Warp 2.0 — Transpiling Directly From Solidity to Cairo

Guy McComb

Follow

Nethermind.eth

--

Listen

Share

By

[Guy McComb

](https://twitter.com/0xGai_)and

[Swapnil Raj

](https://twitter.com/swp0x0)Thanks to

[Jorik Schellekens

](https://twitter.com/mempoolsurfer), Dom Henderson,

[Eli Barbieri

](https://twitter.com/elicbarbieri), and

[Carmen Rodriguez

](https://twitter.com/cicr_99)for their contributions.

We are excited to announce the second version of Warp, now designed to compile your Solidity code directly into Cairo.

Warp 1 set out to show that compiling Solidity code into Cairo was possible, and paved the way for developers to access the benefits of StarkNet without needing to master Cairo. Using everything we learned from Warp 1, we have written a new version adding vast improvements to contract efficiency and user experience. In this blog, we will talk through the improvements made to Warp, transpile OpenZeppelin's ERC20 contract, and describe future plans for the project.

Warp 2 vs Warp 1

Warp 2 improves on the prior version by transpiling directly from Solidity to Cairo. In Warp 1, the Solidity smart contract was first compiled into YUL (Solidity's intermediate representation) and then transpiled to Cairo. Skipping the YUL intermediary means we don't have to transpile many low-level calls and rather transpile the higher-level Solidity. Sounds interesting, but what does this mean for users?

- Reduced code size
- Smaller step count when calling functions (functions require less computation)
- Improved Cairo readability
- Unsupported feature messages to know which features in Solidity are not supported

Reduced code size & smaller step count

YUL will always have more instructions than the Solidity it is generated from. As an example, the simple 1-line function in Solidity grows to 6 lines of YUL with additional function calls.

Solidity:

YUL:

With Warp 2 going directly from Solidity to Cairo, we avoid many of these unnecessary code additions since Cairo has equivalents to most Solidity expressions. The result is a much smaller Cairo contract and a reduced step count for all

functions.

Looking at the graph below, you can see that the transpiled file size has been reduced by half. This is important since many Warp 1 transpiled contracts were too large to deploy.

Onto step count. Similar to line count, Warp 2 provides a step count that is a fraction of Warp 1's.

The goal is to have the line and step count of the transpiled contracts closer to those of the handwritten contracts. Generated code will always have more instructions than handwritten, but there are optimizations we have planned in our road map that can bring both these metrics down even further.

Improved Cairo readability

Looking at the images below, you can see the transpiled code is a manageable size, is easily readable, and the comments from the Solidity contract are preserved.

Helpful unsupported-feature messages

There are now clear unsupported-feature messages to show which Solidity features aren't supported by the transpiler.

There are fundamental differences between StarkNet and Ethereum. Certain features are not yet implemented on StarkNet, and other Ethereum features would not make sense on StarkNet and therefore won't be implemented. The goal of the Warp team is to get the transpilation as safe and efficient as possible, but right now most off-the-shelf contracts will still need modification to get transpiled. These unsupported feature messages will help guide users. If you are interested in seeing which unsupported features are being worked on at the moment see our Github repo.

Testing & auditing

We've held the new implementation to the same testing standard as the last one: adapting solc's (Solidity compiler) semantic test suite to execute all tests that contain our supported features and running a number of our own behavior tests against it. An audit is also being performed by ABDK which is expected to be completed in a few months.

Installation & usage

Installation

Enough talking about it, let's get to using it.

To get Warp running you will need to have:

1. Install Python3.7 and the venv module
2. Install yarn
3. Follow the Warp install instructions on our repo

If Warp Version 2.XX

is returned when the command warp version

is run then Warp was successfully installed.

Transpiling

Now onto transpiling and deploying a contract. We will be using OpenZeppelin's ERC20 contract in this example.

1. Clone the OpenZeppelin contracts repo
2. Run the Warp transpile command:

You will see that the command fails, this is because currently, the contract contains some unsupported features (remember what we said about picking projects off the shelf). We will have to remove these features to have the transpilation successfully run.

First, we will be removing the indexed parameters for the emitted events in the IERC20.sol interface. Indexed parameters in

events are currently being developed.

Before:

After:

The second modification is to comment out the _msgData function in the Context.sol contract. Due to the fundamental differences in calldata encoding between Solidity and StarkNet, msg.data

does not transpile nicely to what a user would expect.

Now that those two unsupported features are removed, we can successfully transpile the contract.

Run:

Warp outputs the transpile Cairo contract into a folder called warp_output. This folder will be present in the directory where you called the warp transpile

command. Inspecting the folder you should find the transpiled ERC20 contract named ERC20__WARP_CONTRACT__ERC20.cairo.

Now the file can be compiled and deployed. Run the following to compile the Cairo contract:

Deploying

Assuming you have set up your StarkNet account you can now deploy the contract onto L2. To deploy our contract we need a name and ticker for our dummy/test coin. We will use NETHERCOIN and NETH. StarkNet does not support Solidity-style strings, so the Warp team treats them as dynamic arrays of bytes. To match the StarkNet abi we need to pass the length of the arrays and the value of each byte.

Once the contract is compiled and deployed, a contract and transaction hash will be returned. If you wait a few minutes your transaction will be accepted on L2 and you should be able to see the contract on the StarkNet block explorer Voyager.

Deploying Solidity to Starknet has never been so easy. As more features are added to StarkNet we will be matching them to Solidity allowing the transpiler to handle more complex contracts.

Next steps for Warp

With Cairo v0.9 being recently released, the team will be adding support for contract factories (contracts deploying contracts), fallback functions with arguments, delegate calls, and possibly some version of inline Cairo. Inline Cairo has been chosen over inline assembly since inline assembly will introduce the inefficiencies of Warp version 1.

We are also investigating integration with other Solidity development tools like brownie and hardhat.

To learn more about how Warp works under the hood, follow us on Twitter. We will be posting further blogs explaining how we handled some of the more complex challenges involving modifiers, inheritance, and memory/storage copy semantics.

If you had any trouble with the tutorial above, or you just want to hang out, get in touch with us on the Nethermind Discord server.

We have a ton of cool projects at Nethermind, and we're always looking for passionate people to join us. If you're interested in working at the forefront of crypto, drop us an email with your CV at talent@nethermind.io. If you're interested in compilers, then the Warp team is interested in you.