Vesta: Expanding the Universe of Blockchain-Based Virtual Machines

Marston Connell

14 December, 2022

Abstract

Ethereum was the biggest milestone in blockchain technology since the inception of Bitcoin.[1] 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.[6] With the introduction of IBC,[5] this became an even better avenue for developers wanting to have sovereignty over their applications. 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 its product and not rely on the token economics of the underlying protocol? So far we have seen a few major Cosmos chains be built to be 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 its projects moved after the collapse, but one of the larger projects, Kujira, made the move to build a sovereign chain. Even DyDx, an EVM-based app is building on Cosmos after out scaling what the EVM can do for them. Virtual machines in Cosmos need to be rethought 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 is. 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 VM

Roma is a Javascript interpreter running inside a CosmosSDK module. Roma can't 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. Roma, 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 Principles

Roma was built with a set of core principles. Throughout the design process, every decision was made with 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 in the network. Every time a contract is stored in 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 a 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 OpenZeppelin 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 a solution that is not all that dissimilar from an on-chain package manager. Using custom Javascript hooks, Roma can 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)[4] 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.[3]

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, and 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 Block Usage

One of the main benefits of building a new blockchain rather than building smart contracts is often that a developer could want an event to trigger every block. This is impossible in traditional smart contract systems as they rely on a user calling a function, this restricts the possibilities of what smart contracts are capable of and prevents a plethora of both economic and technical models from being possible. Some third parties have taken it upon themselves to build automated runners, however, they require trust and that is sometimes impossible to give in decentralized applications.[2] In Roma, developers can load their smart contracts with funds of the respective denomination as described in the module's parameters. These funds can then be used to pay for jobs run automatically by the module. A developer can specify a contract, the function to be executed by the module, and how many blocks are between jobs. For example, if a developer specifies that a contract runs a mint function every 20 blocks, whenever $block\_height\%interval == 0$ then the mint function is called on the given contract. This function is not free to run and will drain the contract of funds specified by the module parameters. If the contract does not have enough funds to be run, it is never run and is removed from the job queue, meaning it won't be run again without the developer manually restarting the job.

8 Conclusion

Vesta is not only a new smart contracting platform, but it is an experiment into what is possible in blockchain tech when we treat smart contracts as higher-class citizens within the blockchain structure. When smart contracts are given the ability to meld deeply into not only the blockchain composable modules but the underlying consensus mechanisms, they are given a much greater opportunity to push forward what is possible in web3. Bringing a language that is familiar to web developers into web3, often built as experiences in the browser, makes onboarding new blockchain developers easier and allows the developer experience to be homogenous.

References

[1] Vitalik Buterin. *Ethereum*. 2014. URL: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[2] *Croncat*. URL: https://cron.cat/.

[3] *Fetch*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch.

[4] Python Software Foundation. *Python - Abstract Syntax Trees*. URL: https://docs.python.org/3/library/ast.html.

[5] *IBC-Go*. URL: https://github.com/cosmos/ibc-go.

[6] Ethan Buchman Jae Kwon. *A Network of Distributed Ledgers*. URL: https://v1.cosmos.network/resources/whitepaper.