Smart Contract Verification

Details about the verification microservice

A microservice written in Rust provides fast and efficient contract verification. The application runs as an HTTP server and verification requests are made using a REST API.

Basic Info

Location

- Application:https://github.com/blockscout/blockscout-rs/tree/main/smart-contract-verifier
- Http API:https://github.com/blockscout/blockscout-rs/blob/main/smart-contract-verifier/smart-contract-verifier-http
- Latest release information and changelogs:https://github.com/blockscout/blockscout-rs/releases

Activation

- Update Blockscout to the 4.1.8+ release
- Add the following ENV variables:
  - MICROSERVICE_SC_VERIFIER_ENABLED=true
  - MICROSERVICE_SC_VERIFIER_URL=...
  - (verifier endpoint, seehttp configuration
  - for more details)
- *
- If running locally, the smart-contract-verifier is integrated withdocker-compose
- ``
  - Registry of docker images is available via the docker-compose file. For registry information seehttps://github.com/blockscout/blockscout-rs/pkgs/container/smart-contract-verifier
- *

Architecture

The service consists of 2 parts, a verification library and a transport layer that serves requests.

- Verification Library
- : Provides verification and a communication interface for verification functions. Includes modules for Solidity, Sourcify, and Vyper.
- Transport Layer
- : Verification requests are sent via HTTP.

Benefits

We've seen a 10X improvement in verification speed over the previous implementation. Benefits of the new microservice include:

- Scalability
- : Remove CPU intensive tasks from the primary Blockscout instance. Modularity provides the ability to scale quickly.
- Speed
- : Using native compilers greatly increases compilation speed relative to the JS wrapped versions running on Node.js used previously.
- Improved efficiency
- . New algorithm supports contracts that contain several metadata hashes, improving efficiency.

Library Initialization

On startup, the library must retrieve all available compilers. Compiler information consists of a link to download the compiler and its hashsum[xample]. A separate thread also starts during initialization to update this list periodically.

Library processes differ based on verification for Solidity/Vyper contracts or verification via Sourcify. Both are detailed below.

Solidity and Vyper Verification

Verification processes are similar for contracts written in Solidity and Vyper.

For Solidity contracts, verification is available for multiple files or standard json input. Single file verification is also possible - users still verify via the multiple file method and only upload a single file.

For Vyper contracts, only multiple file verification is available.

Request Values

Each request contains the following values:

- Deployed bytecode
- : bytecode of the contract stored on chain
- Creation transaction (tx) input
- : input data provided in transaction to create the contract
- Compiler version
- : Version of compiler used to compile the contract
- Content
- : Verification method related fields. E.g. for Solidity multiple files those fields are:
  - Sources
  - : mapping from source files to their contents
  - Evm version
  - : version of the EVM used to compile sources
  - Optimization runs
  - : optional parameter that enables optimizations and specifies number of optimization runs
  - Contract libraries
  - : optional parameter that specifies addresses of the contract libraries used in compiled contracts
- *

Verification Steps

1. Download the compiler
2. using the specified compiler version from the url retrieved oninitialization
3. (or update).
4. Compilation
5. : compile using the provided and slightly modified (seeVerification Algorithm
6. section) sources.
7. Verification
8. : compare compiler output from step 2 with bytecodes (deployed bytecode and creation tx input) provided by the caller.
9. During verification all contracts obtained through local compilation are checked one by one, and if any corresponds to provided bytecodes, the verification is considered successful. Contract paths and names are already indexed from available contracts, and constructor arguments are extracted.
10. Return Values
11. : See below.
12.

Returned Values

As a result of successful verification the following struct is returned:

- Compiler input
- : Data used as an input for the compiler
- Compiler version
- : version of the compiler used in verification (corresponds to the version provided in request)
- File path
- : path of the file containing verified contract
- Contract name
- : name of the verified contract
- Abi
- : application binary interface of the verified contract (https://docs.soliditylang.org/en/v0.8.13/abi-spec.html?highlight=abi
-)
- Constructor arguments
- : optionally returned constructor arguments used during on chain creation of the contract.
-

In case of an error, enum is returned with the corresponding error and its description (e.g. if compilation failed, a Compilation error is returned with the error data obtained from the compiler).

Sourcify Verification

The request is sent via proxy to theSourcify service. It accepts the following arguments:

1. Address of the contract to be verified.
2. Chain id where the contract is deployed.
3. Files required for Sourcify verification.
4. (optionally)Contract index, if the provided files contain multiple metadata files (i.e. multiple contracts).
5.

These arguments are sent to the Sourcify Api, which returns the verification status.

Successful Verification

Either the data provided was valid and the contract was verified by Sourcify, or the contract was previously verified. To obtain the actual data used for verification we send another request to retrieve source files used in verification.

We expect data used for verification to be in themetadata.json file, so it is parsed and the information retrieved. This includes file and contract names, compiler and evm versions, info about optimizations, contract libraries, abi, and sources.

Note : Constructor arguments cannot currently be obtained from the metadata via Sourcify. This requires the input used for contract creation transaction and the bytecode resulting from local compilation, which are not provided.

Failed Verification

If verification fails we return an error with description obtained from Sourcify response

Verification Algorithm

Verification is divided into 2 parts to effectively deal with metadata hash differences in contracts. Often, very similar contracts will have very different metadata hashes based on small changes within the contract (ie an extra space in the source code). To address this, the algorithm:

1. Extracts all metadata hash fields from c_reationTxInput_.
2. Compares non-metadata sections of the onchain bytecode with the results of local compilation.
3.

Metadata hashes are extracted by compiling the same contract twice (with a slight modification to the 2nd contract that does not change any other aspect of the resultant bytecode). The resulting c_reationTxInput_s are compared and metadata hashes extracted by finding which parts of the inputs differ.

Example

An additional space in the source code can be used for the modification process. InExample 1 andExample 2 the only difference between both contracts is a space in the beginning ofExample 2 . However, this still completely changes the metadata hash (metadata hash differences shown inbold red).

Note : To update the metadata hash in production we use an additional library passed into the compiler. It does not effect the resultant bytecode, but the compilation input is altered, leading to updated metadata and an updated metadata hash.

Example 1

```

Copy // pragma solidity ^0.8.7; contract Main {}

```

creationTxInput: 6080604052348015600f57600080fd5b50603f80601d6000396000f3fe6080604052600080fdfea264697066735822
12203cc19b771ee83d8cc6995191bd9092a82f3e66b95aacaf1cfc893d3dc27abcd2 64736f6c63430008070033

Example 2

```

Copy // pragma solidity ^0.8.7; contract Main {}

```

creationTxInput: 6080604052348015600f57600080fd5b50603f80601d6000396000f3fe6080604052600080fdfea264697066735822
1220b3902f5797f05bb62858c8d41f09402093c85957c4ae1d773e4eb1cb2000f093 64736f6c63430008070033

Verification Process

1. Compile the original contracts using data provided by the caller then compile those contracts again while passing in the additional library. The result is two slightly differentCreationTxInputs
2. if there is any encoded metadata.
3. Separate theMainPart
4. and theMetadataPart
5. .
6.

1. MainPart
7.
    1. consists of consecutive bytes which are not related to the metadata hash (all bytes are the same for both local c_reationTxInputs_).MetadataPart
8.
    1. consists of consecutive bytes related to metadata hash (some bytes differ between local c_reationTxInputs_).
9. 4.
10. Iterate through the two local contracts byte by byte to find the first byte that differs.
11.
    1. If there are no such bytes, then there is no metadata hash and the wholecreationTxInput
12.
    1. consists of just oneMainPart
13.
    1. .
14.
    1. If any differences are found, identify it as a hash section of the metadata hash.
15.
    1. Iterate back byte by byte to find the beginning of that metadata hash. When we find the beginning we know where that metadata hash starts and ends. All non-categorized bytes before the start are identified as theMainPart
16.
    1. and bytes between the start and end identified as theMetadataPart
17.
    1. . Continue until there are no uncategorized bytes.
18.
    1. Once the entire localcreationTxInput
19.
    1. is categorized, we compare those parts with the corresponding remotecreationTxInput
20.
    1. bytes. For theMainPart
21.
    1. remotecreationTxInput
22.
    1. must have exact equivalence. For theMetadataPart
23.
    1. we compare some parts of the metadata (e.g. that encoded solc versions are the same), but in general we do not require them to be equal.
24. 14.
25.

Example

LocalCreationTxInput1:
6080604052348015610010057600080fd5b506040518060200161002190610050565b6020820181038252601f19601f8201166040525060009080519060200190610004a92919061005c565b5061015f565b6
6080604052348015600f57600080fd5b50603f80601d6000396000f3fe6080604052600080fdfea2646970667358221220805aa9fe4ec055702024afa5ac21c2104f0b14be0ffab086f0d6e9b5701073f864736f

Parts 1: ``[
MainPart(6080604052348015610010057600080fd5b506040518060200161002190610050565b6020820181038252601f19601f8201166040525060009080519060200190610004a92919061005c565b5061(

MetadataPart(a264697066735822 1220187a584947e37ebca43172929f7e159fe28696b0edd97bbfad5b0a265f6f8869 64736f6c634300080e0033),

MainPart(6080604052348015600f57600080fd5b50603f80601d6000396000f3fe6080604052600080fdfe),

MetadataPart(a264697066735822 1220805aa9fe4ec055702024afa5ac21c2104f0b14be0ffab086f0d6e9b5701073f8 64736f6c634300080e0033)]

LocalCreationTxInput2:
6080604052348015610010057600080fd5b506040518060200161002190610050565b6020820181038252601f19601f8201166040525060009080519060200190610004a92919061005c565b5061015f565b6
6080604052348015600f57600080fd5b50603f80601d6000396000f3fe6080604052600080fdfea2646970667358221220bd9f7fd5fb164e10dd86ccc9880d27a177e74ba873e6a9b97b6c4d7062b26ff064736

Parts 2:

[
MainPart(6080604052348015610010057600080fd5b506040518060200161002190610050565b6020820181038252601f19601f8201166040525060009080519060200190610004a92919061005c565b5061(

MetadataPart(a264697066735822 12202e82fb6222f966f0e56dc49cd1fb8a6b5eac9bdf74f62b8a5e9d8812901095d6 64736f6c634300080e0033),

MainPart(6080604052348015600f57600080fd5b50603f80601d6000396000f3fe6080604052600080fdfe),

MetadataPart(a264697066735822 1220bd9f7fd5fb164e10dd86ccc9880d27a177e74ba873e6a9b97b6c4d7062b26ff0 64736f6c634300080e0033)

]

Constructor arguments

Constructor arguments are extracted at the end of the verification process. Constructor arguments are appended at the end of thecreationTxInput returned by the compiler. So, to extract these arguments we remove parts presented in localcreationTxInput , and consider everything else as likely constructor arguments.

After that we attempt to map potential constructor args to the constructor abi of the contract. If successful, extracted args are returned as actual constructor arguments.

Verification Http Api

Currently, verification requests are sent via HTTP requests. This module runs an HTTP server and provides endpoints through which external services (main Blockscout instance, in our case) send verification requests and receive responses. Endpoints provided by the service and configuration details are described in thecorresponding Readme file .

Currently only synchronous endpoints are available. In the future a new service that handles asynchronous requests on top of the current smart-contract-verifier may be implemented.

Last updated2 months ago