Vesta: Expanding the Universe of Blockchain Based Virtual Machines

Marston Connell

28 November, 2022

Abstract

Ethereum was the biggest milestone in blockchain technology since the inception of Bitcoin. It allowed developers to deploy programs on the network itself and give the ability to users to interact with these decentralized apps (dApps). This brought forth a wave of new applications that relied on the underlying blockchain as a verification mechanism. After some time though, developers looked to expand what a blockchain could do, and one of these expansions was the Cosmos ecosystem. Cosmos suddenly allowed developers to build purpose built blockchains almost as easily as writing a smart contract. With the introduction of IBC, this became an even better avenue for developers wanting to have sovereignty over their application. In this way, smart contracts have had a hard time finding a home in the Cosmos. If building a new blockchain is only marginally harder than building a smart contract, why would a new project not go the extra mile to have much greater control of their product and not rely on the tokenomics of the underlying protocol? So far we have seen a few major Cosmos chains be built with the purpose of being a Smart Contract platform, although not intentionally they have generally become testing grounds for new cosmos app-chains. If we look at Terra Luna, many of their projects moved after the collapse, but one of the larger projects, Kujira, made the move to build their own chain. Even DyDx, an EVM based app is building on Cosmos after outscaling what the EVM can do for them. It's clear that virtual machines in Cosmos need to be rethought of as prototyping tools rather than fully fledged apps.

Contents

1 Introduction

Vesta is a Smart Contract platform. Vesta is boring because Vesta doesn't do anything by itself. Vesta will only do exactly what a developer wants, and that's a good thing. Vesta doesn't aim to replace any particular virtual machine currently in use in the Cosmos ecosystem, for apps wishing to interface with Osmosis liquidity. it makes a lot of sense to build robust applications on the Osmosis CosmWasm deployment. What Vesta does aim to do is bring attention to the way that developers have been using existing VMs and point out how overly complicated the current state of Smart Contracts are. Developers wanting to quickly prototype a new system should not need to use a systems language like Rust, but they also shouldn't need to use a domain specific language like Solidity. Vesta builds on what developers already know and allows them to build their Smart Contracts and Web-based front-ends in the same language. Vesta takes advantage of the RomaVM which was custom built for this deployment, however it is open-source and can be used by anyone for free.

2 Roma

Roma is a Javascript interpreter running inside a CosmosSDK module. Roma doesn't have the ability to interface with any other modules on its own which makes it very safe for the chain it is being deployed on. However, with the help of an injection system, Roma becomes much more powerful. Any module can be injected into Roma including both core CosmosSDK modules and custom chain-specific modules that Roma could be deployed alongside. Roman theoretically could have injections for itself allowing Smart Contracts to alter how Smart Contracts behave on the network. Being interpreted allows Roma to do much more than execute arbitrary code, it allows Roma to respond to user requests in ways that could only be described as meta-contracts.

3 Core Priniples

Roma was built with a set of core principles. Throughout the design process, every decision was made in respect to these ideas. They are as follows:

- All contracts must not reuse source code unless the user explicitly chooses to.
- All contracts must be able to control what data is public and what data is private.
- All contracts must be able to contribute to other contracts.
- All contracts must be able to alter the state of modules given the modules have allowed for it.
- All contracts must be able to reference themselves and other contracts as first-class citizens alongside modules.

4 Smart Contracts

Every Smart Contract contains a few pieces of information that allow Roma to know what it is supposed to execute and when. Smart Contracts start by being written in Javascript, these JS programs are then stored to the network. Every time a contract is stored to the network it is given an incrementing ID value, this value is used to reference this specific script and will never change. From there, a developer can instantiate a contract by passing in both a name for the instance and a code value corresponding to the ID given to a block of JS source code. From here anytime a user or developer wishes to interact with this instance, they can do so by referencing its name rather than some long arbitrary address hash.

These instances don't hold a virtual machine themselves, but merely hold reference to the source code to which a virtual machine should run. This means that a contract can change which source code it is built with merely by changing the ID number. This gives library makers the ability to update their libraries without needing contract makers to update the libraries manually. However if a developer does prefer, they can pass in the version number of the library to ensure it is being executed as if it was never updated. This way is the safest as libraries could potentially update to malicious code.

5 Composability

With the idea that contracts must not reuse source code without purpose, Roma needed a new way for contracts to be written. In legacy Smart Contract systems, when users wish to import a library they must do so by telling the compiler to download linked source code, add that source code to their work and compile all of it into one large program. This compilation step has plenty of benefits, however in blockchain, one of the most important dimensions a blockchain can consider is space. More specifically the amount of data being stored on the network should remain small since the validators must carry this cost. When uploading two NFT contracts that rely on the same OpenZepplin library, the massive library is uploaded in its entirety to the network twice. This costs the user more to upload this data and it costs the validators more to keep both copies of identical work.

This problem introduces us to the solution that is not all that dissimilar from an on-chain package manager. Using custom Javascript hooks, Roma is able to import existing contract deployments at runtime. Better yet, these imported contracts encapsulation preferences are respected. If a contract declares a helper function and doesn't export it, that function can still be run by an exported function but it can never be called on its own. This gives library builders much more control over how developers use their libraries. On top of this, contracts are references by name and allow developers to potentially allow for importing a library based on user input at runtime.

To manage this, we essentially build a second virtual machine to parse and build an Abstract Syntax Tree (AST) of the imported contract and pass that tree into the place of the import in our original contract. This strips out the data of the contract that isn't used and can allow us to easily import contracts that import other contracts themselves.

6 Interoperability

One of the most important Smart Contract features is the ability to interact with other contracts. Applications like UniSwap wouldn't exist without this technology, and as such it is something that Roma needed. We didn't want to follow suit in the way that contracts commonly talk to each other on existing platforms, instead we leveraged the ability to name a contract in the same way we make web requests by URL. Fetch is a common function in Javascript that allows applications to make web requests, in Roma we repurpose this function to make requests to other contracts.

When making requests, we choose what type of request we want to make, either a POST or a GET request. Posting allows the user to make a transaction on the contract, getting allows the user to query the state of the contract. To make a request, we simply pass in the name of the contract, the function we wish to execute, what type of request we want to make and any arguments we wish to pass in. This method of interaction allows us to use user input to define what contract we make requests to, allowing for much more than statically linked contracts on existing platforms.

7 Conclusion