintext value m which is returned to the smart contract. If the validator is executing a transaction then m is also stored by updating $\delta(h_c) = m$. Note that each validator obtains the same plaintext message m so δ is consistent across the validators. Also, any existing entry in δ for h_c is overwhelmingly likely for the same ciphertext c, and in turn also with the same plaintext message m.

If the node is not a validator, then it obtains $m = \delta(h_c)$ from a node closer to the head of the chain, and assumes this to be the correct decryption. Note that the state consensus protocol will detect if this was not the value the validators committed to when they executed the threshold protocol.

Note that decryptions can also happen in non-committing function calls (e.g., view function calls). In this case, the query must be propagated to enough threshold protocol parties. Those parties will execute the threshold decryption protocol without persisting the decrypted value on-chain, and continue execution normally.

4.8 Optimistic require statements

The fhEVM exposes a precompiled contract for optimistic require statements. It maintains a list of pending encrypted booleans that it has received as inputs over the course of an execution. This list is checked either when an explicit decryption or re-encryption is requested, or at the very end of executing a transaction or view function call. This involves homomorphically computing the logical `and` operator across all items in the list and decrypting the resulting ciphertext as previously explained. If the result of the decryption is "false", execution is reverted—otherwise, it continues.

Optimistic require statements could reduce the number of decryptions per execution. However, the amount of gas used in case of reversal might be more than what it would have been in case of an early reversal. It is a trade-off developers might use to optimize for gas.

4.9 Gas estimation

Users sending transactions to an EVM blockchain with calls to smart contract functions would typically like to know the approximate gas amount (or just gas) required before actually signing and sending them. Gas depends on the number of operations executed in a call, how much data is persisted in smart contract state and so on. It could also depend on current state itself, because program flow can branch on contract state variables. Furthermore, state variables can change as blocks are added and, therefore, gas could vary from one block to another for the same call.

To get an approximate gas of the transaction before sending it, tools such as wallets use the gas estimation feature. One way of implementing it is to send the call to a single blockchain node that would execute it locally (potentially multiple times) to find the minimum amount of gas that is sufficient and return that to the user. Any state changes during gas estimation execution are reverted by the node and the user pays no fees.

Gas estimation in a typical EVM blockchain executes with the exact same input and, potentially, state as the actual transaction. That works, because all data on a blockchain is public.

In fhEVM, execution flow might depend on the decryption of encrypted values. However, decryption cannot happen during local gas estimation, because it requires interaction between validators as described in Section 7.4. Therefore, fhEVM introduces the following techniques:

- When a decryption is encountered, a constant plaintext value `d` is returned for any input ciphertext.
- Since there are no actual decryptions and flow cannot depend on ciphertexts, when an FHE operation is en-

countered, a unique ciphertext is generated as a result, independently from the inputs. That allows for execution to continue as if the operation was executed. This technique saves on unnecessary compute time.

- Ciphertexts are not persisted and garbage collected on an `SSTORE` opcode. This technique saves on input-output operations.

Above changes make gas estimation work for users in a very similar way to existing blockchains and EVM implementations. However, it is less precise when encrypted data is used as compared to only plaintext data.

4.10 Re-encryption

The fhEVM also exposes a re-encryption precompiled contract that takes as arguments a handle and a classical (non-FHE) public encryption key.[2] This function is not supported in transactions, so the fhEVM first checks that a view function is indeed being executed, and reverts if not. Since only validators hold a share of the key, this precompile should only be called on validators.

If the above conditions are satisfied, then it triggers participation in the threshold re-encryption protocol described in Section 7.5. This returns the validator's share of the plaintext, which is then encrypted under the user-provided (classical) public encryption key and signed by the validator's signature key.

5 FULLY HOMOMORPHIC ENCRYPTION

Allowing efficient and arbitrary computations on encrypted data had been an open research problem for a long time. By the late 00's it was believed by many to be infeasible. However, in 2009 a seminal result by Gentry [Gen09, Gen10] showed that such a scheme is indeed possible to construct. Gentry's scheme is based on an area of cryptography known as *lattice-based* cryptography. His overall idea starts with an encryption scheme which allowed a large amount of additions and a *few* multiplications to be computed over an encrypted message. The reason only a limited amount of operations are possible is due to each operation resulting in "noise" being added to the message encrypted, meaning that if too much noise is added, it would drown the actual message and thus lead to an incorrect decryption. To combat this Gentry introduced the idea of a "bootstrapping" stage which involves reducing the noise contained in the encrypted message. Conceptually it involves evaluating the decryption algorithm on a ciphertext, using an encryption of the secret decryption key, by leveraging the homomorphic properties of the scheme. Since decryption involves removing the noise, this procedure can be carried out indefinitely many times, allowing the construction of a fully homomorphic scheme.

Unfortunately, his initial scheme was quite inefficient. However, much research—in both industry and academia—has since then been carried out in this area, in order to construct schemes and protocols efficient enough for real-world applications, such as the public-key variant of the TFHE scheme [CGGI20, Joy23] we use in the fhEVM. One such efficiency improvement is *programmable bootstrapping* (PBS) [CJP21]. PBS is an extension to the bootstrapping technique, enabling the homomorphic evaluation of a (non-linear) univariate function as part of the bootstrapping process.

5.1 Black-box FHE

The fhEVM takes advantage of an FHE scheme through the TFHE-rs library [Zam22], which is an open source, Rust-based library developed by Zama. The library implements an efficient TFHE scheme, which we discuss in more detail in Section 5.2. It also implements algorithms to handle more advanced homomorphic operations, besides just addition and multiplication. This includes subtraction, bit-shifts, division, logical operations, and comparison. Furthermore, it allows composition of multiple ciphertexts to seamlessly emulate computation over integers of arbitrary bit-length. This is essential to allow for general computations, since typical FHE schemes, including the plain TFHE scheme, only works over messages of a few bits.

For completeness we sketch all of the essential methods required by an FHE scheme, which involves the typical public-key encryption methods, along with a method for computation, and methods allowing one to switch the key used to decrypt a ciphertext *without* decrypting the ciphertext first.

While the fhEVM relies fully on TFHE-rs, and is *black-box*

[2] Concretely, the `crypto_box` construction from `libsodium` is used for encryption, which internally relies on X25519 [Ber06] and XSalsa20-Poly1305 [Ber05].

in the underlying FHE scheme, it requires some of the general FHE methods to be implemented in a threshold manner in order to distribute the private key material over a set of multiple parties. Specifically KeyGen, Dec and KeySwitchGen need to be computed in a distributed fashion, which we discuss in more detail in Section 7. In this section we outline descriptions of these methods in the classical single-party (i.e. non-threshold) case.

$\mathsf{KeyGen}(\mathsf{sec}, \mathcal{M}) \to (pk, evk, sk)$: The key generation, where sec is the security parameter and $\mathcal{M}$ is the plaintext space, which is used to generate the following keys:

- pk : The public encryption key used to encrypt a message into a ciphertext;
- evk : The public evaluation key used to homomorphically evaluate an arbitrary function of ciphertexts (evk typically includes the bootstrapping material to control the noise in ciphertexts);
- sk : The private decryption key used to decrypt a ciphertext into the message it contains;

$\mathsf{Enc}_{pk}(m) \to c$: The encryption function which uses the public key pk to encrypt a message $m \in \mathcal{M}$ into a ciphertext c.

$\mathsf{Dec}_{sk}(c) \to m$: The decryption function which uses the private key sk to decrypt a ciphertext c into the corresponding plaintext $m \in \mathcal{M}$.

$\mathsf{Eval}_{evk}(f, c_1, c_2, \dots) \to c'$: The homomorphic evaluation of a function f, consisting of comparisons, arithmetic and logical operations, on ciphertexts c_1, c_2, ... s.t.

$$\mathsf{Dec}_{sk}\big(\mathsf{Eval}_{evk}(f, c_1, c_2, \dots)\big) = f(m_1, m_2, \dots)$$

where $m_1 \leftarrow \mathsf{Dec}_{sk}(c_1)$, $m_2 \leftarrow \mathsf{Dec}_{sk}(c_2)$, ...

$\mathsf{KeySwitchGen}(sk_1, sk_2) \to ksk$: Generates a public key-switching key ksk that can convert a ciphertext which can only be decrypted with key sk_1, to a ciphertext, which only private key sk_2 can decrypt.

$\mathsf{KeySwitch}_{ksk}(c) \to c'$: Converts a ciphertext c, which can only be decrypted with sk_1, into a ciphertext c', which can only be decrypted with sk_2, provided that $ksk = \mathsf{KeySwitchGen}(sk_1, sk_2)$.

$\mathsf{TrivialEnc}(m) \to c$: Encodes a message $m \in \mathcal{M}$ into a ciphertext representation. The result can be used in FHE computations, but is *not* secure.

5.2 The underlying TFHE scheme

Our FHE scheme is given by Joye [Joy23]. It is based on a variant of Learning With Errors (LWE) problem, first introduced by Regev [Reg05]; namely on the (polynomial) ring version of the decisional variant of the LWE problem. The (non-ring variant) decisional LWE problem consists in distinguishing samples drawn from the distributions $\{(\boldsymbol{a}, b = \langle \boldsymbol{a}, \boldsymbol{s}\rangle + e) \mid \boldsymbol{a} \xleftarrow{\$} \mathbb{Z}_q^\ell, e \leftarrow \mathcal{E}\}$ or $\{(\boldsymbol{a}, t) \mid \boldsymbol{a} \xleftarrow{\$} \mathbb{Z}_q^\ell, t \xleftarrow{\$} \mathbb{Z}_q\}$, for some error distribution $\mathcal{E}$ over $\mathbb{Z}$ and a secret vector $\boldsymbol{s}$. The problem is conjectured to be hard to solve, even for a quantum computer. The problem has been generalized to multiple mathematical structures such as polynomial rings or modules and used to construct multiple encryption schemes. In particular this includes the one used in the fhEVM (through TFHE-rs).

Concretely, the computation of the scheme in its most basic form can be specified as follows, letting $\circledast$ denote a certain vector convolution[3] and $\langle \cdot, \cdot \rangle$ denote the inner product:

$\mathsf{KeyGen}(\mathsf{sec}, \mathbb{Z}_p) \to (pk, evk, sk)$:

1. Based on sec define $\ell = 2^\eta$ for some $\eta > 0$ and select positive integers p, q s.t. $p \mid q$, let $\Delta = q/p$, and define an error distribution $\mathcal{E}$ over $\mathbb{Z}$ with standard deviation σ.[4] The plaintext space is $\mathcal{M} = \mathbb{Z}_p$.
2. Sample a uniformly random ℓ-bit vector as the secret key, $\boldsymbol{s} = (s_1, \dots, s_\ell) \xleftarrow{\$} \{0, 1\}^\ell$.
3. Sample a uniformly random vector $\boldsymbol{a_0} \xleftarrow{\$} \mathbb{Z}_q^\ell$.
4. Define $\boldsymbol{b_0} = \boldsymbol{a_0} \circledast \boldsymbol{s} + \boldsymbol{\epsilon} \in \mathbb{Z}_q^\ell$ where $\boldsymbol{\epsilon} \leftarrow \mathcal{E}^\ell$.
5. Let public parameters $pp = (\ell, p, q, \sigma, \Delta)$ and define the public key $pk = (\boldsymbol{a_0}, \boldsymbol{b_0})$ and the private key $sk = \boldsymbol{s}$.
6. The generation of the evaluation key evk (needed for the programmable bootstrapping functionality) is abstracted away for simplicity. See the original TFHE paper [CGGI20] or [Joy22, § 6.2] for details.

[3] Specifically, $\circledast$ denotes the *reverse negative wrapped convolution*. Given two vectors $\boldsymbol{u} = (u_1, \dots, u_\ell)$ and $\boldsymbol{v} = (v_1, \dots, v_\ell) \in \mathbb{Z}^\ell$, $\boldsymbol{w} := \boldsymbol{u} \circledast \boldsymbol{v} = (w_1, \dots, w_\ell) \in \mathbb{Z}^\ell$ with $w_i = \sum_{j=1}^{i} u_j\, v_{\ell+j-i} - \sum_{j=i+1}^{\ell} u_j\, v_{j-i}$. For example, $(1, 2, 3) \circledast (4, 5, 6)$ is the vector $(-17, 5, 32)$.

[4] The concrete numbers can be computed using a tool like the Lattice Estimator at https://github.com/malb/lattice-estimator.

7. Return (pk, evk, sk).

$\mathsf{Enc}_{pk}(m) \to c$: For a plaintext message $m \in \{0, \ldots, p-1\}$:

1. Parse $pk = (\boldsymbol{a_0}, \boldsymbol{b_0})$ and sample

$$\boldsymbol{r} \xleftarrow{\$} \{0,1\}^{\ell}, \quad \boldsymbol{\epsilon} = (\epsilon_1, \ldots, \epsilon_\ell) \leftarrow \mathcal{E}^{\ell}, \quad e \leftarrow \mathcal{E} \ .$$

2. Again setting $\Delta = q/p$, compute:

$$\text{mask } \boldsymbol{a} = \boldsymbol{a_0} \circledast \boldsymbol{r} + \boldsymbol{\epsilon} \in \mathbb{Z}_q^{\ell},$$
$$\text{body } b = \langle \boldsymbol{b_0}, \boldsymbol{r} \rangle + \Delta \cdot m + e \in \mathbb{Z}_q \ .$$

3. Output the ciphertext $c = (\boldsymbol{a}, b)$.

$\mathsf{Dec}_{sk}(c) \to m$:

1. Parse $sk = \boldsymbol{s}$.
2. Compute $m = \lceil (b - \langle \boldsymbol{a}, \boldsymbol{s} \rangle \mod q)/\Delta \rfloor \mod p$.

$\mathsf{TrivialEnc}(m) \to c$: For a plaintext message $m \in \{0, \ldots, p-1\}$: Output the (trivial) ciphertext $c = (\boldsymbol{0}, \Delta \cdot m)$.

Algorithms $\mathsf{Eval}_{evk}(f, c_1, c_2, \ldots) \to c'$, $\mathsf{KeySwitchGen}(sk_1, sk_2) \to ksk$ and $\mathsf{KeySwitch}_{ksk}(c) \to c'$ are abstracted away for simplicity. See [Joy22] for details.

Parameters. We outline the default parameters from TFHE-rs which we use in the fhEVM in Table 5.1. They are picked to ensure security equivalent to providing 128 bits of computational security, similar to AES-128. The smallest supported ciphertext will contain a 2-bit message-space and a 2-bit carry-space. Larger data types (see Section 3.2) are offered by TFHE-rs by bundling multiple of these smaller ciphertexts together and handling them accordingly. The user is not exposed to the underlying details and handles encrypted integer types seamlessly.

Table 5.1: Default parameters for the FHE scheme used [Joy23]. These ensure 128 bits of security and an error probability smaller than 2^{-40} per PBS.

Parameter	Explanation	Value
p	Plaintext modulus	2^4
q	Ciphertext modulus	2^{64}
ℓ	Vector dimension	2048
σ	Standard deviation of error	$2^{12.51}$

Table 5.2: Benchmark of typical FHE operations on encryptions of unsigned integers of various sizes. CPU times are multi-threaded using an Intel Xeon Gen 3 processor on AWS m6i.metal. .

Operation	Platform	`uint8`	`uint16`	`uint32`
Addition	CPU	78 ms	113 ms	141 ms
Subtraction	CPU	81 ms	111 ms	139 ms
Multiplication	CPU	127 ms	227 ms	368 ms
Division[5]	CPU	226 ms	322 ms	544 ms
Equality	CPU	40 ms	40 ms	65 ms
Comparison	CPU	58 ms	80 ms	110 ms
Bitshift	CPU	107 ms	138 ms	200 ms

Benchmarks. In Table 5.2 we outline the speed of the different FHE operations on differently sized unsigned integers and in Table 5.3 we outline the sizes of keys and ciphertexts based on the default parameters mentioned in the previous paragraph, when using TFHE-rs v. 0.3.1. When encrypting a plaintext of many bits, in is possible to use a *packing* technique to compress the bundle of ciphertexts resulting from such an encryption. This is achieved by cleverly sharing randomness between the different ciphertexts in the bundle. Packed ciphertext size grows only by a very small factor for increasing bit lengths, due to the use of packing as described in [Joy23, § 4]. However, we currently use unpacked ciphertexts after homomorphic operations have been carried out. We leave for future work the use of packed ciphertexts during operations. It is also possible to use other clever techniques to compress the storage space needed for ciphertexts as we discuss in Section 8.

For completeness we highlight that while the private and public keys are small, although the key needed for bootstrapping and key switching is around a hundred megabytes. However, these keys are generated just once. They are made public and do not need to be stored by clients.

6 ZERO-KNOWLEDGE PROOFS

In order to provide a blockchain solution offering both security and privacy for multiple users and smart contracts, it is essential to implement a flexible system that precludes misuse by both users and smart contract developers. In the setting of the fhEVM where values are encrypted, such require-

[5] Assumes a publicly known divisor.

Table 5.3: Sizes of the different FHE scheme elements. For ciphertexts, the values indicate the respective size of the encryption of unsigned integers `uint`$*$.

Keys		
pk	*sk*	*evk*
16.6 kB	256 B	112 MB

Ciphertexts					
Packed			Unpacked		
`uint8`	`uint16`	`uint32`	`uint8`	`uint16`	`uint32`
16.5 kB	16.5 kB	16.6 kB	65.8 kB	132 kB	263 kB

ments are exacerbated and much more complex to achieve than on plaintext values. While transferring and processing encrypted values protects the confidentiality of the underlying data, it is much harder to detect maliciously crafted erroneous messages, that could trigger undesired behavior or leak information about sensitive data, for example through so-called selective failure attacks [KS06]. Hence to prevent misuse the fhEVM relies heavily on zero-knowledge proofs, which can safely be posted and verified on the blockchain, in order to ensure admissible usage. By this it is ensured that data that gets sent as input into the fhEVM is well-formed and originates from the claimed sender.

As briefly described in Section 2.2 the ZKPoKs used in the fhEVM specifically ensure the following three properties: 1) the ciphertext is correctly formed, 2) the user knows the underlying plaintext message, and 3) the ciphertext cannot be used in another context.

Property 1 is generally required by the FHE scheme since decrypting malformed ciphertexts could leak something about the private decryption key.

Property 2 ensures that the users can neither reuse a ciphertext from someplace else, nor can they modify their input based on another ciphertext. This is particularly important in the blockchain setting since all ciphertexts are stored on-chain and therefore widely available. In more general cryptographic terms, this ensures *input independence*.

Finally, Property 3 intuitively prevents a malicious entity from reusing a proof (and in turn a ciphertext) in another context, for instance as an input to a smart contract they control. This is particularly important in our setting since smart contracts by design are free to do anything they want with ciphertexts sent to them, including having them decrypted or re-encrypted. This also illustrates that users must validate and trust a given smart contract before sending encrypted and confidential information to it, including any sub-contract it may call.

6.1 General realization of requirements in a proof-of-knowledge

To achieve the above-described properties a randomized non-interactive proof system is used and the randomness used depends on the identity of the user constructing the ciphertext and the smart contract receiving it as input. It is well-known that interactive proof systems consisting of one or more rounds of public-challenge/response steps between a prover and a verifier can be made non-interactive through the Fiat–Shamir transform [FS87]. That is, let x be public information and w be the witness for which the prover wants to prove knowledge, such that a public relation $R(x, w) = \top$ holds. Then if the interactive proof consists of multiple rounds of communication where the prover returns partial states $p_1, p_2, \ldots$ to the verifier and the verifier responds with public randomness $r_1, r_2, \ldots$ then it is possible to replace r_i with $\mathcal{H}(x, p_1, \ldots, p_{i-1}, r_1, \ldots r_{i-1})$ where $\mathcal{H}$ is a hash function modelled as a random oracle, making the proof non-interactive.

In general, a proof of knowledge can be specified with the following methods:

- $\mathsf{CRS} \rightarrow pp$: optional algorithm which generates the public parameters pp for the proof system. *Must* be executed either by a *trusted party* or through a secure protocol.

- $\mathsf{Prove}(pp, x, w) \rightarrow \pi$: takes as input a public input x and a witness w and construct a proof π that the prover knows a witness w s.t. $R(x, w) = \top$. Within this algorithm, the public randomness values $r_1, \ldots$ are instead computed as described above using $\mathcal{H}$ and the internal state of the proof.

- $\mathsf{Verify}(pp, x, \pi)$: may be executed by *any* verifier to validate that someone knows w s.t. $R(x, w) = \top$.

In our specific case, at the high level, x is the FHE public key pk and newly encrypted ciphertext $c \leftarrow \mathsf{Enc}_{pk}(m)$, and witness w is the message m. However, this is not completely sufficient as it also needs to be ensured that Verify not only enforces that *someone* knows w, but also who this is and in which context the proof happens. To achieve this the Fiat–Shamir heuristic is augmented with the necessary auxiliary information, a construction referred to as *strong Fiat–Shamir* in the literature [BPW12]. Specifically, let a_u be the user's address, a_c the smart contract's address and a_f the specific function that the user intends to call in the smart contract. Then public challenge $r_i \leftarrow \mathcal{H}(pp, x, a_u, a_c, a_f, p_1, \ldots, p_{i-1}, r_1, \ldots r_{i-1})$ is computed. This ensures that the two addresses and the specific function get entangled with the proof and thus that the verifier can validate that the proof is only used in the intended context, while at the same time proving knowledge of the witness w without revealing it.

Our primary goal for the specific choice of non-interactive zero-knowledge (NIZK) proof is compactness due to the need for posting it on the blockchain, while at the same time offering suitable performance in a blockchain context. For this reason the fhEVM uses the recent scheme of Libert [Lib23, § 5 and § G.3] which exactly follows the public randomness structure described above. A security analysis for this construction and our application of it is provided in [Sma23, § 2.4].

6.2 Setup

The fhEVM relies on the NIZK proofs from [Lib23], and thus require a common reference string (CRS). More specifically, during the global setup phase a "power-of-tau" random reference string needs to be generated, which is the same type of CRS required for many SNARK schemes, such as those used in Zcash. This will be done before the full commercial release, using a standard ceremony for CRS generation and the threshold protocols described in [NRBB22, KMSV21].

7 THRESHOLD PROTOCOLS

At various times during smart contract execution values need to be decrypted. As mentioned earlier, no single party knows the secret key that would allow decryption of values. Instead, this key is secret shared among the validators[6], who can interact in a threshold decryption protocol to decrypt and reconstruct the plaintext from a ciphertext. For this we use the protocols from [DDE+23], for which reasoning about the formal security and composability is provided in [Sma23].

7.1 Background

Throughout this section we use the same linear secret sharing scheme (LSSS) as in [DDE+23] with respect to a modulus Q and a corruption threshold $t < n$. The notation for a value $a \in \mathbb{Z}_Q$ as being secret shared across a set of n parties such that more than t parties are required to reconstruct the original value is:

$$[a]^{\langle t,Q\rangle} = ([a]_1^{\langle t,Q\rangle}, \ldots, [a]_n^{\langle t,Q\rangle})$$

where each party P_i has their corresponding share $[a]_i^{\langle t,Q\rangle}$.

Due to the linearity of the secret sharing scheme, given two secret shares $[a]^{\langle t,Q\rangle}$ and $[b]^{\langle t,Q\rangle}$, producing a secret sharing value of $\alpha \cdot a + \beta \cdot b + \gamma$ for any values $\alpha, \beta, \gamma \in \mathbb{Z}_Q$ can be done without interaction, i.e. each party P_i computes:

$$[\alpha \cdot a + \beta \cdot b + \gamma]_i^{\langle t,Q\rangle} \leftarrow \alpha \cdot [a]_i^{\langle t,Q\rangle} + \beta \cdot [b]_i^{\langle t,Q\rangle} + \gamma.$$

The same rules apply for the dot product operation $\langle \cdot, \cdot \rangle$ where one of the terms is public as it is a linear operation.

To recap, a TFHE ciphertext is of the form $(\boldsymbol{a}, b) \in \mathbb{Z}_q^\ell \times \mathbb{Z}_q$ with ciphertext modulus q, where

$$b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + \Delta \cdot m + e \pmod q$$

which encrypts a message $m \in \mathbb{Z}_p$, where p is the plaintext modulus, $\Delta = q/p$ is a scaling factor used to shift the message into the higher-order bits, e is some "noise" term, and ℓ is the LWE dimension and depends on the desired security. More details of the underlying encryption scheme can be found above in Section 5.2.

In order to mask the noise e coming from the decryption of $c = (\boldsymbol{a}, b) \in \mathbb{Z}_q^\ell \times \mathbb{Z}_q$, which might leak information about the secret key s, we apply the operation $c' \leftarrow \mathsf{Switch}\text{-}n\text{-}\mathsf{Squash}(c)$ from [DDE+23]. This operation converts a ciphertext c with LWE parameters (ℓ, q) to a ciphertext $c' = (\boldsymbol{a'}, b') \in \mathbb{Z}_Q^L \times \mathbb{Z}_Q$ with LWE parame-

[6] We here assume that the validators hold the secret shares and execute the threshold protocols, but one could alternatively imagine a separate set of parties for this task.

ters (L, Q) that encrypts the same plaintext m:

$$b' = \langle \boldsymbol{a'}, \boldsymbol{s'} \rangle + \Delta' \cdot m + e' \pmod{Q}$$

where $\boldsymbol{s'} \in \{0,1\}^L$ and $\Delta' = Q/p$. In practice, according to [DDE+23] the modulus q is set to $q = 2^{64}$ and modulus $Q = 2^{128}$.

In the threshold setting the secret key $\boldsymbol{s'}$ is secret shared modulo Q across a set of n parties with a threshold $t < n$ as $[s']^{\langle t,Q \rangle}$. More precisely, every bit of the secret key $s'_j \in \boldsymbol{s'}$ is secret shared separately, such that each party's share $[\boldsymbol{s'}]_i^{\langle t,Q \rangle}$ is a vector of L separate shares of the bits of $\boldsymbol{s'}$. The individual bits' shares are computed by evaluating for each bit $s'_{j \in [L]}$ a random polynomial F_j of degree t that has its zero-degree coefficient equal to s'_j; i.e., $F_j(0) = s'_j \in \mathbb{Z}_Q \quad \forall j \in \{1, \ldots, L\}$.

When a ciphertext $c = (\boldsymbol{a}, b) \in (\mathbb{Z}_q^\ell \times \mathbb{Z}_q)$ is input to be decrypted, the parties first apply $c' \leftarrow \mathsf{Switch}\text{-}n\text{-}\mathsf{Squash}(c)$ to get $c' = (\boldsymbol{a'}, b') \in (Z_Q^L \times \mathbb{Z}_Q)$ and then parties jointly generate a secret shared noise term $[E]^{\langle t,Q \rangle}$. Each party P_i computes a partial decryption

$$\begin{aligned} \mathsf{PDec}_i(c') &= b' - \langle \boldsymbol{a'}, [\boldsymbol{s'}]_i^{\langle t,Q \rangle} \rangle + [E]_i^{\langle t,Q \rangle} \\ &= [e' + E + \Delta' \cdot m]_i^{\langle t,Q \rangle} \ , \end{aligned}$$

where e is the error present in the encryption.

The variable $[E]^{\langle t,Q \rangle}$ is used to statistically mask e in order to avoid leaking information about the secret key $\boldsymbol{s}$; see [DDE+23] for more details. Note that the generation of $[E]^{\langle t,Q \rangle}$ can be deferred into an input-independent (preprocessing) phase since it does not depend on the input ciphertext $(\boldsymbol{a}, b)$.

Combining more than t valid $\mathsf{PDec}_i(c')$ values allows to reconstruct the contained message m in the clear. The $\mathsf{Switch}\text{-}n\text{-}\mathsf{Squash}(c)$ procedure takes about $\approx 300\,\text{ms}$ while the computation of $\mathsf{PDec}_i(c')$ and the reconstruction of m can be done in $\approx 2\,\text{ms}$ in a setting with $n = 4$ parties and threshold $t = 1$ according to [DDE+23].

7.2 Security model

Our protocols work with any number of parties n, as long as there are at most $t < n/4$ active corruptions, although it can also support $t < n/3$ for small values of $\binom{n}{t}$. The protocol falls under the umbrella of the so-called robust MPC frameworks, which means that even if a fraction of t parties lie about their shares or send malformed values then the protocol still continues and the honest parties get the correct output.

We assume that the communication channels between parties are encrypted[7] and authenticated and that an adversary $\mathcal{A}$ can corrupt at most t players. For the preprocessing phase we assume that the communication channels are synchronous as these are needed to resolve disputes, using mechanisms similar to [DN07], where parties need to broadcast conflicts.

In a synchronous network every round has a time-out value δ. Thus if a party sends a message in a round it is expected to be received within time frame δ by the receiving party. We use a slightly modified version of Bracha's broadcast [Bra87][8] where a sender S wishes to broadcast a message m to all parties that gives us the following guarantees:

- Every honest party will terminate with either a message m, or a value $\perp$ (different from any possible m).
- Any two honest parties that terminate, output the same value, and if the sender S is honest then that message is m.

The synchronous broadcast protocol allows us to assume that all honest parties will send, and receive, the expected messages from all other honest parties within the δ time limit. If some messages are not received within that time-step, then the parties simply move on without them. Thus the protocol will output $\perp$ if the sender sends inconsistent values but also if the sender sends nothing at all.

Without delving into specifics, we believe that a synchronous preprocessing phase coupled with an asynchronous on-line phase is a good compromise, as the decryption parties running the protocols usually have a given time-frame δ to complete each protocol step, and running a decryption for days would be impractical. Another reason for which we chose the synchronous variant of Bracha [Bra87] and not the asynchronous broadcast [CP23, DYX+22, DXR21] is that the synchronous broadcast leads to a simpler implementation.

[7] Note that this is only needed for the offline phase, but to get authenticated channels you might as well use encrypted channels.

[8] See https://hackmd.io/@alxiong/bracha-broadcast for a discussion of this.

During the on-line phase we can assume asynchronous channels where messages can be arbitrarily (but finitely) delayed [BKR94, BTH07, BCG93]. The on-line phase consists of retrieving some secret material generated in the preprocessing phase to which we simply call the robust open protocol (see Figure 7.2 and Figure 7.3) from [DDE+23]. Malicious players are detected during calls to robust reconstruction, which we describe in detail in Section 7.7.

Depending on the number of parties we add various optimizations, the fastest protocol being when $\binom{n}{t}$ is small. The system currently supports the on-line phase and was extensively benchmarked using $n \in \{4, 10, 40\}$ parties with various threshold sizes; see [DDE+23].

Our threshold protocols focus solely on data that is being processed in smart contracts or blockchain transactions, and do not directly modify how consensus is achieved or how transactions are processed. This means that attacks on them can only affect the confidentiality of processed data, but cannot influence the consensus mechanism of the blockchain. An attacker targeting the threshold protocols cannot steal tokens or forge signatures, as these functionalities are secured independently by the blockchain itself.

7.3 Distributed key generation for TFHE

In our setting no single entity knows the secret key so the process of generating such a shared secret key $[\boldsymbol{s'}]^{\langle t,Q\rangle}$ can be done through an MPC protocol. The problem of performing asynchronous distributed key generation has been extensively studied in many works [KHG12, DYX+22, KMS20, DXR21]. Due to the difficulty of adapting existing protocols to the ring case, we take the distributed BGV key-generation method from [RST+22] for the dishonest majority case with abort and replace its generic functionalities with robust MPC. The case for robust key generation follows immediately from the dishonest majority case. Next, we explain only how the secret and public key is generated for TFHE using tools from [RST+22] as well as the auxiliary keys to perform the FHE operations such as bootstrapping and key-switching keys.[9]

[9] We observe that key-switching is needed for technical reasons to increase efficiency. Specifically allowing decryption with a sufficiently large noise term $[E]^{\langle t,Q\rangle}$ requires a choice of a modulus $Q \gg q$. As a result, before decryption, ciphertexts are key-switched from a modulus q to a larger domain Q in which the threshold decryption is carried out. For more details see [DDE+23].

The secret key $\boldsymbol{s'}$ for distributed decryption is a random binary vector $\boldsymbol{s'} \in \{0,1\}^L$, which can be generated by simply calling the shared random bit GenBit protocol L times described in [DDE+23, Fig. 20]. Thus the secret key share is represented as $[\boldsymbol{s'}]^{\langle t,Q\rangle} = ([s'_1]^{\langle t,Q\rangle}, \ldots, [s'_L]^{\langle t,Q\rangle})$, where each $[s'_j]^{\langle t,Q\rangle}$ is a random secret shared bit.

To generate the public key we need to compute something reminiscent of encryptions of zero based on the secret key. Concretely $(\boldsymbol{a}, \boldsymbol{a} \circledast \boldsymbol{s} + \boldsymbol{\epsilon})$ for a random $\boldsymbol{a} \in \mathbb{Z}_q^\ell$ where $\boldsymbol{\epsilon}$ is a vector of small error terms from the normal distribution with variance σ^2. Note here that the LWE parameters are (q, ℓ) can be obtained by generating the samples using the larger LWE parameters (Q, L) by 'down-casting'. The downcast is done by defining $\boldsymbol{s}$ to be the first (for eg.) ℓ bits of $\boldsymbol{s'}$ and truncate the convolution $(\boldsymbol{a} \circledast \boldsymbol{s} \bmod Q) \bmod q$. Generating $\boldsymbol{a}$ is easy since parties only need to agree to a public random seed which is then input into a PRG and expanded in a pseudo-random manner to generate all random elements $a_j \in \mathbb{Z}_q$ that constitute $\boldsymbol{a} = (a_1, \ldots, a_\ell)$.

The main difficulty lies in generating a secret shared $\boldsymbol{\epsilon}$. Fortunately sampling numbers according to the normal distribution can be achieved by generating $\approx \sigma/2$ shared random bits; see [RST+22, Fig. 11].

The bootstrapping keys are ring-GSW encryptions of bits of the secret key $\boldsymbol{s}$ under another secret key $\hat{\boldsymbol{s}}$ [Joy22]. Key-switching keys are symmetric-key encryptions of bits of $\hat{\boldsymbol{s}}$ under the secret key $\boldsymbol{s}$ which are scaled by known scalars B^i. Both the bootstrapping and key-switching keys can be computed using secret shared bits of $[\boldsymbol{s}]^{\langle t,Q\rangle}$ and $[\hat{\boldsymbol{s}}]^{\langle t,Q\rangle}$ in the same manner as described for the public key above. Hence the tools needed are the same as those needed to generate the public key.

7.4 Decryption requests

Decryption requests are initiated by a smart contract execution calling the decryption precompiled contract. Note that if a view function call is being executed, it must be gossiped to all other validators in order to get (enough of) them to participate in the distributed decryption protocol. The precompiled contract then calls out of the fhEVM to trigger the local decryption party P_i into participating in an execution of the distributed decryption protocol using its key share $[\boldsymbol{s'}]_i^{\langle t,Q\rangle}$. Throughout the protocol, parties broadcast

their partial decryption $\mathsf{PDec}_i(c)$ (equivalent to a sharing of the plaintext m) such that when the protocol finishes, all parties know the decrypted value m. Parties independently return this value to their local fhEVM instance, which resumes execution.

The behavior of full nodes on decryption requests is detailed in Section 4.7.

7.5 Re-encryption requests

As briefly sketched in Section 2.4, when a user U asks to view some encrypted value, e.g. their balance $c \leftarrow \mathsf{Enc}(m)$, a request is sent to the decryption parties $\{P_1, \ldots, P_n\}$ to execute the distributed decryption protocol on $c' = \mathsf{Switch\text{-}}n\mathsf{\text{-}Squash}(c)$. At the end of the protocol the parties will have a share of the result $[m]^{\langle t,Q\rangle}$ and will send that share to user U.

More precisely, the user generates a classical and possibly ephemeral public key pk, signs it using its signature key, and sends it in a call to a view function to a single node, called gateway G, which in turn forwards the user request to the set of all current validators. Upon reception of this view function call, validators locally execute it in the fhEVM to retrieve the ciphertext to decrypt c' as well as to make sure that according to the smart contract code, the user has the right to ask for a re-encryption of this ciphertext. This local execution eventually leads to the re-encrypt precompile which calls out of the fhEVM to trigger the local decryption party P_i into executing the interactive distributed decryption protocol using its key share $[\boldsymbol{s'}]_i^{\langle t,Q\rangle}$ to produce the preprocessing material required to compute $\mathsf{PDec}_i(c')$. As a result, each node can now compute a partial decryption $\mathsf{PDec}_i(c')$ of the requested plaintext output m. Each party now encrypts their partial decryption under the user-provided key and sign with their individual signature key before they return it to the user via G. The calling user can then reconstruct the plaintext as described in Section 7.7. The flow of re-encryption requests is depicted in Figure 7.1. Note that the gateway G can be a separate entity who acts as a proxy with the single task of relaying messages, but this task could also be done by one of the decryption parties or validators themselves.

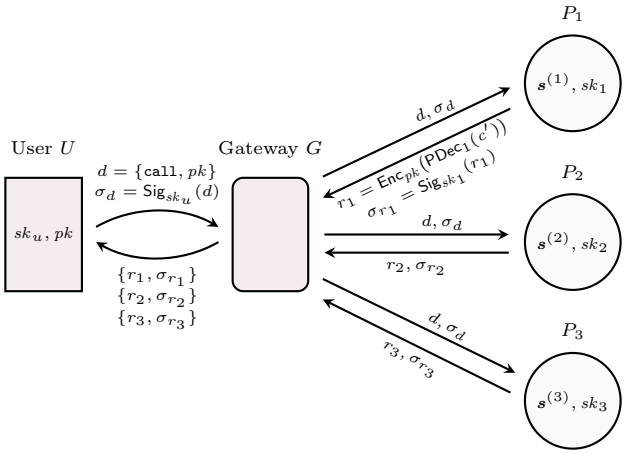

Figure 7.1: View function communication flow between user U and the threshold decryption parties.

7.6 Implementation of decryption requests

Distributed decryptions happen in a specific and configurable time frame, which we call an *epoch* e, in which the set of decryption parties and thus parties for the distributed decryption are fixed. In order to process incoming ciphertext decryption requests in epoch e for a specific shared secret key $[\boldsymbol{s_e}]^{\langle t,Q\rangle}$ the parties need to run our threshold decryption software which acts as a gRPC server to deal with multiple and concurrent requests.

Each request (e, c) is a tuple containing the epoch number e and the ciphertext to be decrypted c. The decryption parties will then derive a session id $\mathsf{sid} \leftarrow \mathcal{H}(e\|c)$ where $\mathcal{H}$ is a cryptographic hash function and all inputs to the hash function are assumed to have a unique encoding.

Parties hold a (bounded) channel map per sid for each incoming message. The messages sent around between players have an sid tag attached to them and are locally dispatched to its corresponding sender and session id. Since the dispatch map contains bounded channels it's impossible for malicious parties to mount attacks that flood up the honest parties' storage.

Once the threshold decryption protocol terminates, the parties will store the outputs o into a map $\mathsf{sid} \mapsto o$, which can be queried afterwards.

RobustOpen ($t < n/4$)

This protocol depends on the ratio between t, and n, and whether the underlying network is synchronous or asynchronous. It can be run by a player $\mathcal{P} = \mathcal{P}_i$ who already holds their share or by an external player $\mathcal{P}$. The output reconstruction happens on the wallet side, hence we describe the external player reconstruction.

RobustOpen$(\mathcal{P}, [a]^{\langle t,Q\rangle})$ where $4\,t < n$:

1. Player $\mathcal{P}_i$ sends $[a]_i^{\langle t,Q\rangle}$ securely to player $\mathcal{P}$.
2. $\mathcal{P}$ waits until they have received $3\,t + 1$ share values $\{[a]_j^{\langle t,Q\rangle}\}_j$.
3. They apply a Reed–Solomon error correction algorithm like Berlekamp–Welch to the $3\,t + 1$ shares they hold to robustly compute a.
4. Return a.

Figure 7.2: Robust opening protocol for asynchronous network when $t < n/4$.

7.7 Robust reconstruction

After the user has received a sufficient number of signed and encrypted shares they can locally run the reconstruction of the contained plaintext value. The amount of incoming shares depends on the number of parties n and the corruption threshold t. We detail the protocol from [DDE+23] where we consider the case that the network is asynchronous depending on the threshold t: when $t < n/4$ in Figure 7.2 and $t < n/3$ in Figure 7.3. Note the full robust open protocol in [DDE+23] also works with synchronous network with timeouts. Another feature of the robust opening procedure is that it also allows the external user to identify parties that have sent an incorrect share. This could later be connected with the proof of stake mechanism to incentivize honest behavior and slash malicious actions, but we currently leave this as an open problem.

7.8 Rotating decryption parties

With each new epoch e the set of decryption parties can change. For this case we devise a mechanism to reshare the secret key from the decryption party set $\mathcal{S} = \{P_1, \ldots, P_n\}$ to a new group of parties $\mathcal{S}' = \{P'_1, \ldots, P'_m\}$. We visualize the

RobustOpen ($t < n/3$)

RobustOpen$(\mathcal{P}, [a]^{\langle t,Q\rangle})$ where $3\,t < n$:

1. Player $\mathcal{P}_i$ sends $[a]_i^{\langle t,Q\rangle}$ securely to player $\mathcal{P}$.
2. For $r = 0, \ldots, t$ do :
   (a) $\mathcal{P}$ waits until $2\,t + r + 1$ shares have been received.
   (b) Apply Reed–Solomon error correction on the $2\,t + r + 1$ shares, assuming there are r errors in these shares.
   (c) If error correction outputs a degree t polynomial then output a.

Figure 7.3: Robust opening protocol for asynchronous network when $t < n/3$.

general concept abstractly in Figure 7.4. In the literature this problem is called as *dynamic-committee proactive secret sharing* (DPSS) [SLL08, MZW+19, BGG+20, YXXM23].

In our case the main challenge is to make DPSS work over a ring $\mathbb{Z}_q$ since all existing constructions make use of the secret $\boldsymbol{s}$ being an element in a prime field.

To reshare $\boldsymbol{s}$ from a set $\mathcal{S}$ to a set $\mathcal{S}'$, the parties $P_i \in \mathcal{S}$ share their shares $[\boldsymbol{s}]_i^{\langle t,Q\rangle}$ to the parties in $P_j \in \mathcal{S}'$ using a VSS (Verifiable Secret Sharing) scheme [GIKR01, CCP21]. From these shares $[[\boldsymbol{s}]_i^{\langle t,Q\rangle}]_j^{\langle t,Q\rangle}$ the underlying syndrome polynomial [Hal15, GTLBNG21] is computed in secret shared form (which is a linear operation). Then this syndrome is used to locally construct any error vector introduced by adversaries in $\mathcal{S}$. Given the error vector the parties in $\mathcal{S}'$ can correct, locally, the shares they just received. Then by applying the recombination vector for the sharing in $\mathcal{S}$ to the shares of the shares, the parties in $\mathcal{S}'$ can construct a sharing of the original secret $\boldsymbol{s}$.

8 FUTURE WORK

8.1 Dynamic validator set

By default, we assume the blockchain's validators also act as decryption parties in the threshold protocols. This means that the threshold-decryption parties must change whenever the set of validators changes. We leave as future work ensuring that the re-sharing algorithm described in Section 7.8

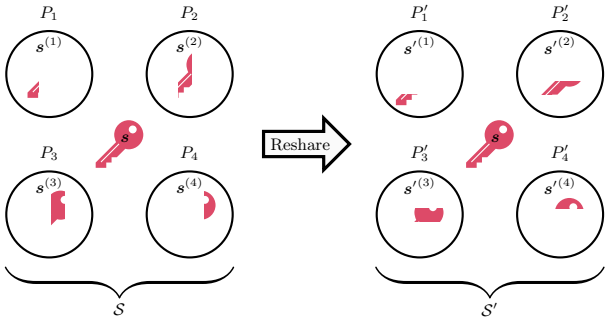

Figure 7.4: Conceptual visualization of resharing from a set of validators $\mathcal{S}$ to a new set of validators $\mathcal{S}'$, holding different shares of the same secret key s.

is executed honestly; i.e., that validators delete their secret share after having participated in the protocol. One potential way to achieve this would be for validators to keep the key share in a secure enclave and execute all MPC protocols inside of it.

Note that one could also consider a separate and dedicated set of decryption parties, which only follow the classical threshold assumptions and do not engage in the consensus protocol.

8.2 Ciphertext size

We leave as future work to find a way to reduce the amount of on-chain storage space taken by TFHE ciphertexts.

One way to do so would be to use *transciphering*. Evaluating symmetric-key cryptographic primitives homomorphically can be used to transcipher data encrypted under a FHE scheme to and from a symmetric-key encryption scheme, thereby removing the blowup in size of ciphertexts. Trivium can be used to transcipher TFHE ciphertexts efficiently [BOS23]. By using this technique, keeping encrypted data in on-chain storage would have no overhead compared to storing the associated plaintext data.

REFERENCES

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

[BCDF23] Carsten Baum, James Hsin-yu Chiang, Bernardo David, and Tore Kasper Frederiksen. Eagle: Efficient privacy preserving smart contracts. In Foteini Baldimtsi and Christian Cachin, editors, *FC 2023: 27th International Conference on Financial Cryptography and Data Security*, Lecture Notes in Computer Science, Bol, Croatia, May 1–5, 2023. Springer, Heidelberg, Germany. To appear. Preprint available as Cryptology ePrint Archive, Report 2022/1435 at https://ia.cr/2022/1435.

[BCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *25th Annual ACM Symposium on Theory of Computing*, pages 52–61, San Diego, CA, USA, May 16–18, 1993. ACM Press. doi:10.1145/167088.167109.

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

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

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

[Ber05] Daniel J. Bernstein. The poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption – FSE 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49, Paris, France, February 21–23, 2005. Springer, Heidelberg, Germany. doi:10.1007/11502760_3.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228, New York, NY, USA, April 24–26, 2006. Springer, Heidelberg, Germany. doi:10.1007/11745853_14.

[BGG+20] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part I*, volume 12550 of *Lecture Notes in*

Computer Science, pages 260–290, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-64375-1_10.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In Jim Anderson and Sam Toueg, editors, *13th ACM Symposium Annual on Principles of Distributed Computing*, pages 183–192, Los Angeles, CA, USA, August 14–17, 1994. Association for Computing Machinery. doi:10.1145/197917.198088.

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

[BOS23] Thibault Balenbois, Jean-Baptiste Orfila, and Nigel P. Smart. Trivial transciphering with Trivium and TFHE. Cryptology ePrint Archive, Report 2023/980, 2023. https://eprint.iacr.org/2023/980.

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 626–643, Beijing, China, December 2–6, 2012. Springer, Heidelberg, Germany. doi:10.1007/978-3-642-34961-4_38.

[Bra87] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

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

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

[BTH07] Zuzana Beerliová-Trubíniová and Martin Hirt. Simple and efficient perfectly-secure asynchronous MPC. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 376–392, Kuching, Malaysia, December 2–6, 2007. Springer, Heidelberg, Germany. doi:10.1007/978-3-540-76900-2_23.

[CCP21] Anirudh C, Ashish Choudhury, and Arpita Patra. A survey on perfectly-secure verifiable secret-sharing. Cryptology ePrint Archive, Report 2021/445, 2021. https://eprint.iacr.org/2021/445.

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

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann, editors, *Cyber Security, Cryptology, and Machine Learning (CSCML 2021)*, volume 12716 of *Lecture Notes in Computer Science*, pages 1–19, Be'er Sheva, Israel, July 8-9, 2021. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-78086-9_1.

[CL06] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 78–96, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany. doi:10.1007/11818175_5.

[CP23] Ashish Choudhury and Arpita Patra. On the communication efficiency of statistically secure asynchronous MPC with optimal resilience. *Journal of Cryptology*, 36(2):13, April 2023. doi:10.1007/s00145-023-09451-9.

[CS97] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups (extended abstract). In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO'97*, volume 1294 of *Lecture Notes in Computer Science*, pages 410–424, Santa Barbara, CA, USA, August 17–21, 1997. Springer, Heidelberg, Germany. doi:10.1007/BFb0052252.

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

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

[DDE+23] Morten Dahl, Daniel Demmler, Sarah Elkazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Samuel Tap, and Michael Walter. Noah's ark: Efficient threshold-FHE using noise flooding. Cryptology ePrint Archive, Report 2023/815, 2023. https://eprint.iacr.org/2023/815.

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

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg,

Germany. doi:10.1007/978-3-540-74143-5_32.

[DXR21] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2705–2721, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press. doi:10.1145/3460120.3484808.

[DYX+22] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew K. Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *2022 IEEE Symposium on Security and Privacy*, pages 2518–2534, San Francisco, CA, USA, May 22–26, 2022. IEEE Computer Society Press. doi:10.1109/SP46214.2022.9833584.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany. doi:10.1007/3-540-47721-7_12.

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

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press. doi:10.1145/1536414.1536440.

[Gen10] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, 2010. doi:10.1145/1666420.1666444.

[GIKR01] Rosario Gennaro, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. The round complexity of verifiable secret sharing and secure multicast. In *33rd Annual ACM Symposium on Theory of Computing*, pages 580–589, Crete, Greece, July 6–8, 2001. ACM Press. doi:10.1145/380752.380853.

[GTLBNG21] José Gómez-Torrecillas, Francisco Javier Lobillo Borrero, and Gabriel Navarro Garulo. Decoding linear codes over chain rings given by parity check matrices. *Mathematics*, 9(16):1878, 2021. doi:10.3390/math9161878.

[Hal15] Jonathan I. Hall. Notes on coding theory. Lecture Notes, 2015. URL: https://users.math.msu.edu/users/halljo/classes/codenotes/coding-notes.html.

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

[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.

[Joy23] Marc Joye. TFHE public-key encryption revisited. Cryptology ePrint Archive, Report 2023/603, 2023. https://eprint.iacr.org/2023/603.

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

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press. doi:10.1109/SP.2019.00002.

[KHG12] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. Cryptology ePrint Archive, Report 2012/377, 2012. https://eprint.iacr.org/2012/377.

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

[KMS20] Eleftherios Kokoris-Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1751–1767, Virtual Event, USA, November 9–13, 2020. ACM Press. doi:10.1145/3372297.3423364.

[KMSV21] Markulf Kohlweiss, Mary Maller, Janno Siim, and Mikhail Volkhov. Snarky ceremonies. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 98–127, Singapore, December 6–10, 2021. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-92078-4_4.

[KS06] Mehmet S. Kiraz and Berry Schoenmakers. A protocol issue for the malicious case of Yao's garbled circuit construction. In *Proceedings of 27th Symposium on Information Theory in the Benelux*, pages 283–290, 2006. URL: https://www.win.tue.nl/~berry/papers/wic06.pdf.

[Lib23] Benoît Libert. Vector commitments with short proofs of smallness. Cryptology ePrint Archive, Report 2023/800, 2023. https://eprint.iacr.org/2023/800.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck

and Adrienne Porter Felt, editors, *USENIX Security 2018: 27th USENIX Security Symposium*, pages 973–990, Baltimore, MD, USA, August 15–17, 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

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

[MZW+19] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. CHURP: Dynamic-committee proactive secret sharing. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 2369–2386, London, UK, November 11–15, 2019. ACM Press. doi:10.1145/3319535.3363203.

[NRBB22] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. Powers-of-tau to the people: Decentralizing setup ceremonies. Cryptology ePrint Archive, Report 2022/1592, 2022. https://eprint.iacr.org/2022/1592.

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

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

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

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

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 84–93, Baltimore, MA, USA, May 22–24, 2005. ACM Press. doi:10.1145/1060590.1060603.

[RST+22] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. *Journal of Cryptology*, 35(1):5, January 2022. doi:10.1007/s00145-021-09416-w.

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

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

[SLL08] David A. Schultz, Barbara Liskov, and Moses Liskov. Mobile proactive secret sharing. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *27th ACM Symposium Annual on Principles of Distributed Computing*, page 458, Toronto, Ontario, Canada, August 18–21, 2008. Association for Computing Machinery. doi:10.1145/1400751.1400856.

[Sma23] Nigel P. Smart. Practical and efficient FHE-based MPC. Cryptology ePrint Archive, Report 2023/981, 2023. https://eprint.iacr.org/2023/981.

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

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

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

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

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

[Woo21] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Yellowpaper, 2021. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

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

[YXC+18] Rui Yuan, Yu-Bin Xia, Hai-Bo Chen, Bin-Yu Zang,

and Jan Xie. ShadowEth: Private smart contract on public blockchain. *Journal of Computer Science and Technology*, 33:542–556, 2018. doi:10.1007/s11390-018-1839-y.

[YXXM23] Thomas Yurek, Zhuolun Xiang, Yu Xia, and Andrew Miller. Long live the honey badger: Robust asynchronous DPSS and its applications. In Joe Calandrino and Carmela Troncoso, editors, *USENIX Security 2023: 32nd USENIX Security Symposium*, pages 5413–5430, Anaheim, CA, USA, August 9–11, 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/yurek.

[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.