ADVM Specification (Major Redesign!)

The Adamnite Virtual Machine (ADVM) is a bytecode interpreter that executes native smart contracts and programs for Adamnite's Blockchain. In the technical paper, the current specification is essentially identical to Ethereum's Virtual Machine (EVM), save for the mention of specific opcodes that support 32 byte and 64 byte calculations. However, this is not ideal, as the EVM itself is rigid, only has support for a limited number of high-level languages (requiring compatibility to be built on top), and has other efficiency related issues that often result in clunk and slow decentralized applications. Thus, we propose an alternative design that extends WebAssembly (WASM) for the purpose of executing operations on Adamnite's native distributed ledger.

The ADVM will now be a virtual machine based mostly on WASM, but extended and slightly modified to support blockchain-related operations. This is in theory similar to current implementations such as EWASM and the EOS VM. However, we make slight improvements for optimization with current This change is due to several reasons:

- WASM is natively 32/64 bit, thus providing compatibility with a majority of hardware.
- WASM is used ubiquitously across the Internet: WASM bytecode supports compilation from Python, C++, JavaScript, and a host of other high-level programming languages.
- A1 can be a Pythonic Smart Contract language for high-level development and with features native to the Adamnite Protocol. Furthermore, due to WASM's support of functions and some form of standard libraries, it makes it an ideal target for A1.

We expand WASM for blockchain-related computations, specifying 256-bit operations for elliptic curve cryptography as functions built directly on top of the WASM library. Note that the ADVM is still an independent VM extending WASM; it can be thought of as an implementation of the common WASM Engine Paradigm. The major requirements for the ADVM are deterministic execution, a bounded computational model, and secure compilation to bytecode.

ADVM will have essentially the same op code set as a traditional WASM Engine. The specification will largely follow the general model established by EOS, with changes as needed for Adamnite's specific model. The ADVM will be built from the ground up, and will be a WASM engine optimized for blockchain development. Certain functions and opcodes, such as those for accessing account or overall state, may be implemented. Smart contracts executing on the ADVM should follow an application state paradigm; contracts should be stored as applications on the distributed ledger, being able to both read and manipulate state. Ideally, deploying a smart contract that does not involve intensive computations or does not involve large amounts of state (i.e. simple mutli-sig wallets, atomic transactions) will cost the same as a normal transaction. Contracts that require these features will likely be more expensive: however, due to Adamnite's efficient computational model, these costs still will be far below those of traditional Proof Of Work chains.

The smart contract execution model, for now, should follow the traditional paradigm established by Ethereum, Algorand, and other smart contract platforms. Opcodes matching Intel X86 instructions should be metered with specific and predictable transaction fees. Once a smart contract is compiled down to bytecode, it should be deployed to mainnet along with its ABI. The mainnet deployment process is described below:

- Contract Calling (and storage) are broken into two pieces: on-chain validation and offchain storage. A contract's account-state parameters and most recent state should be hashed with the SHA-512 function and stored on a binary merkle tree, as described in the documentation.
- Compiled Contract Code and overall storage are stored off-chain, either in a remote file database such as IPFS or other decentralized directory. Alternatively, this can be stored in a regular database. A reference to this is stored on-chain within the account parameters (similar to how metadata is stored for NFTs)
- When a user makes a smart contract call, the validator processes the call by first checking via a cryptographic proof that the off-chain contract has not been modified in any way and verifying that the transaction is correctly formed. It then evaluates the contract and makes the appropriate changes to the state by doing the following: evaluating the contract via an API call, and then ordering transactions accordingly. If the user's variables (such as total amount) change in the short time period before the transactions associated with the contract call are pushed to mainnet, then the transactions are simply not added by the validators.
- Once this process has been completed, the off-chain contract will update its storage, while the on-chain representation will update the hash it has stored.

For now, contract data will be stored in a binary merkle tree; eventually, this will be formalized and extended to a verkle tree, where vector commitments and proofs will be used to validate that off-chain contract code has not changed in nature. Verkle Trees will also for account data to be manipulated in a much faster way.

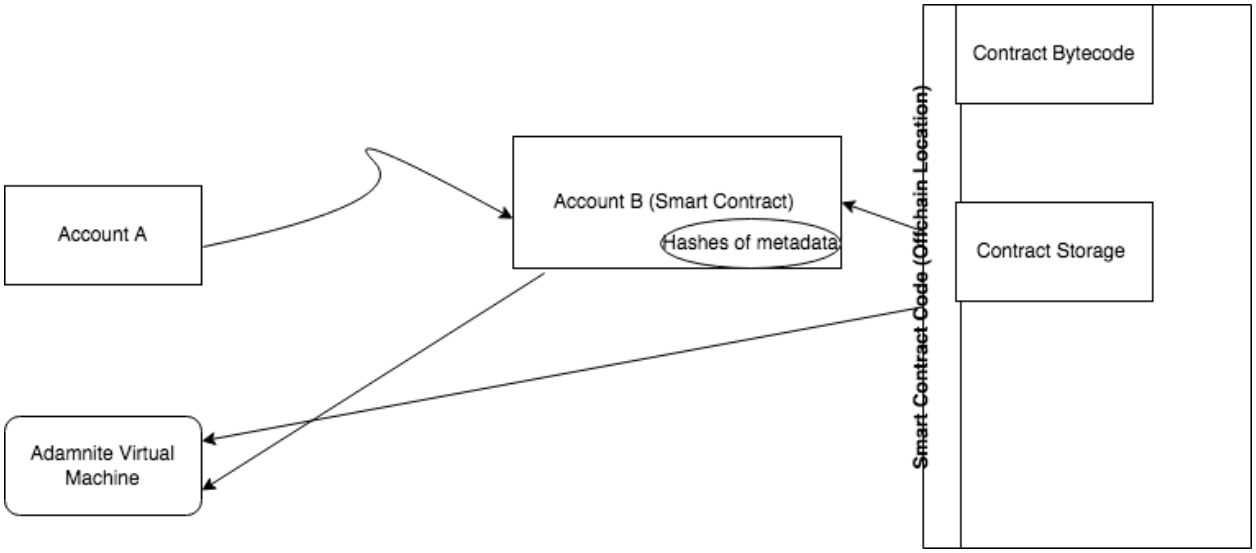
VM Blockchain Specific Opcodes/Functions

The virtual machine, beyond being a generalized WASM implementation, will also have several blockchain specific functions that will enable to execute cryptocurrency-specific message calls. The initial model will follow the opcode metering model established by Ethereum and followed by other chains: though int the future, a transition to a rent model (as with EOS) could also take place.

- The getAddress retrieves the address for the targeted account/contract
- The CREATE opcode defines the creation of a new autonomous account/smart contract

- Add ATE defines the total amount of additional ATE needed to fufill the execution of the current set of instructions.
- Other parameters such as TIMESTAMP, BLOCKSIZE, WITNESS_LIST are also available. These are self-explanatory in their definition.
- More opcodes will be added in the near future.

As contract storage itself is stored off-chain, it can be easily manipulated: there is no need to define different parameters for runtime manipulation and long-term persistence. Below, a model for the proposed runtime ecosystem for executing a message/smart contract call (and therefore an overall state transition) is shown:



A future improvementl on this model will be storing the entirety of the state externally, with the only on-chain representation being the world state that is simply a k-ary tree that stores cryptographic proofs of the external storage.

A1 can simply compile