Private voting with Fully Homomorphic Encryption

Confidential voting made easy

Ryan Orendorff

2025

Who is Sunscreen?

SUNSCREEN

Ultra-fast, easy-to-use Fully Homomorphic Encryption infrastructure

1. We build Fully Homomorphic Encryption (FHE) tooling, including a compiler and processor.
2. Our mission is to make data privacy easy and accessible for every developer.
3. We believe FHE is an integral part of the future of data privacy.

Outline

- Basics of Fully Homomorphic Encryption
- Building voting systems: from binary to ranked choice
- Integrating with blockchain using SPF
- Live demonstration
- Other FHE applications

Basics of Fully Homomorphic Encryption

Normally, a server needs to see your data to process it

While data is often encrypted in transit from a user to a server, the server must decrypt that data to be able to process a request.

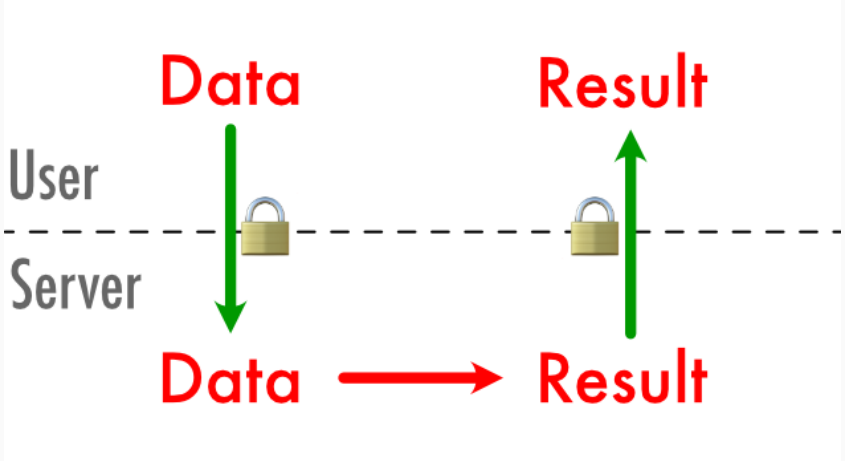


Figure 1: Server processing

This represents a data privacy concern; the server operator could mishandle your data.

FHE allows computation on encrypted data without decryption

Fully Homomorphic Encryption (FHE) enables servers to perform computations on encrypted data without ever decrypting it.

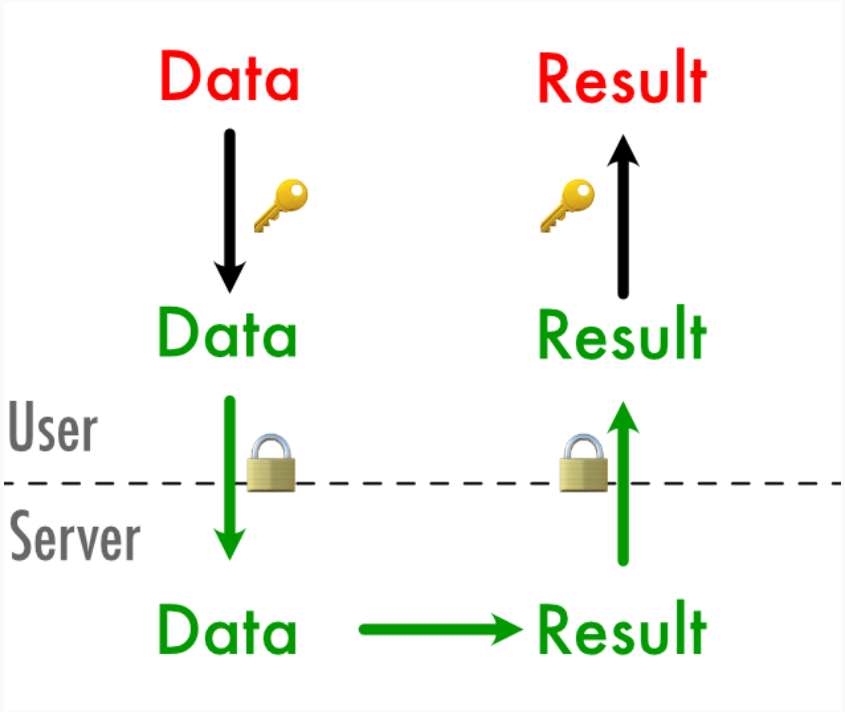


Figure 2: FHE processing

The server can never read your data.

How FHE accomplishes this: base operations

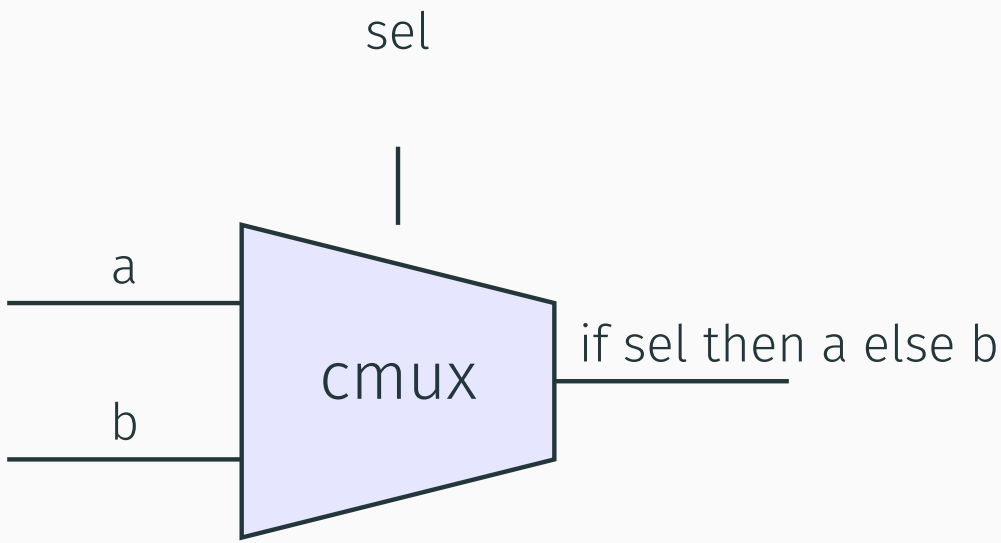
FHE schemes support two primary operations on ciphertexts that have the following properties (homomorphisms):

- Addition: $\mathrm{Enc}(a) + \mathrm{Enc}(b) = \mathrm{Enc}(a + b)$
- Multiplication: $\mathrm{Enc}(a) \cdot \mathrm{Enc}(b) = \mathrm{Enc}(a \cdot b)$

The term ‘fully’ means it can compute any function on encrypted data.

Torus FHE enables efficient comparison operations

Sunscreen has its own variant of the Torus FHE scheme, which allows for efficient comparison operations as well.



Using this computation allows us to build out FHE-based circuits, much like one would do with standard computer hardware.

So what can we compute with TFHE?

Sunscreen's variant of TFHE supports the following 64-bit operations on plaintext or ciphertext data.

- Arithmetic operations (add, sub, mul)
- Logical operations (and, or, not)
- Bit operations (shifts)
- Comparison operations (equal, less than, greater than)
- Branching (cmux)
- Arrays

The Sunscreen stack: a modular FHE compute engine

The Sunscreen TFHE stack is a complete compute system. Most importantly, this stack includes:

1. A compiler that converts C code to FHE programs.
2. An off-chain compute service for high-performance blockchains and dApps.

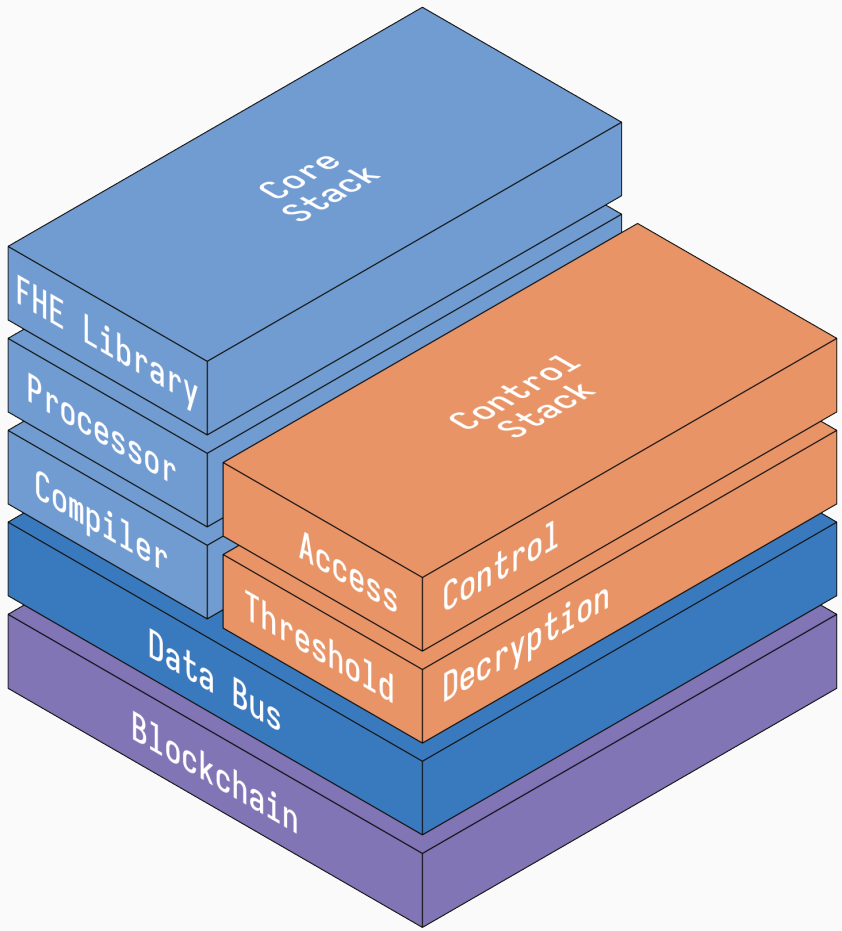


Figure 3: Secure Processing Framework

Why is voting a good FHE application?

Private voting is an ideal application for FHE because it requires:

- Ballot secrecy: votes must remain confidential during counting
- Verifiable execution: the election process must be auditable (through deterministic outputs)
- Complex computation: tallying requires data validation and transformation

FHE enables all of these properties simultaneously.

Let’s build voting systems with FHE!

Binary voting: choosing if an issue passes or fails

Our first voting scheme will be binary voting, where voters choose whether to accept or reject an issue.

The basic process for tallying these votes is to

- Collect encrypted yes/no votes from voters
- Tally the votes homomorphically
- Determine if the issue passes based on majority

Binary voting: the plaintext code

This specifies whether an issue passes if the majority of voters vote “yes”.

```
#include <parasol.h>



void binary_vote_plain(uint8_t *voter_choices,
                       uint16_t num_voters,
                       bool *issue_passes) {
    uint16_t tally = 0;
    for (uint16_t i = 0; i < num_voters; i++) {
        // Coerce to boolean (0 or 1)
        tally += (bool) voter_choices[i];
    }
    *issue_passes = tally > (num_voters / 2);
}
```

Binary voting: the encrypted code

This specifies whether an issue passes if the majority of voters vote “yes”.

```
#include <parasol.h>

[[clang::fhe_program]]
void binary_vote([[clang::encrypted]] uint8_t *voter_choices,
                 uint16_t num_voters,
                 [[clang::encrypted]] bool *issue_passes) {
    uint16_t tally = 0;
    for (uint16_t i = 0; i < num_voters; i++) {
        // Coerce to boolean (0 or 1)
        tally += (bool) voter_choices[i];
    }
    *issue_passes = tally > (num_voters / 2);
}
```

Quadratic voting: binary voting with more numbers

Binary voting can be extended to quadratic voting by allowing users to cast n votes that have a quadratic cost.

```
[[clang::fhe_program]]
void quadratic_vote([[clang::encrypted]] int16_t *voter_choices,
                    [[clang::encrypted]] uint16_t *voter_credits,
                    uint16_t num_voters,
                    [[clang::encrypted]] bool *issue_passes) {
    int32_t tally = 0;
    for (uint16_t i = 0; i < num_voters; i++) {
        bool valid = is_valid_quad(voter_choices[i], voter_credits[i]);
        tally += iselect16(valid, voter_choices[i], 0);
    }
    *issue_passes = tally > 0;
}
```

Binary voting: a good first start

We can now choose between two options privately!

This method can be extended to quadratic voting, where voters can assign values beyond zero or one.

However, what if we wanted to run a more complex election with multiple candidates?

First past the post: choosing a candidate from multiple options

In first past the post (FPTP) voting, each voter selects one candidate from a list of options.

This requires a few more pieces on top of binary voting:

- Verifying voter choices.
- Converting the data into something that can be tallied.
- Finding the winner among the final tally.

First past the post: encoding the vote

We want to tally the votes for each candidate. To facilitate this, we will convert each voter's choice into an array with a single 1 in it.

```
inline void encode_ballot(
    [[clang::encrypted]] uint8_t choice,
    uint8_t num_choices,
    [[clang::encrypted]] uint8_t *encoded_ballot) {
    uint8_t is_valid_ballot = choice < num_choices;

    for (uint8_t i = 0; i < num_choices; i++) {
        // We turn invalid votes into ballots of all zeros.
        encoded_ballot[i] = select8(choice == i, is_valid_ballot, 0);
    }
}
```

First past the post: tally helpers

We need some helpers for handling the tally process.

```
inline void initialize_tally([[clang::encrypted]] uint16_t *tally,
                             uint8_t num_choices) {
    for (uint8_t choice = 0; choice < num_choices; choice++)
        tally[choice] = 0;
}

inline void add_to_tally([[clang::encrypted]] uint8_t *ballot,
                         [[clang::encrypted]] uint16_t *tally,
                         uint16_t num_choices) {
    for (uint8_t choice = 0; choice < num_choices; choice++)
        tally[choice] += ballot[choice];
}
```

First past the post: tallying the votes

Tallying the votes involves converting all ballots to the encoded form and then summing all ballots.

```
inline void tally_votes([[clang::encrypted]] uint8_t *ballots,
                        uint16_t num_ballots, uint8_t num_choices,
                        [[clang::encrypted]] uint16_t *tally,
                        [[clang::encrypted]] uint8_t *encoded_ballot) {
    initialize_tally(tally, num_choices);
    for (uint8_t i = 0; i < num_ballots; i++) {
        encode_ballot(ballots[i], num_choices, encoded_ballot);
        add_to_tally(encoded_ballot, tally, num_choices);
    }
}
```

First past the post: deciding the winner

As the last step, we can find the winner from the final tally by selecting the candidate with the most votes.

```
inline void determine_winner([[clang::encrypted]] uint16_t *tally,
                             uint8_t num_choices,
                             [[clang::encrypted]] uint8_t *winner) {
    *winner = 0; uint16_t max_votes = tally[0];

    for (uint8_t choice = 1; choice < num_choices; choice++) {
        bool is_greater = tally[choice] > max_votes;

        max_votes = select16(is_greater, tally[choice], max_votes);
        *winner = select8(is_greater, choice, *winner);
    }
}
```

First past the post: full election circuit

We can now call all our functions in one main program.

```
[[clang::fhe_program]]
void fptp_election([[clang::encrypted]] uint8_t *ballots,
                   uint16_t num_ballots, uint8_t num_choices,
                   [[clang::encrypted]] uint16_t *tally,
                   [[clang::encrypted]] uint8_t *encoded_ballot,
                   [[clang::encrypted]] uint8_t *winner) {
    tally_votes(
        ballots, num_ballots, num_choices, tally, encoded_ballot);
    determine_winner(tally, num_choices, winner);
}
```

First past the post: full elections with validations

With first past the post, we saw

- Data validation.
- Converting between data structures (int and array) for counting.
- Use of scratch buffers (must be passed in).

Can we take this one step further and enable people to rank their preferences instead of just picking one?

Ranked choice voting: ranking candidates by preference

In ranked choice voting, each voter ranks candidates in order of preference. Points are assigned based on the rank, and the candidate with the highest total points wins.

```
ballot = [
  0, // Give 0 points to the first candidate
  3, // Give 3 points to the second candidate
  2, // Give 2 points to the third candidate
  1  // Give 1 point to the fourth candidate
]
```

Ranked choice voting: similar to FPTP with different ballots

We can implement ranked choice voting with the Borda method.

```
[[clang::fhe_program]]
void borda_election([[clang::encrypted]] uint8_t *ballots,
                    uint16_t num_ballots, uint8_t num_choices,
                    [[clang::encrypted]] uint16_t *tally,
                    [[clang::encrypted]] uint8_t *decoded_ballot,
                    [[clang::encrypted]] uint8_t *winner) {
    initialize_tally(tally, num_choices);
    for (uint16_t i = 0; i < num_ballots; i++) {
        decode_borda_ballot(
            &ballots[i * num_choices], num_choices, decoded_ballot);
        add_to_tally(decoded_ballot, tally, num_choices);
    }
    determine_winner(tally, num_choices, winner);
}
```

Ranked choice voting: what did we learn?

Ranked choice voting is a demonstration of more complex voting methods.

Specifically, it also shows that you can build up *a library of composable FHE programs* that can be reused in other scenarios (`add_to_tally`, `determine_winner`, etc).

Are any voting systems off-limits?

One strongly desired property of a voting system is called the *Condorcet condition*, which states that *the candidate who wins the most head-to-head matchups should win the overall election*.

- Majority rule: more than half of the voters are satisfied with the result
- Reduces the effect of spoiler candidates
- Encourages candidates to appeal to a broader electorate

The Borda method shown can be extended to be a Condorcet election method in FHE using the Copeland election scheme.

What did we learn about voting with FHE?

We have seen FHE implementations of core components of a voting scheme.

- FHE can handle complex data validation and transformation.
- Many voting systems can be implemented homomorphically, including sophisticated methods like the Copeland method.
- FHE preserves voter privacy throughout the process.

How do we use FHE to bring these private voting methods to web3?

Voting onchain with the Secure Processing Framework (SPF)

What we've built so far

- Binary voting: simple majority decisions
- First past the post: multi-candidate selection with validation
- Borda count: ranked preference with bit-encoded ballots

These are standalone FHE programs. To make them useful onchain, we need infrastructure.

Secure Processing Framework (SPF) brings FHE onchain

The Secure Processing Framework (SPF) is Sunscreen's solution for integrating FHE into blockchain environments.

- An oracle listening to FHE operations onchain.
- A data storage system for encrypted data.
- An access control system to manage who can run programs and decrypt results.

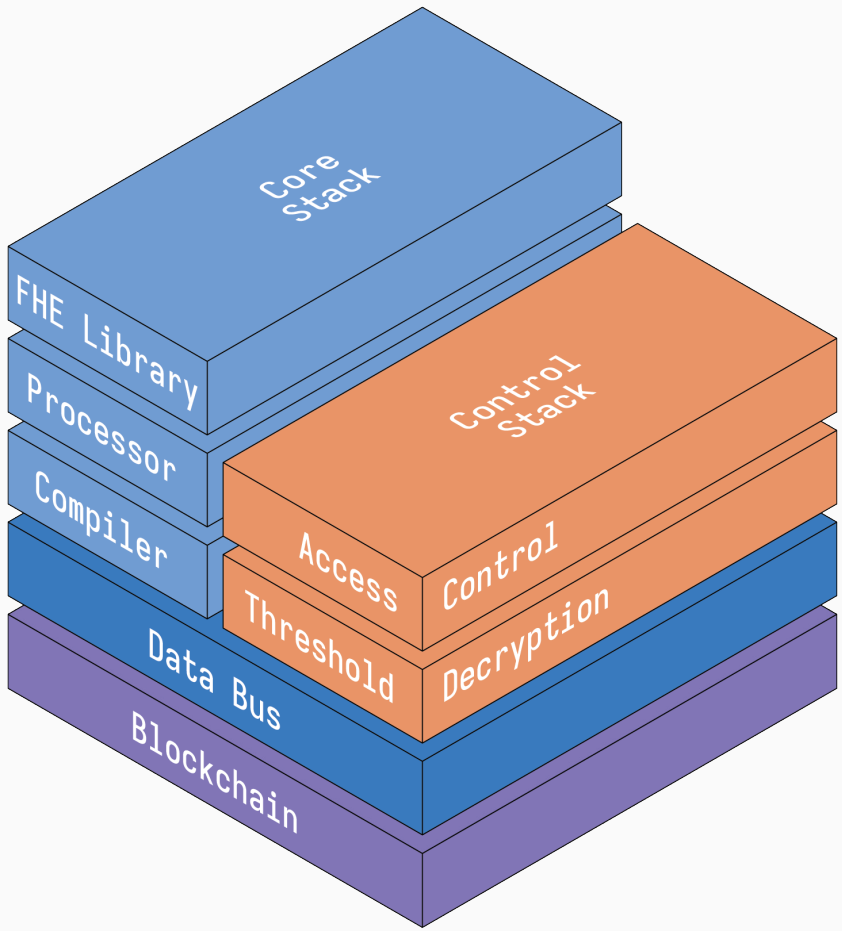


Figure 4: Secure Processing Framework

SPF allows developers to call FHE applications from smart contracts

1. Write an FHE application using Sunscreen's compiler
2. Write a smart contract to call the FHE program
3. Upload the FHE application to the SPF service
4. Deploy the smart contract to the blockchain
5. SPF listens for FHE events
6. SPF calls contract callback to post results

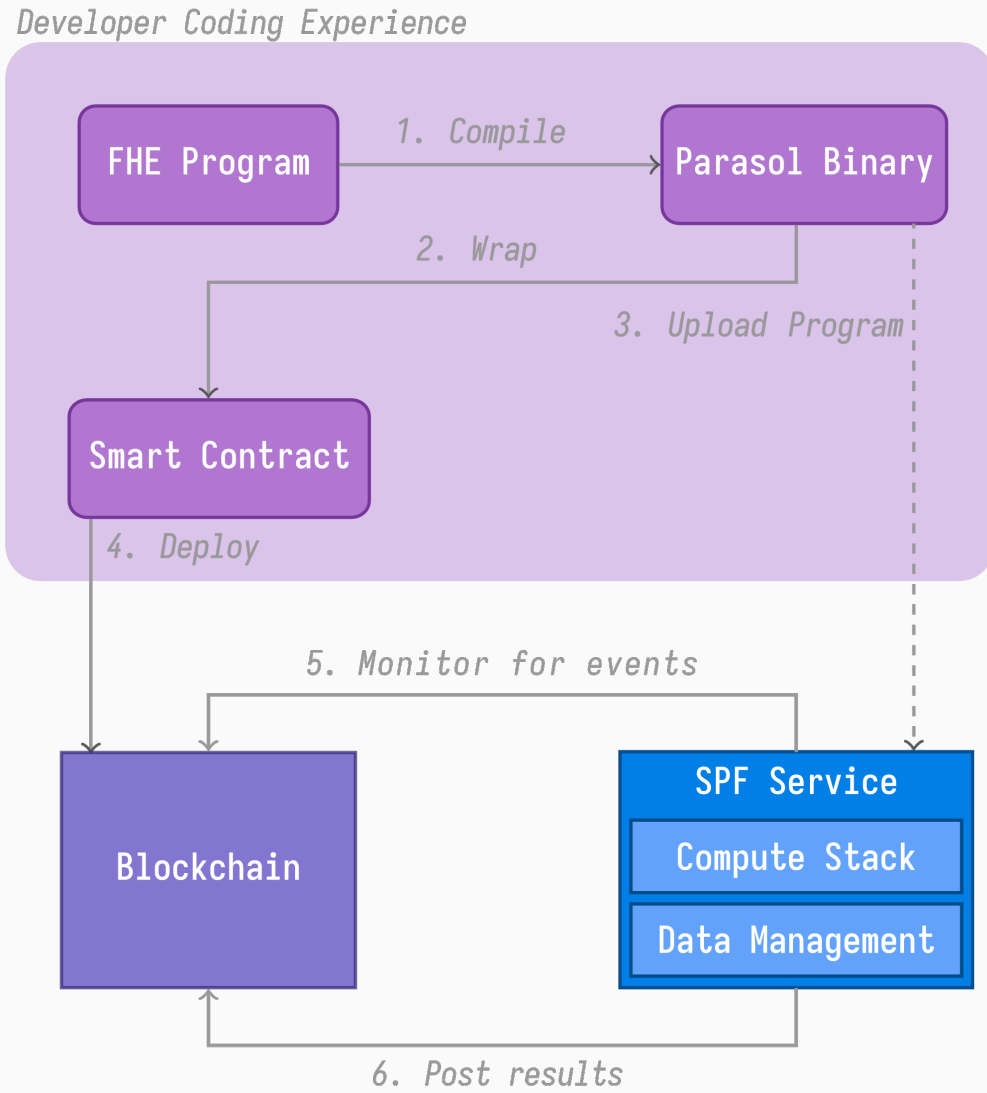


Figure 5: Developer workflow in web3

To make a smart contract with FHE, we need to know a few things

We need to know the following information:

1. The function signature in C.
2. A callback function for how to handle the results.

Let's wrap the binary voting example

We will build up the contract for binary voting in parts. First, we define some state for our contract.

```
import "@sunscreen/contracts/Spf.sol";
import "@sunscreen/contracts/TfheThresholdDecryption.sol";

contract BinaryVoting is TfheThresholdDecryption {
  Spf.SpfLibrary public constant VOTING_SPF_LIBRARY =
      Spf.SpfLibrary.wrap(hex"edd540489dac8e6dab39c9f99aa6a3fc"
                          hex"100c96899e772286800b0bd1ac6479ee");
  Spf.SpfProgram public constant VOTING_PROGRAM =
      Spf.SpfProgram.wrap("binary_vote");

  Spf.SpfParameter[] public votes;
  bool public issuePassed;
```

Users provide votes through their ciphertext identifier

This returns an identifier that the user can use to reference their data onchain.

```
function submitVote(Spf.SpfParameter calldata vote) public {
    Spf.SpfParameter memory updatedVote =
      Spf.senderAllowCurrentContractRun(
        vote, VOTING_SPF_LIBRARY, VOTING_PROGRAM);
    votes.push(updatedVote);
}
```

Packing votes together

Remember we have this syntax for our binary vote program.

```
void binary_vote([[clang::encrypted]] uint8_t *voter_choices,
                 uint16_t num_voters,
                 [[clang::encrypted]] bool *issue_passes);
```

We can tell Solidity how to pack these parameters into the run request.

```
function packParameters() internal view
  returns (Spf.SpfParameter[] memory) {
    return Spf.pack3Parameters(
        Spf.createCiphertextArrayFromParams(votes),
        Spf.createPlaintextParameter(16, votes.length),
        Spf.createOutputCiphertextParameter(8));
}
```

Running the binary voting program

We can then run the program:

```
function run() public {
    Spf.SpfParameter[] memory parameters = packParameters();

    // Request the FHE program be run off-chain
    Spf.SpfRunHandle runHandle = Spf.requestRunAsContract(
        VOTING_SPF_LIBRARY, VOTING_PROGRAM, parameters);
    Spf.SpfParameter memory issuePasses = Spf.getOutputHandle(
        runHandle, 0);

    // Request the FHE result be posted onchain
    requestDecryptionAsContract(this.postResult.selector,
        Spf.passToDecryption(issuePasses));
}
```

Decryption

When the binary voting computation is complete, the SPF oracle will post the results back onchain through the specified callback.

```
function postResult(bytes32 identifier, uint256 _issuePassed)
    public onlyThresholdDecryption {
    issuePassed = (_issuePassed != 0);
}
```

The `identifier` is passed in to allow the contract to keep track of which value is being decrypted in case multiple values could be returned.

SPF enables onchain voting

The SPF service enables onchain private voting using FHE.

- Ballot secrecy: FHE keeps the ballots private during the tally process.
- Verifiable execution: the coordinating smart contract is auditable, and the FHE program being run is known by a verifiable identifier.
- Auditability: computations can be rerun by anyone (but not decrypted) to verify the output ciphertext is correct.

Demo time!

Private voting

Let's walk through binary voting as a dApp.

Figure 6: https://voting-demo.sunscreen.tech

Beyond voting

The techniques demonstrated in voting—data validation, transformation, and conditional logic on encrypted data—enable private computation across many domains.

Uses for Sunscreen FHE stack

FHE for private auctions

FHE can be used to build private auctions:

- First/second price sealed bid auctions
- Bid for priority to some resource
- MEV auctions: private bids prevent frontrunning and protect block builder strategies

FHE for private payments

FHE can enable private ERC20 and other token payments.

```
[[clang::fhe_program]]
void balance_transfer([[clang::encrypted]] uint64_t *sender_balance,
                      [[clang::encrypted]] uint64_t *recipient_balance,
                      [[clang::encrypted]] uint64_t transfer_amount,
                      [[clang::encrypted]] uint64_t zero) {
    // Set invalid transfer amount to 0
    bool has_sufficient = transfer_amount ≤ *sender_balance;
    uint64_t transfer_value = has_sufficient ? transfer_amount : zero;

    // Update balances
    *recipient_balance += transfer_value;
    *sender_balance -= transfer_value;
}
```

FHE for data transformations

FHE enables complex data transformations to happen privately.

- Private recommendation engines based on encrypted user preferences
- Confidential credit scoring and applicant ranking
- Private ML inference

FHE for private database lookups (PIR)

FHE can be used to make a database where lookups are private; the database operator does not know what someone looked for.

- Search
- Financial services (screening)
- Caller ID
- Visual search

FHE powers a private and verifiable blockchain ecosystem

What we've demonstrated:

- Complete voting systems: from C code to onchain execution
- Privacy preservation: ballot secrecy throughout the entire process
- Verifiability: auditable smart contracts and replayable computations

The same principles extend to auctions, payments, ML inference, and database queries—any computation requiring confidentiality with verification.

Questions?

Figure 7: https://spf-docs.sunscreen.tech

SPF documentation

Figure 8: https://docs.sunscreen.tech

Parasol documentation