Cairo: the Starknet way to writing safe code

TL;DR

You will write safe code, and you will be happy.

Kalzak

Follow

Nethermind.eth

--

Listen

Share

By

[Kalzak

](https://twitter.com/0xKalzak), with special acknowledgements to

[Erim Varis

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

[Rodrigo Pino

](https://twitter.com/rodro_pino)for discussions and reviews.

Intro

Developing secure and robust code can be quite a challenge, demanding both time and effort. The language you use can be a factor; some make it easy to write safe code, and some maybe don't as much as they should. Cairo, the native language of Starknet, is a language that makes writing secure code easier. Drawing inspiration from Rust, Cairo may initially present a bit of a learning curve, but it's designed in a way that promotes (and sometimes forces, hence the title) well-written code.

In this article, we'll explore some of Cairo's features that empower developers to write high-quality code. If you're a developer interested in what Cairo has to offer, this article is for you!

No payable functions

Starknet stands apart from most blockchain networks in a notable way — it lacks a native currency. Let's contrast this with Ethereum, where Ether serves as the native currency. Native currencies are transferred directly and don't have an underlying contract of any kind, but this means that protocols need two distinct methods for accepting value: one for the native currency and another for token-based currencies. The reduced payment complexity can lead to simpler logic, reducing the potential attack surface on your protocol.

Additionally, the concept of value-based reentrancy is no longer a concern since there aren't any payable functions. All value is transferred through tokens, including paying for network execution (gas) costs, thanks to Starknet's approach to account abstraction.

Name hashed storage slots

Cairo determines the storage slot for a given storage variable by hashing its name. This is unlike Solidity, which assigns storage slots incrementally as new variables are defined. You may have seen some codebases that storage gaps like uint256[50] __gap

to make space when dealing withproxies and inheritance to prevent storage collisions in case new storage variables are added. In Cairo, you don't need to worry about these types of storage collisions since the slots are determined by hashing the name. For the curious, the hashing algorithm used is sn_keccak

.

Just remember, you don't want to change the name of an existing storage variable when upgrading a contract because it'll point to a different slot than before, and you should keep storage variable names unique to avoid clashes between contracts when importing.

Upgradeability built-in

When you make the move to Cairo, proxy patterns will be a thing of the past. Starknet introduces the concept of a "class hash", where instead of each contract having its own bytecode, a contract will point to a given class hash, which contains all the logic. What sets Starknet apart is the ability for multiple contracts to point to a single class hash. Unlike Ethereum, for example, where there are thousands of different contracts with the same underlying ERC20 bytecode, on Starknet, all these contracts would point to the same class hash.

If you want to upgrade a contract, you can simply declare a new class hash to the Starknet network and pass that as an argument to replace_class_syscall()

in your contract.

In that one function call, you have now changed the class hash that your contract points to. No additional contracts or proxies needed! This streamlined approach to upgradeability is a significant improvement to overall safety, reducing the room for potential pitfalls, and combined with hashed storage slots, upgrading becomes a breeze — easy and safe.

Separated internal/external functions

In Cairo, all external functions must be defined in a separate area compared to internal functions. This stands in contrast to Solidity, where visibility can be specified per function, making grouping depending on the function visibility only a recommended practice rather than a mandate. While this doesn't necessarily directly impact security, a clear distinction between your internal logic and your entrypoint functions can greatly improve the overall structure and readability of your codebase.

Cheap execution means readable algorithms

The cost of execution on Starknet is far cheaper than that of Ethereum, which means you don't need to resort to extreme measures to craft heavily optimized yet potentially convoluted code. Now, we're not saying you should start writing O(n^3)

algorithms just because costs are lower, but that you can shift your focus towards sound algorithm design rather than obsessing over optimizing every single line. This approach often leads to code that is not only elegant but also more readable and less prone to errors.

For example, on more expensive chains where Solidity is used, you may find inline assembly is used in some places to write more efficient code and reduce gas costs, sometimes at the cost of readability. Make no mistake, at Nethermind, we love looking at optimization problems, and there is absolutely a place for inline assembly. When done right, it can have a great impact; however, experience is needed to ensure your code behaves exactly the way you intend it to. In Cairo, you can simply write well-designed code that's easy to read with less complexity while still enjoying the benefits of lower execution costs.

Immutable variables by default

As previously mentioned, Cairo is heavily inspired by Rust. This, along with the remaining points, will cover Rust features, which also pass their benefit to Cairo. Variables in Cairo are immutable by default, and only those explicitly marked with the mut

keyword can be modified. This design allows you to enforce compile-time guarantees regarding which variables can or cannot change. From a readability perspective, this feature is a great help to anybody reviewing the code as it provides clear insights into which variables are subject to modification and which will remain unchanged.

Safe type conversions

In Cairo, type conversions are a[safety-conscious] action, complete with checks to ensure no information is lost and the resulting data remains valid. In Solidity, if you want to convert a uint256

to a uint128

, the operation will truncate the remaining 128 bits, regardless of whether they contain non-zero data. In Cairo, the same conversion operation would not allow information to be lost.

This example only demonstrates the built-in integer type, but this feature really shines when dealing with custom types and composability. You can implement your own types with all the validations for try_into

for each type that you want it to be able to convert to. When it comes to composability, you can confidently build upon other protocols, knowing that the original developers have already implemented the validations for converting from one type to another, reducing the potential for edge cases or unintended consequences.

Option and Result traits

Sometimes, you may need to call a function that could fail or not return a value under specific inputs. In Solidity, handling such scenarios often involves designating certain values as “special’, for example, returning zero if no valid calculation exists or some non-zero value for valid outputs. In Cairo, the Option

trait can be used, and a return value can be represented as Some(value)

or None()

. This allows for explicit handling of return values from functions without having a special case representing “no-return”. While this may seem quite simple, consider a scenario where a developer is building upon the function of another protocol. They don’t need to know and handle this special value; they can simply handle the Some(value)

and None()

, and what’s more, the compiler actively enforces that all possible cases are considered when dealing with such values.

The Result

trait follows a similar concept, offering the flexibility to represent outcomes as either Ok(value)

or Err(error)

. This allows you to gracefully recover from Result

errors when calling functions. It’s important to note that some errors can still beunrecoverable, but now it’s possible to have functions that are designed to return errors that can be handled by your calling logic, and thanks to the strict compiler, this type of error handling must be done at compile time!

Conclusion

The Starknet network and Cairo language have a lot of features to encourage (and force) writing good code. We hope this high-level overview of some of these features has been helpful. Of course, these features shouldn’t be mistaken as a silver bullet for all security concerns. Business logic issues, cross-chain communication flaws, and other mistakes can still make their way into your code.

Interested in learning more about Cairo? If you’re a reader, theCairo Book provides all the detailed information you need. However, if you prefer hands-on learning, Starknet by Example is the way to go. Whichever method you choose, both resources will help you get started on your Cairo journey!

If you’re presently engaged in or contemplating a Starknet project and require security reviews or guidance, feel free to contact us at hello@nethermind.io.

Disclaimer

This article has been prepared for the general information and understanding of the readers. No representation or warranty, express or implied, is given by Nethermind as to the accuracy or completeness of the information or opinions contained in the above article, or as to the quality of any code or project that may be developed by the reader, including without limitation any code and projects written in Cairo.

About Nethermind Security

Nethermind Security

is the security arm of Nethermind, providing services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring.

Their work includes Solidity and Cairo smart contract auditing, building protocol-specific custom Forta Network detection bots, and formal verification services across ZK-circuit, EVM-based smart contract, and Starknet smart contract verification.