

General

Welcome

Seismic is a stack for encrypted blockchains



If you're an entrepreneurial developer who finds Seismic interesting, we'd love to work with you. We're actively looking for collaborators on apps in the pipeline for our upcoming devnet. Our founder's personal telegram is [@lyronc](#).

Seismic is the encrypted blockchain. It has encryption built into the base layer, allowing developers to access new token bootstrapping models, consumer payment flows, rwa markets, and more. To our knowledge, Seismic is the first chain to focus entirely on encrypting protocols to produce new apps.

If you're learning about Seismic, we suggest

1. Reading through [why](#), [what](#), and [how](#).
2. Understanding [core concepts](#).

If you're building on Seismic, we suggest

1. Stepping through [installation](#) and [quickstart](#).
2. Referencing [prototype contracts](#).
3. Skimming our [viem docs](#).
4. Messaging [@lyronc](#) on TG.
5. Asking questions in our [developer TG group](#).

introduction

Why

A vision-centric overview of Seismic

Ethereum launched with two key problems: limited scalability and forced transparency. Limited scalability made every transaction slow and expensive. Forced transparency left every transaction exposed and scrutinized.

We've solved scalability, but not transparency. Every major L1 today is still transparent. None are encrypted.

Why? Because for the last decade, our industry optimized for encryption at the wallet level to provide user privacy. It enabled actions like purchasing goods without exposing balances or speaking without exposing identities. Though crucial for personal freedom, this focus on new wallets came at the cost of producing new apps. Without new apps, our industry slowed down.

So how do we get new apps? Encrypt at the protocol level instead of at the wallet level.

To achieve this, encryption must be built into the core—the base layer—not just attached to the edges of a transparent chain. That's why we created Seismic, the encrypted blockchain.

What

A product-centric overview of Seismic

Seismic is the encrypted blockchain. It has all the attributes you need to encrypt protocols:

1. Encrypted global state. Enables encrypted interactions between multiple users. Essential for everything from exchanges to lenders.
2. Encrypted memory access. Enables encrypted pointers. Essential for everything from auctions to stablecoins.
3. Encrypted data flow. Enables controlled exchange between encrypted and transparent state. Essential for everything from organizations to launchpads.

By building encryption into the base layer, Seismic lets developers access new token bootstrapping models, consumer payment flows, rwa markets, and more. And unlike previous attempts at base layer encryption, Seismic focuses entirely on encrypting protocols to produce new apps.

How

A tech-centric overview of Seismic

An open-source stack

We're restructuring the modern blockchain stack around secure hardware. The major components are as follows:

1. The language is a fork of [solidity](#). We added `stype`.
2. The execution client is a fork of [reth](#), [revm](#), and [alloy](#). We added encrypted storage and relevant opcodes.
3. The consensus middleware is [omni](#). We used it off-the-shelf.
4. The consensus client is [cometbft](#). We used it off-the-shelf.
5. The secure hardware build is a fork of [yocto](#) manifests from flashbots. We added proxies.
6. The testing framework is a fork of [foundry](#). We added encrypted storage, along with the relevant opcodes.
7. The wallet client is an extension of [viem](#). We added transaction types.

Notice, 99% of our stack is code written by the OS community. We're making sure to maintain this standard, which is why [all of our repositories](#) are **fully open-source under an MIT License**.

Built around secure enclaves

At the core of this stack is the secure enclave. The term describes a set of hardware components that provide confidentiality measures over data in use, protecting it from being read by any outside entity, including the host machine. Our secure enclave of choice is Intel TDX.

Our system uses TDX by cloning all the primary memory segments of the EVM. This results in a set of encrypted segments and a set of transparent segments, where

the former can leverage the confidentiality properties of the underlying hardware.

Data flows between these segments with cloned storage opcodes. For example, the vanilla EVM has `SLOAD` / `SSTORE` to manage elements in storage, and our EVM adds `CLOAD` / `CSTORE` to manage elements in encrypted storage. The same pattern is applied to calldata, transient storage, and memory.

These segments allow us to know whether every element added to the stack is considered transparent or encrypted. Then it becomes a matter of tracking these elements to enforce rules over how they interact and how they should be handled by the execution loop.

Our github repositories hold the v0 implementation for this. Like every v0, it's a far cry from the final spec. Notably, **the current implementation only clones the storage segment and does not keep track of element status in the stack. It compensates by treating all memory segments as encrypted.** This heavy handed approach has major drawbacks in both UX and liveness. However, it's enough to test our core hypothesis around encrypting protocols while decreasing our time to market, which makes it a great fit for our first release.

With these trust assumptions

Our dependency on secure enclaves comes with strong trust assumptions. The most prominent one is on hardware confidentiality, which leaves us wary of [side-channel attacks](#). Our short-term mitigation for this is the restriction to cloud-based validators.

Though this assumption has led to [many issues](#) in the past, we're cautiously optimistic about the most recent generation of VM-based enclaves like Intel's TDX and AMD's SEV-SNP. They function with an untrusted hypervisor and patch up major flaws present in the first generation.


We believe these enclaves are sufficient to support entire blockchain ecosystems. And with the current momentum behind confidential compute, we expect the rate of improvement only increases from here.

onboarding

Installation

Everything you need to run Surface on your local machine

sforge, sanvil, and ssolc

 We currently support devices running MacOS with ARM architecture.

The local development suite uses `sforge` as the testing framework, `sanvil` as the local node, and `ssolc` as the compiler.

1. Install [`rust` / `cargo` / `brew` / `jq`] on your machine if you don't already have them. Default installations for all work well.

```
# install rust and cargo
curl https://sh.rustup.rs -sSf | sh

# install brew
bash -c "$(curl -fsSL raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
source ~/.zshenv # or ~/.bashrc or ~/.zshrc

# install jq
brew install jq
source ~/.zshenv # or ~/.bashrc or ~/.zshrc
```

2. Download and execute the sfoundryup installation script.

```
curl -L \
  -H "Accept: application/vnd.github.v3.raw" \
  "https://api.github.com/repos/SeismicSystems/seismic-foundry/content"
source ~/.zshenv # or ~/.bashrc or ~/.zshrc
```

4. Install `sforge`, `sanvil`, `ssolc`. Expect this to take between 5-20 minutes depending on your machine.

```
sfoundryup
source ~/.zshenv # or ~/.bashrc or ~/.zshrc
```

5. Remove old build artifacts in existing projects.

```
sforge clean # run in your project's contract directory
```

VSCode extension

We also recommend adding syntax highlighting via the `seismic` extension from the VSCode marketplace.

Quickstart

You're two commands away from running a shielded contract

You can play around with `stype` using our [starter repository](#). This assumes you went through everything in [Installation](#).

```
git clone "https://git@github.com/SeismicSystems/seismic-starter.git"
cd seismic-starter/packages/contracts
sforge test -vv
```

core

Basics

A handle on `stype` unlocks all shielded computation and storage

Mental model



We assume familiarity with [Solidity](#).

Developers communicate to Seismic through the `stype`. A thorough understanding of this one concept unlocks all shielded computation and storage. The `stype` consists of three elementary types:

- `suint` / `sint`: shielded integer
- `sbool`: shielded boolean
- `saddress`: shielded address

The primary difference between them and their vanilla counterparts is that they're shielded. Any operations you apply to them are carried out as expected, but the values won't be visible to external observers.

There are special considerations unique to each individual type. These are covered in the next three sections. For now, we'll develop a general understanding of `stype` that applies to all its component types.

Here's the mental model you should have for shielded contracts. Whenever a tx is broadcasted by a user, it goes through the same submission, execution, and storage phases as a tx in a regular blockchain. The only difference is that when you look at the tx at these different stages- whether it's as a calldata payload during submission, a trace during execution, or as leaves in the MPT tree during storage- any bytes that represent `stype` variables are replaced with `0x000`.

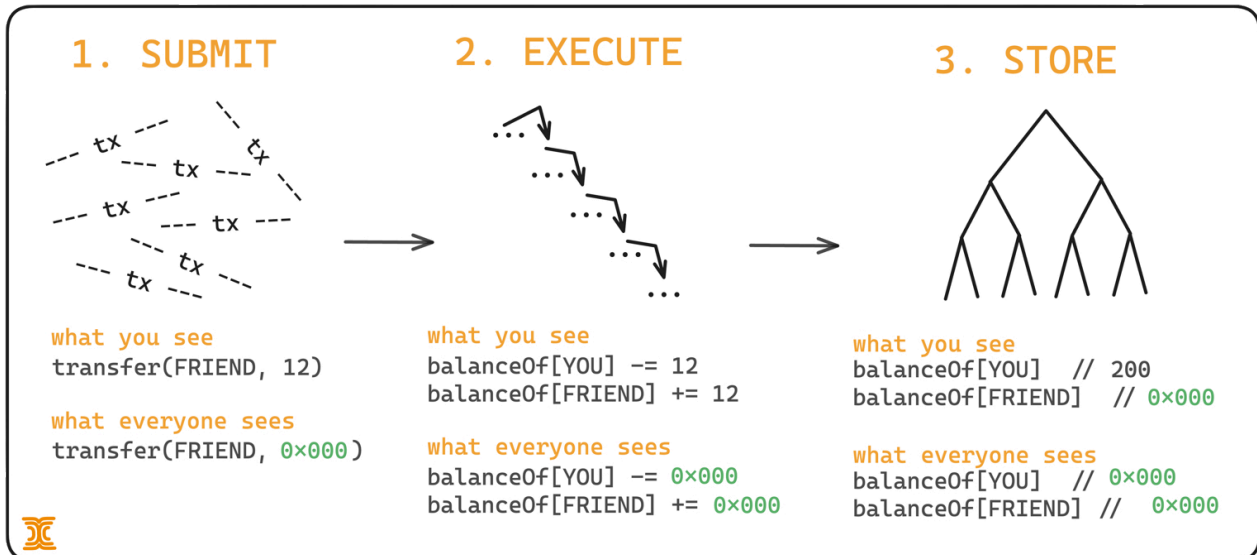
Let's step through a concrete example. We'll follow the lifecycle of a `transfer()` tx for an `ERC20` variant. This variant shields user balances and transfer amounts:

```

mapping(address => uint256) public balanceOf; // shielded balance

function transfer(address to, uint256 amount) public { // shielded tran
    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;
}


```



Observers see 0x000 in place of state variables during transaction submission, execution, and storage.

Shielding user balances is done by changing the values of the `balanceOf` array to `uint256`. Shielding transfer amounts is done by changing the `amount` parameter in `transfer()` to `uint256`. Now we can see what happens at every stage of the tx lifecycle:

1. Submit. The tx is sitting in the mempool. You know that you're sending 12 tokens to your friend. Observers can look at the calldata and figure out that your friend is the recipient, but will see `0x000` instead of the number 12.
2. Execute. The tx is processed by a full node, and its trace is open. You know that 12 tokens were removed from your balance and 12 were added to your friend's. Observers know that the same number that was deducted from your balance was added to your friend's, but they see `0x000` instead of the number 12.
3. Store. The effects of the tx are applied to the state tree of all full nodes. You know that your new balance goes down by 12, to 200. You know that your friend's balance went up by 12, but you only see `0x000` for what its final state is. Observers know that your new balance is down the same amount that your friend's new balance is up, but they see `0x000` for both balances.

 Surface currently shields a lot more than just the bytes representing `stype` variables, so the above model is more granular than you technically need to be. However, this will soon stop being the case. You should not fit your contracts to this temporary discrepancy.

Casting

You can cast `stype` variables to their unshielded counterparts, and vice-versa. Only explicit casting is allowed- no implicit. Note that whenever you do this, observers can look at the trace to figure out either the initial (if going from not `stype` to `stype`) or final (if going from `stype` to not `stype`) value.

```
uint256 number = 100;  
suint256 sNumber = suint256(number);
```

Restrictions

There are two restrictions in how you can use `stype` variables:

1. You can't return them in `public` or `external` functions. This also means `stype` contract variables can't be `public`, since this automatically generates a getter. If you want to return one, you'll have to cast it into its unshielded counterpart.

```
/*  
 * Throws a compiler error  
 */  
suint256 public v;  
  
// =====  
  
/*  
 * Throws a compiler error  
 */  
function f() external view returns (suint256) {}
```

2. You can't use them as constants.

```
/*  
 * Throws a compiler error  
 */  
suint256 constant MY_CONSTANT = 42;
```


suint / sint

shielded unsigned integer / shielded integer

All comparisons and operators for `suint` / `sint` are functionally identical to `uint` / `int`. The universal casting rules and restrictions described in [Basics](#) apply.

```
suint256 a = suint256(10)
suint256 b = suint256(3)
```

```
// == EXAMPLES
a > b // true
a | b // 11
a << 2 // 40
a % b // 1
```

saddress

shielded address

An `saddress` variable has all `address` operations supported. As for members, it supports `call`, `delegatecall`, `staticcall`, `code`, and `codehash` *only*. It also does not support `saddress payable`.

The universal casting rules and restrictions described in [Basics](#) apply.

```
saddress a = saddress(0x123);  
saddress b = saddress(0x456);
```

```
// == VALID EXAMPLES
```

```
a == b // false
```

```
b.call()
```

```
// == INVALID EXAMPLES
```

```
a.balance
```

```
payable(a)
```

sbool

shielded boolean

All comparisons and operators for `sbool` function identically to `bool`. The universal casting rules and restrictions described in [Basics](#) apply.

We recommend reading the point on conditional execution in [Common mistakes](#) prior to using `sbool` since it's easy to accidentally leak information with this type.

```
sbool a = sbool(true)
sbool b = sbool(false)
```

```
// == EXAMPLES
a && b // false
!b    // true
```

Collections

Using type variables in arrays and maps

All `stype` variables can be stored in Solidity collections, much like their unshielded counterparts. They behave normally (as outlined in [Basics](#)) when used as values in these collections. It's when they're used as *both* the keys and values where it gets interesting. This applies to arrays and maps in particular:

```
suint256[] a; // stype as value
function f(suint256 idx) {
    a[idx] // stype as key
    // ...
}

// =====

mapping(saddress => suint256) m; // stype as key and value
function d(suint256 k) {
    m[k]
}
```

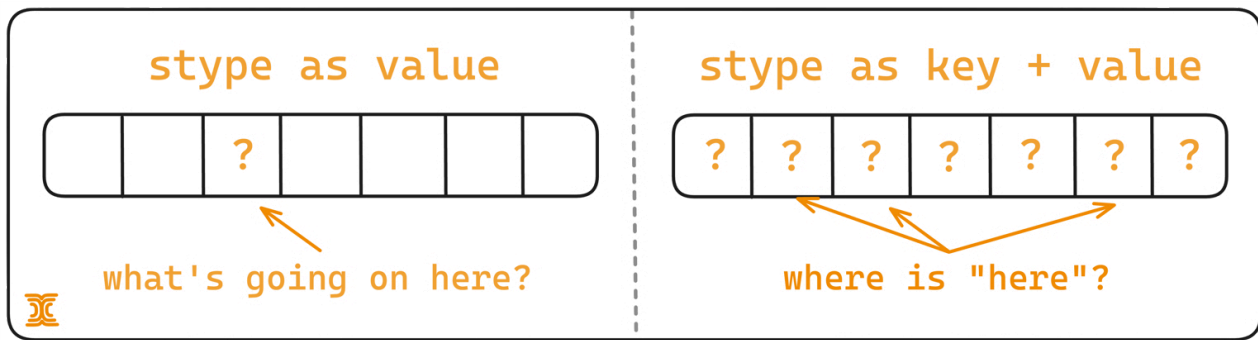
What's special here is that you can hold on to `a[idx]` and `m[k]` without observers knowing which values in the collection they refer to. You can read from these references:

```
sbool b = a[idx] < 10;
suint256 s = m[k] + 10;
```

You can write to these references:

```
a[idx] *= 3;
m[k] += a[idx];
```

Observers for any of these operations will not know which elements were read from / written to.



Using an `stype` as the key and value to a collection shields which element you're using.

In the previous section, we only knew how to shield what was happening for certain elements. Now, we know how to shield which elements are being modified in the first place.

We can take the ERC20 variant discussed in the [Basics](#) section and extend it further to shielded balances, transfer amounts, *and now recipients*.

```
mapping(saddress => uint256) public balanceOf; // key is now saddress

function transfer(saddress to, uint256 amount) public { // recipient no
    balanceOf[msg.sender] -= amount;
    balanceOf[to] += amount;
}
```

Appendix