OpenZeppelin

Bringing your own cryptographic identity to Smart Accounts

November 15th, 2024 | Devcon SEA, Bangkok, Thailand

Agenda

1. Background
2. How to *Account Abstraction*
3. Custom signature algorithms
4. Technical Goals of an Account implementation
5. Security Considerations
6. Building ERC4337 Accounts with OpenZeppelin Contracts
7. Questions

Why are we building an AA framework?

Background

OpenZeppelin

Grant Received from Ethereum Foundation

State of Account Abstraction offering for developers

Wide range of implementations

Multiple providers developed their own offering of Solidity implementations of ERC-4337 accounts

Emerging security challenges

Ranging from battle-tested implementations to newest minimal approaches

Lack of a straightforward approach

ERC-4337 defines minimal requirements, fragmenting the developer experience

Our observations suggested a lack of consensus at certain points in the stack

ERC-4337 Base Layer

The Account implementations should all complain with basic ERC-4337 rules.

An **extensible** base implementation to **lower the entry barrier** to new innovations was missing.

Validation Layer

Accounts **must implement a mechanism to validate user operations securely.**

Digital Signatures are the go-to authentication for cryptographic systems.

Implementing these is not trivial

Modularity Layer

Both smart account providers and ERC-4337 infrastructure providers developed ERCs to address modularity.

Modules are great for extending accounts. Still, **a developer needs to understand the whole stack before writing a module**.

Wizard-like experience

At the end, achieving a wizard-like experience to bootstrap account implementation requires to put together **a standardized version of these layers**.

¿Mass Adoption?

We'll get there

Related ERCs and best practices

ERC-4337

ERC-4337 lays down the definition of the components in Account Abstraction infrastructure.

ERC-7562

To use the ERC-4337 canonical mempool, developers must comply with ERC-7562 validation rules since the start.

ERC-7739

Initially started by @vectorized for Solady, ERC-7739 is a defensive re-hashing scheme that relies on EIP-712 to avoid replayability issues.

Steps for building a custom account

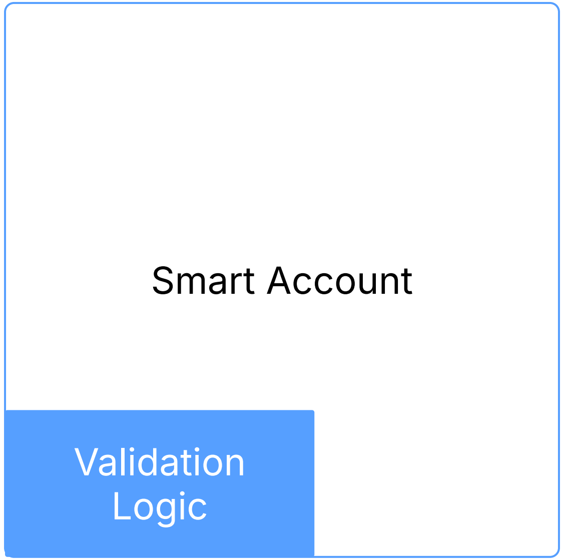
How to *Account Abstraction*?

What's an Smart Account in the first place?

Smart Account

A smart contract with **its own validation algorithm.**

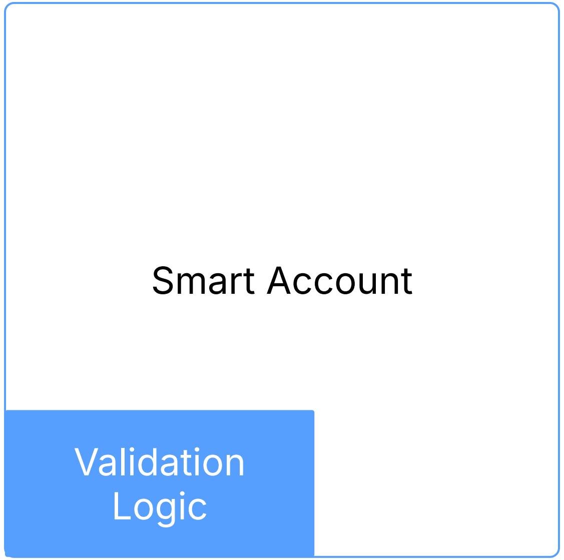
Most of them are based on digital signature schemes.



Innovation in validation logic

Sometimes presented as **modules**, custom validation schemes have produced innovation breakthroughs:

- ZK Email
- ZK Identity
- P256 Validation
- BLS Aggregators
- Social Recovery



Ideally, to build an account you would

MyAccountECDSAClonable.sol

```solidity
// contracts/MyAccountECDSAClonable.sol
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import {AccountECDSAUpgradeable} from "@openzeppelin/contracts-upgradeable/account/draft-AccountECDSAUpgradeable.sol";

contract MyAccountECDSAClonable is AccountECDSAUpgradeable {
    constructor() {
        _disableInitializers();
    }

    function initialize(address signer, string memory name, string memory version) public virtual initializer {
        __AccountECDSA_init(signer, name, version);
    }
}
```

Things to note

Upgradeable?

Not really.

ERC-4337 Accounts are deployed through a factory, requiring an **initialization mechanism**; characteristic present in upgradeable contracts.

Draft?

Yes.

These standards are still marked as draft and some other auxiliary ERCs are changing quite fast.

Clonable?

Yes.

A clone is cheap to deploy and its implementation very straight forward.

Custom signature algorithms

Exploring digital signatures

Traditional Public Key Infrastructures have become an innovative anchor to build on.

- A **cryptographic primitive** well embedded into traditional systems
- **Widely adopted** in corporate and government environments
- **Battle tested?** These had secured communications and authentication systems before we were doing ICOs

means regulation gud

Available on OpenZeppelin Contracts

ECDSA

The good old secp256k1 signature verification algorithm available to the EVM as a precompile with malleability protection.

P256

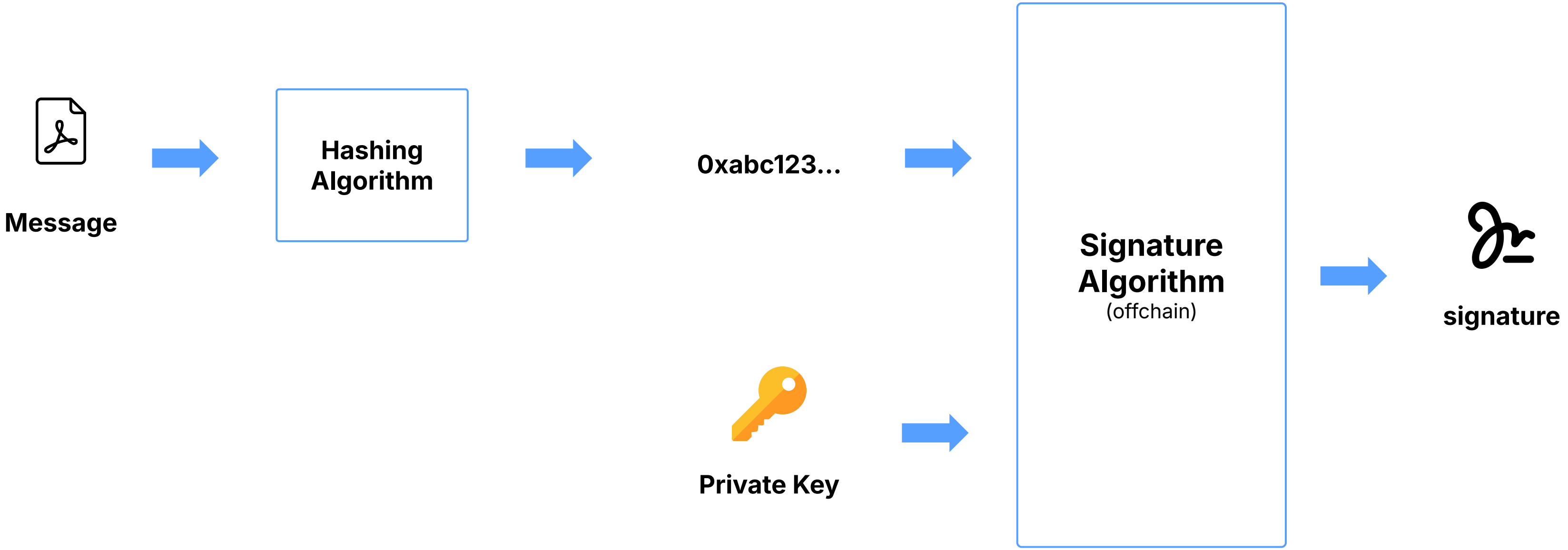
A gas-efficient implementation of the ECDSA algorithm over secp256r1 curve, known as P256.

Solidity implementation that relies on the RIP-7212 precompile if it's available.

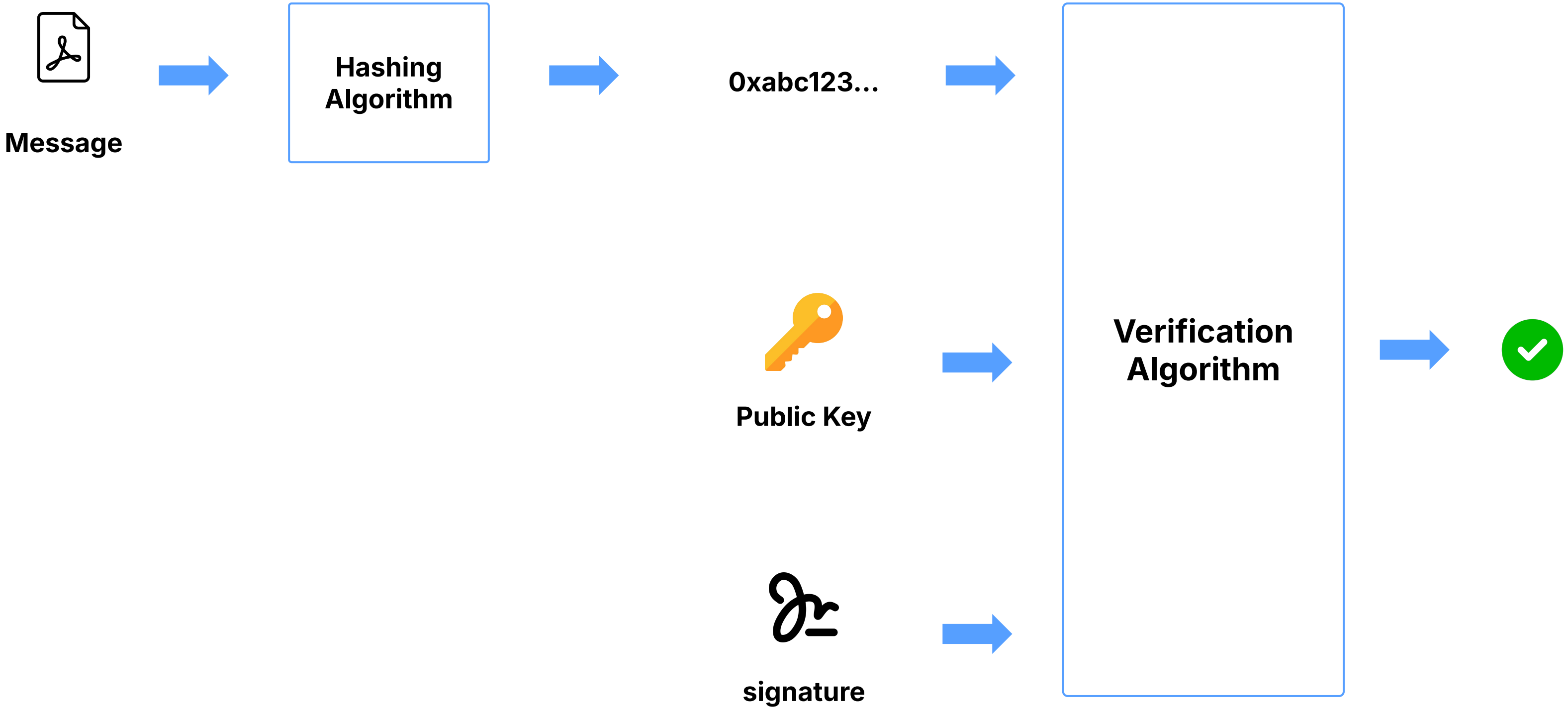
RSA

The RSA PKCS1.5 verification algorithm outlined in RFC-8017 are still of popular usage amongst governments and DNSSEC providers.

Cryptographic Signatures 101



Cryptographic Signatures 101



Two algorithms

A signature scheme requires an algorithm to **produce digestives** and another to **verify a signature**.

These combinations are defined by the offchain signer. Common combinations include:

- P256 + SHA256
- Native ECDSA + Keccak256
- RSA + SHA256

Hashing Algorithm

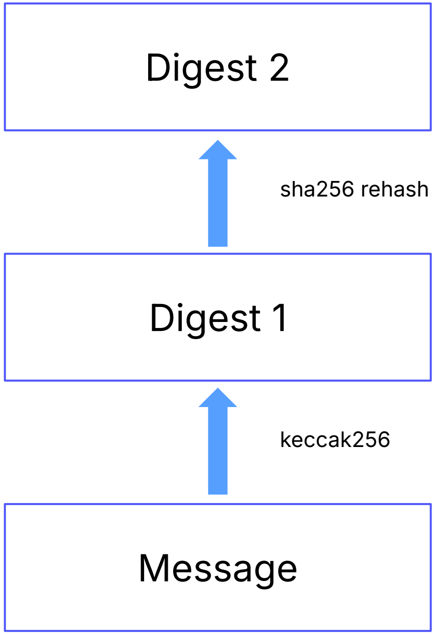
Verification Algorithm

Dependency on offchain signers

Having two algorithms complicates things since a smart contract developers cannot arbitrarily choose the hashing scheme of their users.

They're constrained by the offchain device, breaking standards like ERC-1271 for signature validation if the signer cannot sign over a different hashing function.

Rehashing mechanisms can be used for **normalization.**

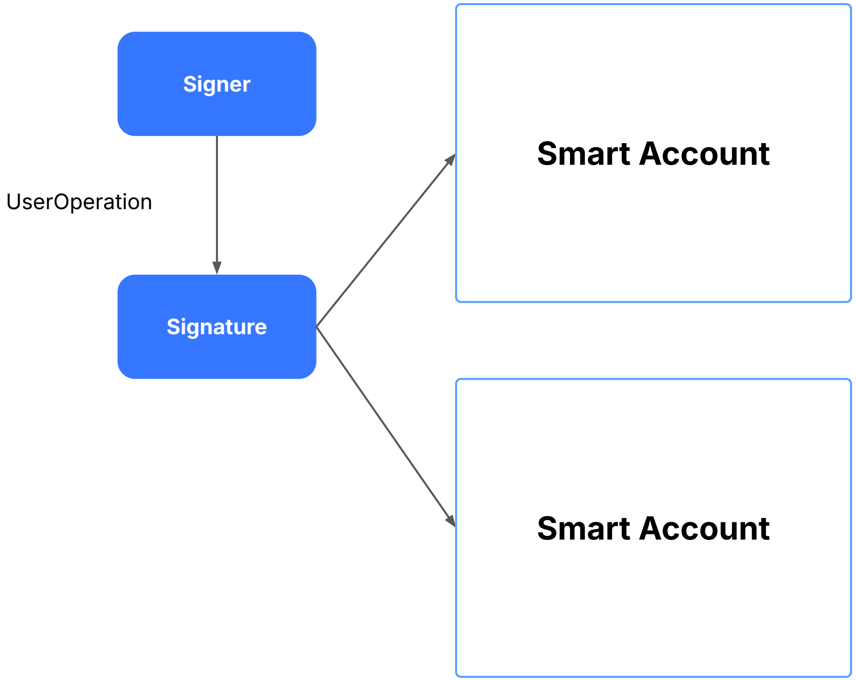


OpenZeppelin

Replayability across same-key accounts

Given 2 accounts controlled by the same private key, **an user operation could be replayed*** on the other account unless the signature is tied to the **contract address** and **chain id.**

Best way to do this is with EIP-712.



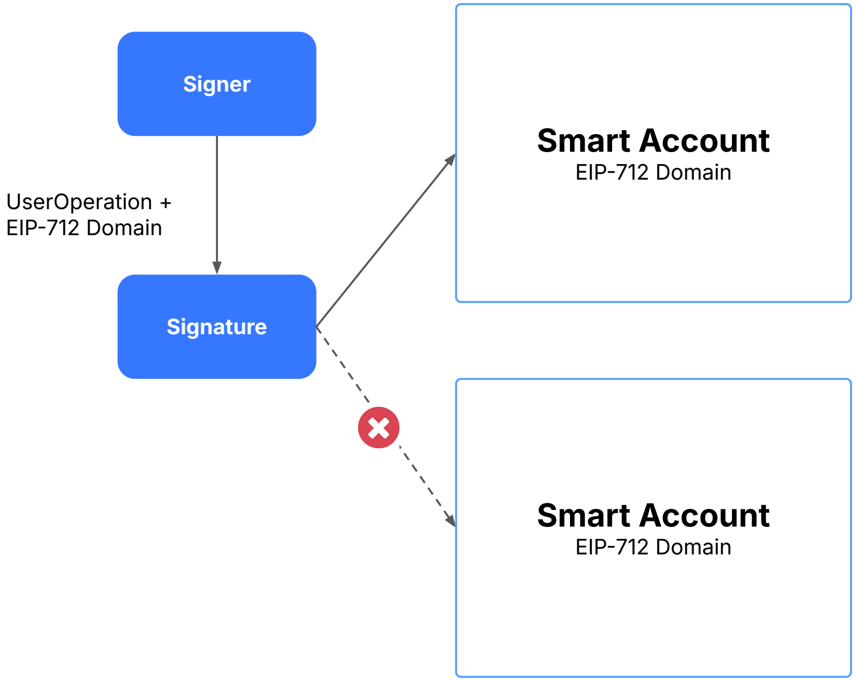
**Issue initially discovered by curiousapple.eth*

OpenZeppelin

Replayability across same-key accounts

Given 2 accounts controlled by the same private key, **an user operation could be replayed*** on the other account unless the signature is tied to the **contract address** and **chain id.**

Best way to do this is with EIP-712.

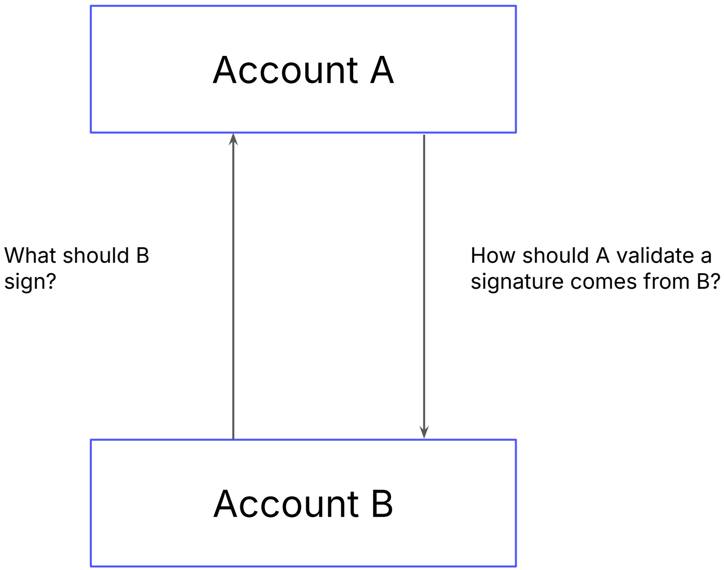


∗Issue initially discovered by curiousapple.eth

Accounts owned by accounts

In theory, an Account Contract should be able to own another account (e.g. as in a multisig scheme). However, accessing other account storage **violates ERC-7562 storage validation rules**.

Similarly, nesting EIP-712 domains makes it difficult for users to read what they're signing



"There are only ~~three~~ four hard problems in computer science: Cache invalidation, naming things, off-by-one errors and **signatures in Account Abstraction** wallets"

How we're doing it

Technical Goals of an Account implementation

Secure

Our go-to recommendation is a **solid base layer** for developers who're writing their own Solidity implementations of accounts.

Layered

To accomodate developers building with our libraries, the Account contracts **must be consumed in layers**.

Extensible

Modularity has become a **massive source of innovation** that developers should have easy access to.

Security Considerations

Working with custom validation schemes brings implementation challenges we want to address for developers using OpenZeppelin Contracts.

1. **Audited Building Blocks**
   Bootstrap your smart account with battle-tested components.
2. **Strong cryptographic primitives**
   A new wave of innovations in smart accounts must be solid from the ground up.
3. **Avoiding replayability issues**
   Abstracting away replayability prevention.
4. **Maintain user operation readability**
   Allow end-users to inspect their user operations with clarity.
5. **Community Driven**
   Community drives interoperability and standardization.

Building ERC4337 Accounts with OpenZeppelin Contracts

Step 1: Picking Base Account

```
abstract contract AccountBase is IAccount, IAccountExecute {
    // ...

    /**
     * @dev Validation logic for {validateUserOp}.
     *
     * IMPORTANT: Implementing a mechanism to validate user operations is a security-sensitive operation
     * as it may allow an attacker to bypass the account's security measures. Check out {AccountECDSA},
     * {AccountP256}, or {AccountRSA} for digital signature validation implementations.
     */
    function _validateUserOp(
        PackedUserOperation calldata userOp,
        bytes32 userOpHash
    ) internal virtual returns (uint256 validationData);

    // ...
}
```

Step 2: Binding signatures to a domain

```
contract MyAccountECDSA is ERC7739Signer, AccountBase {
  /// ...
}
```

Step 3: Picking a validation mechanism

```
function _validateUserOp(
    PackedUserOperation calldata userOp,
    bytes32 userOpHash
) internal virtual override returns (uint256) {
    return
        _isValidSignature(userOpHash, userOp.signature)
            ? ERC4337Utils.SIG_VALIDATION_SUCCESS
            : ERC4337Utils.SIG_VALIDATION_FAILED;
}

function _validateSignature(
    bytes32 hash,
    bytes calldata signature
) internal view virtual override returns (bool) {
    (address recovered, ECDSA.RecoverError err, ) = ECDSA.tryRecover(
        hash,
        signature
    );
    return signer() == recovered && err == ECDSA.RecoverError.NoError;
}
```

Step 4: Deploy

Thank You.

Ernesto García
ernesto@openzeppelin.com

OpenZeppelin

Community Contracts repository

Creating a factory

MyAccountECDSAClonable.sol

```
// contracts/MyFactoryAccountECDSA.sol
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import {Clones} from "@openzeppelin/contracts/proxy/Clones.sol";
import {Address} from "@openzeppelin/contracts/utils/Address.sol";
import {MyAccountECDSAClonable} from "./MyAccountECDSAClonable.sol"

contract MyFactoryAccountECDSA {
    using Clones for address;

    address private immutable _impl = address(new MyAccountECDSAClonable());

    /// @dev Predict the address of the account
    function predictAddress(bytes32 salt) public view returns (address) {
        return _impl.predictDeterministicAddress(salt, address(this));
    }

    ...
}
```

Creating a factory

MyAccountECDSAClonable.sol

```
// contracts/MyFactoryAccountECDSA.sol
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import {Clones} from "@openzeppelin/contracts/proxy/Clones.sol";
import {Address} from "@openzeppelin/contracts/utils/Address.sol";
import {MyAccountECDSAClonable} from "./MyAccountECDSAClonable.sol"

contract MyFactoryAccountECDSA {
    using Clones for address;

    address private immutable _impl = address(new MyAccountECDSAClonable());

    ...

    /// @dev Create clone accounts on demand
    function cloneAndInitialize(bytes32 salt, bytes calldata data) public returns (address) {
        return _cloneAndInitialize(salt, data);
    }

    ...
}
```

Creating a factory

MyAccountECDSAClonable.sol

```
// contracts/MyFactoryAccountECDSA.sol
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import {Clones} from "@openzeppelin/contracts/proxy/Clones.sol";
import {Address} from "@openzeppelin/contracts/utils/Address.sol";
import {MyAccountECDSAClonable} from "./MyAccountECDSAClonable.sol"

contract MyFactoryAccountECDSA {
    using Clones for address;

    address private immutable _impl = address(new MyAccountECDSAClonable());

    ...

    /// @dev Create clone accounts on demand and return the address. Uses `data` to initialize the clone.
    function _cloneAndInitialize(bytes32 salt, bytes calldata data) internal returns (address) {
        address predicted = predictAddress(salt);
        if (predicted.code.length == 0) {
            _impl.cloneDeterministic(salt);
            Address.functionCall(predicted, data);
        }
        return predicted;
    }
}
```

Benefits of a factory-based account

- Counterfactual addresses
  - Cross-chain except ZkSync
- Cheap deployment thanks to minimal clones
- Deployed and initialized by the factory