ABI SVM: the universal Session Validation Module

Abstract

In 2023 Biconomy pioneered the Modular Session Keys approach. This approach unlocks significant extensibility and composability and allows even very sophisticated use cases to be implemented via the Biconomy Session Keys eco-system.

We have built the *framework.* This framework was built with extensibility as a priority. As a result, it was battle-tested proving its efficiency and ability to handle even very complex tasks.

However, dApps had to build Session Validation Modules (smart contracts that connect to Biconomy Session Key Manager and implement custom dApp-specific checks) within this framework by themselves.
It is still a good pattern for very complex use cases but can introduce additional development overhead for dApps with simpler requirements.

To reduce such overhead and provide dApps with an out-of-the-box solution we release ABI SVM - the universal Session Validation Module for basic use cases.

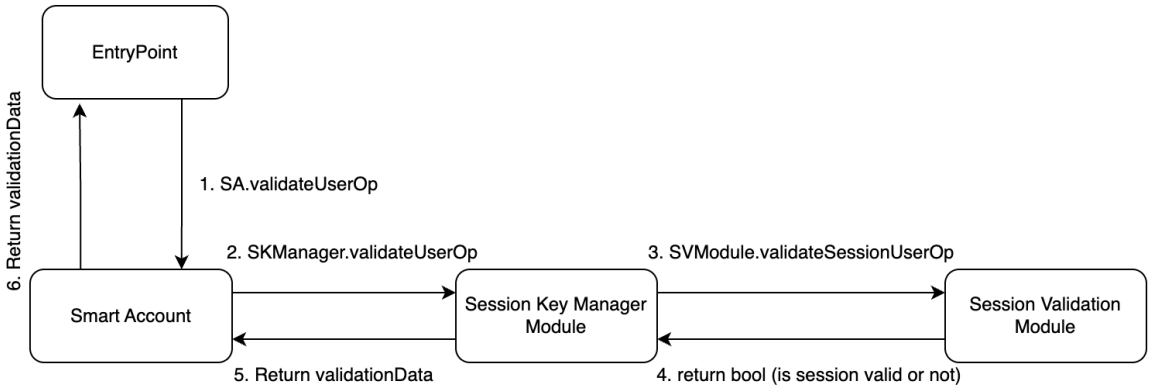
What it enables

SVMs basics

To start with, what is a Session Validation Module?

It is a tiny Smart Contract that implements checks for a pre-defined range of conditions specific to a given use case.

After performing basic checks (general Session Key validity) Session Key Manager interacts with an SVM to get the result of other checks. See the figure below.



Data flow for the validation of the Session Key signed userOp

A very simple example of an SVM is ERC20 SVM which allows validation of ERC20 approvals and transfers. Conditions in this case include:

- Destination: the ERC20 token contract address
- Receiver/Spender address
- Token Amount

ABI SVM

The above-mentioned ERC20 SVM perfectly works for ERC20 transfers/approvals. However if one needs to verify a Session that performs the swap, stake, or some custom interaction with a DeFi protocol, ERC20 SVM won't be able to handle this because it expects some set of parameters and conditions that are hard-coded into the SVM.

> ABI SVM solves this issue by allowing interactions with any function with any set of parameters to be verified through it.

ABI stands for Application Binary Interface and is commonly used as a definition for an entire set of functions of a smart contract. Thus ABI SVM means that this Session Validation Module can validate interactions with any function of any smart contract based on its interface (selector+args).

Let's assume there is a protocol that has a very specific function:

`protocolSpecificFunction(address token, uint256 amount, bytes calldata data)`

A dApp wants to enable a Session for a user that only allows interaction with this function, and only if args comply with certain conditions.

In this case, a Session may be created with the following example parameters:

- Session Validation Module: ABI SVM
- Destination contract: address of the Protocol Contract that implements `protocolSpecificFunction`
- Function selector: `protocolSpecificFunction.selector`
- Permitted value limit: whatever native token value transfer is allowed along with the `protocolSpecificFunction` call
- List of the Rules for the args:
  - `token` should be equal to a certain `allowedTokenAddress`
  - `amount` should be less or equal to a certain `tokenAmountLimit`
  - `data` should start with some `bytes32 starting32BytesWord` and `data.length` should be more than a certain `minimumDataSize`

When a Session Key userOp is submitted, the Session Key Manager calls ABI SVM, then ABI SVM decodes the actual `userOp.callData` to check that exactly the `protocolSpecificFunction` is called from the Protocol Contract with value not exceeding value limit and args complying to the conditions that were set when creating and enabling a Session.

Batched flow

ABI SVM is a perfect match for batched userOps.

Imagine, a dApp that wants to perform the following actions on the user’s behalf:

1) Approve token A

2) Swap token A to token B

3) Transfer token B

And wants to execute it atomically.

In this case, three Sessions should be enabled to be processed via ABI SVM:

1) Define conditions for the destination contract being token A address, value, approve selector, spender address, max amount of token approved

2) Define conditions for the destination contract to be the DEX Pool address, swap selector, token A and B addresses, minimal amount to receive, and other args if any

3) Define conditions for the destination contract being token B address, value, transfer selector, receiver address, max amount of token transferred

Thus the ABI SVM is the universal building block for performing custom batched action flows via Session Keys.

How it works

This section contains a technical deep-dive on how the ABI SVM works.

Validation flows

Like Session Validation Module, ABI SVM supports regular (non-batched) validation flow, and batched validation flow.

The first one is handled by the `validateSessionUserOp` method, and the second by the `validateSessionParams` method.

In the `validateSessionUserOp` method, ABI SVM performs the check that this userOp is actually the regular execution userOp, checking the execution method selector to match the selector of the `execute` or the `execute_ncC` methods of the Smart Account.

It also performs the signature check and the check of the Session parameters.

For the batched flow, `validateSessionParams` just checks Session parameters against what was allowed when the ABI SVM Session was enabled.

Session Parameters check

This is the most interesting part of how ABI SVM works, that is common to both flows. It is performed by an internal `_validateSessionParams` method.

At this stage, ABI SVM parses `_sessionKeyData` arg (provided by Session Key Manager through `validateSessionParams` or `validateSessionUserOp` methods) to get the information about what is permitted for a given transaction (please note, that for a batched flow by transaction we mean one action within `executeBatch` execution).

Then it compares what transaction is trying to execute to what is permitted.

The layout of the `_sessionKeyData` is the following:

```
* The _sessionKeyData layout:
* Offset (in bytes)    | Length (in bytes) | Contents
* 0x0                  | 0x14              | Session key (addre
* 0x14                 | 0x14              | Permitted destinat
* 0x28                 | 0x4               | Permitted selector
* 0x2c                 | 0x10              | Permitted value li
* 0x3c                 | 0x2               | Rules list length
* 0x3e + 0x23*N        | 0x23              | Rule #N
*
* Rule layout:
* Offset (in bytes)    | Length (in bytes) | Contents
* 0x0                  | 0x2               | Offset (uint16)
* 0x2                  | 0x1               | Condition (uint8)
* 0x3                  | 0x20              | Value (bytes32)
*
* Condition is a uint8, and can be one of the following:
* 0: EQUAL
* 1: LESS_THAN_OR_EQUAL
* 2: LESS_THAN
* 3: GREATER_THAN_OR_EQUAL
* 4: GREATER_THAN
* 5: NOT_EQUAL
```

Thus it contains the following parameters that can be verified:

Session Key - address, returned to the `validateSessionUserOp` or to the Session Key Manager to validate that the signer of the userOp is one of the enabled Session Keys.

Permitted destination contract - address of the contract that Smart Account is allowed to call within the `execute` or `executeBatch` execution.

Permitted selector - the selector of the method that is allowed to be called on the Permitted destination contract

Permitted value limit - maximum value in chain native tokens that is allowed to be transferred along with the call

Rules

Rules define permissions for the args of an allowed method.

With Rules you can precisely define what should be the args of the transaction that is allowed for a given Session.

Every Rule works with a single static arg or a 32 bytes chunk of the dynamic arg.

Since the ABI Encoding translates every static param into a 32bytes word, even the shorter ones (like `address` or `uint8`), every Rule defines a desired relation (`Condition`) between n-th 32bytes word of the `calldata` and a reference Value (that is obviously a 32bytes word as well).

So, when dApp is creating a `_sessionKeyData` to enable a session, it should convert every shorter static arg to a 32bytes word to match how it will be actually ABI encoded in the `userOp.callData`.

For the dynamic args, like `bytes`, every 32bytes word of the `calldata` such as offset of the bytes arg, length of the bytes arg, and n-th 32bytes word of the bytes arg can be controlled by a dedicated Rule.

Below is an example of how Rules are defined:

```
rules: [
                // Rule for the uin256 arg
        {
          offset: 0, //defines the position of a 32bytes word in
          referenceValue: ethers.utils.hexZeroPad("0x0400", 32),
          condition: 1, //actual arg should be less than or equa
        },
                // Rule for the offset of the `bytes calldata` a
        {
          offset: 32,
          referenceValue: ethers.utils.hexZeroPad("0x40", 32),
          condition: 0, // equal
        }, // offset == 0x40 = 64bytes = first arg(32) + offset_
```

```
            // Rule for the length of the `bytes calldata` a
            {
      offset: 64,
      referenceValue: ethers.utils.hexZeroPad("0x20", 32),
      condition: 3, // greater or equal
    }, // length of the `bytes` arg should be >= 0x20 (32byt

            // Rule for the first 32bytes word of the `bytes
    {
      offset: 96,
      referenceValue: ethers.utils.hexZeroPad("0xdeafbeef",
      condition: 0, //should be equal to the `referenceValue
    },
            // data part of `bytes` arg can be divided into
  ]
```

Possible conditions range from 0 to 5 and are described above.

With such an approach, ABI SVM allows setting up custom permissions for what is allowed to be executed with every transaction.

Limitations

There are still some checks, that ABI SVM can not perform.

As you can notice, ABI SVM expects the only reference value for the arg. This makes it impossible to check, for example, that an arg of the `address` type is included in the list of allowed addresses.

However, such a check can be still performed via custom SVM and that's why the Biconomy Modular Session Keys approach is so powerful. Any complex use case you have can be served via custom-built SVMs even if there's no pre-built SVM for this purpose yet.

Conclusion

With being a perfect building block for allowing complex batched action flows or just single custom actions via Session Keys, ABI SVM becomes a perfect universal SVM for many dApps in most cases removing the need to build custom SVMs.