IBC Wasm Client Review

Provided by Confio OÜ
Commissioned by Interchain GmbH

Draft v0.4
Date: July 10, 2023

Index

1. Requirements

Before we can begin an analysis of the design, how it could be improved, or what errors it may have, we must clearly express the goals. These requirements are not technical details, but rather a design of how the product should work, which capabilities it should have, and which it should not have. They define the metrics against which any technical implementation or even architecture may be judged.

The following are understood as requirements of the project as a whole, to give a better understanding of the conditions under which this code will be used. The items in red will be at least partially covered by this audit, either as part of or in addition to the questions listed in the Scope of Work.

Required for launch

- No Exploitable Bugs (given proper validation code in contracts).
- No DoS Attack vectors (for any wasm code).
- Code integrates with Cosmos SDK 0.47 and recent Wasmd releases.
- Avoid circular dependencies.
- New wasm client types (code id/byte code) must be approved by governance vote before clients can be made.
- Once the Wasm client type is approved, anyone can permissionlessly create a new client instance of with this code for a specific counterparty chain.
- Instantiated wasm clients interoperate with the rest of the IBC-go stack, specifically in creating connections and validating packets.
- Client contracts write data to the provable store in the format defined by IBC.
- Question: Client contracts cannot call into other systems or effect changes beyond the IBC state under their management
- Question: Client contracts cannot query any chain state beyond their local storage
- Tested client contract that properly validates Tendermint headers and IAVL proofs.
- Tested client contract that properly validates Grandpa headers and SMT proofs.
- Contracts can be written in Rust and compiled to wasm.
- Runs in chains that do not include x/wasm as well as those already embedding x/wasm.

Desired for launch

- CosmWasm VM is used efficiently, enabling existing optimizations.
- Loosely coupled dependencies, such that IBC-go and wasmd can release separately, and both upgrade easily to newer Cosmos SDK versions.
- Wasm client supports all queries and messages and events expected by the go relayer as well as hyperspace.
- Connections between two Cosmos SDK chains using the wasm Tendermint implementation is fully supported by the go relayer.
- Connections between Cosmos SDK and Substrate chain is fully supported via the hyperspace relayer.

- Clients can utilize most of the existing CosmWasm contract libraries (eg. cw-storage-plus is incompatible as it defines a different byte format for storage).
- Visibility of which wasm byte code is used for a given client.
- Clients can use CosmWasm precompiles for large speed-up in crypto signature validation.
- Verify that prost (the major Rust protobuf library) can be used inside contracts.
- Minimize code duplication from other libraries (eg wasmd).
- Remove global variables or ensure correct usage

For future version

- Support for Cosmos SDK 0.50 (and beyond)
- Custom contract-side libraries to interact with IBC data structures (like easily writing compliant state data)
- Support for importing no-std libraries in contracts (much Polkadot code is no-std)
- Ability to migrate a client to another byte code, while maintaining all existing connections and channels. The mechanism and naming are not defined here, but the general requirement are.
    - Question: should migration be authorized by chain governance on the counterparty chain, or by chain governance on the chain upgrading the client? Or another party?
- In-Protocol mechanism to verify the source code for a given client type

For the remainder of the document, I will label recommendations and issues with the following color coding:

- Green items are informational suggestions that are handled well enough in the current implementation.
- Orange items are important issues to address, but not security critical.
- Red items are security related and must be addressed.

2. Architecture Review: CosmWasm Stack

This section provides a high-level review of the architecture of the key subsystems that are integrated with the Wasm light client. It explains key design decisions and requirements of these subsystems individually and then examines the integration of said systems, highlighting any problems or areas of special concern in the design.

This section provides an overview of the various components of the "CosmWasm Stack", looking at the capabilities they provide, as well as their requirements for proper functioning.

2.a Wasmer

Wasmer (TODO: link) is a high-performance engine able to execute Wasm byte code deterministically, which has been deployed in a number of blockchain settings, such as NEAR and Space Mesh. Rather than run a Wasm interpreter, it achieves performance comparable to native performance by first compiling the Wasm code into native byte code along with instrumentation, and then executing this native code. This compilation output can be serialized and cached to reduce startup time in the future.

The compilation step uses a single pass which is a relatively simple compiler, but O(N) and unlike LLVM not considered a potential DoS attack target. During this compilation step, one can insert gas metering instructions, which will trigger a call to the gas counter at every branching instruction. One can also inspect the wasm byte code and do things like reject any floating-point operation.

Every wasm file includes not just the byte code and data blob (static / const data), but also a set of imports and exports. Exports are entry points that can be called from outside (from the VM) and imports are methods it requires to be exposed from the VM that it can call into. The CosmWasm contract specification centers around the APIs and the semantics of these exports and imports, which I will cover below.

During execution, Wasmer registers handlers for some control signals which are used as traps inside the generated native code when the contract panics or hits a resource limit (memory or gas). It is very important that no other system registers signal handlers on those interrupts. This also means that there should only be one instance of the Wasmer code in any given process.

The concurrency story in Wasmer (running contracts in parallel) is generally supported, which allows querying contracts while others execute. There have been some issues around this with the Wasmer 3.0 / 4.0 upgrade and this makes it important for higher-level entry points from CosmWasm when calling into Wasmer, where we ensure these can be safely run concurrently.

2.b. CosmWasm-VM

CosmWasm-VM is part of the CosmWasm/cosmwasm repo and includes all logic for using Wasmer properly to call CosmWasm contracts. This involves customizing the compiler pipeline, caching for optimized performance, providing required imports, as well as providing a high-level Rust API to call into a contract's defined exports.

Since early benchmarks showed the compilation phase to be by far the most time-expensive step in the contract execution, much care was put into a Cache system, which is used extensively by all operations with the contracts. This object puts some constraints on usage, so I will elucidate the functionality and the requirements here.

The other main systems we need to be aware of are the exact exports cosmwasm-vm expects from the contracts as well as the imports the contracts expect from their environment. I will briefly cover their functionality here as well, so we have a common language when discussing higher-level integration into the IBC-go framework.

Cache Functionality

The cache system is given a directory that is fully under its control as well as a per-node-configurable memory limit for in-memory caching. When the wasm code is first uploaded, it is sent to the cache system for storage and assigned a cryptographic hash (called a checksum) that can be used to identify it later. The raw wasm code is no longer needed in any other system but can be retrieved by requesting it from the cache given its checksum.

Upon upload, the cache stores the bytecode in a special wasm directory, identified by its checksum. It then compiles the code to a native machine code, adding gas metering and ensuring only deterministic operations are involved. (This takes a few hundred milliseconds on most machines and typically costs several million Cosmos SDK gas, encouraging the reuse of bytecode between multiple contracts) The native code is stored in another directory, also indexed by the same checksum. It should be noted this is just an optimization and upgrading the CosmWasm-VM or state syncing wasm code to another machine will invalidate this native code and force a recompilation step.

Loading precompiled code and instantiating a contract still takes a few milliseconds (about a 50x speedup), so effort is taken to optimize that as well. There is an in-memory cache that holds Wasmer's instance objects that can cheaply be cloned. Starting up a contract from this cache takes a few hundred *microseconds* (a 10-20x speedup from above) and when used produces optimal performance.

There are two ways a contract may be in this in-memory cache. The first way is that chain governance may "pin" the contract to the cache, ensuring all nodes maintain this bytecode in the in-memory cache always (and preload it on restart). This "permanently pinned" list should be limited to less than approximately 100 contracts for memory constraints, and thus cannot be permissionless. These pinned contracts also are charged a much smaller gas price on startup (1k gas rather than 60k gas for contracts in general).

The other way that contracts are stored in memory is via the LRU cache. Each node may specify a size in megabytes they want to dedicate to this to speed up operations. Bytecode is

stored in an LRU cache which discards the *least-recently-used* module when a new one is added. On a cache hit, we get a very fast startup; on a cache miss, we load from the disk cache. Since there are no guarantees, the caller is still charged the full gas, but it allows node operators to greatly reduce processing time, as the most common contracts are often in the cache, so probabilistically a large percentage of the calls should hit the cache.

Warning: If there are multiple caches, they *must* use different backing directories. Accessing the same files is a clear source of race conditions. We have not extensively tested concurrency using two different caches with different backing directories. It is not known to have any issues but deserves targeted testing if used.

Entry Points

A contract exports the following entry points that can be called from the VM. All of them are passed *Env*, which maintains block time and height, as well as the contract address. Some are provided *MessageInfo*, which contains the caller's address as well as any native tokens being transferred along with the call.

They also have access to the *Api* (pure functions like validating addresses or performing crypto signature verification), the *Querier* (a read-only stateful function to read information about other parts of the system – be it the bank balance of this contract, or the state of an IBC channel), and *Storage* (which provides access to a key-value store, which can be any subset of the full IAVL tree)

- *Instantiate* – Called once to produce the initial contract state. This gets a new address and an empty, unique space in the KV-store, which it can initialize. It accepts calls from any sender but may store the original sender of this call for later use in authorization.
- *Execute* – This is the typical entry point for any state-changing transition in the contract and may be called many times. It receives *MessageInfo* and can use the sender to permission actions, or decide which keys to touch (eg. whom I am sending tokens from)
- *Query* – This is the typical entry point for any off-chain or on-chain query. It is called many times, with no authentication information and with only read-only access to the *Storage*.
- *Sudo* – This is like *Execute* but it doesn't include any sender info. It should never be exposed to external users, but only called from trusted modules. The authentication happens in the runtime outside of the contract. It is designed for use by eg governance-controlled entry points, or access from Go modules.
- *Migrate* – This is a special entry point for a contract used for the case when we change the code running that contract. Like instantiate, it is called once when setting a new code hash, and like *Sudo* it assumes authentication has been performed already in the runtime outside of the contract. The new contract runs one time with the entry point and an optional message. Its job is to ensure the old state is valid (upgrading from something it knows how to upgrade from) and then to perform any transformations to the data as needed. In practice, this often relies on the cw2 spec

in order to store metadata on which contract is/was using this storage space, so it is easy for the new code to detect the previous code version when *Migrate* is run.

(There are other high-level entry points for IBC-enabled contracts, low-level memory access, as well as a *reply* entry point for handling submessage responses, but these are not important in the context of a wasm light client and we don't dig into them here)

Imports

The Wasm imports provided to the contract are wrapped up into structures for the Rust code, called *Api, Querier,* and *Storage* (referenced above). They have the following functionality:

Api exposes pure functions around verifying addresses and verifying cryptographic signatures. The cryptographic functions are provided as precompiles, that can be used by any light client to greatly improve the performance (around 8-10x) and significantly reduce the contract size. This should always be enabled.

Querier allows a contract to make any read-only query that is contained within the QueryRequest enum. This is exposed to all CosmWasm contracts, but it is unclear if this is needed and desired by wasm light client, or whether it should just always return error on all queries.

Storage allows reading, writing, deleting, and iterating on the (prefixed) key value store passed to the contract. The Rust type system restricts at compile-time it to reading and iterating if we get a non-mutable reference (in queries), which we also enforce in the CosmWasm-VM.

2.c. WasmVM

The central purpose of WasmVM is to provide safe cross-language bindings between the Go Cosmos SDK code and the Rust code in which the CosmWasm VM and Wasmer are written in. The repository consists of both the Rust and the Golang side of the bindings and deals with the FFI interface.

We need to both expose complex Rust objects, like the Wasm VM Cache, to the Go side over FFI, as well as pass references to complex Go objects, like the KV Store and a Querier, to the Rust side. You probably don't want to dig into that code but can take a brief look to see what level of pointers is going on there.

More operationally, the Rust code is compiled into shared libraries for each supported platform in the CI during releases. These libraries are then conditionally linked based on the go build system. Look at the *libwasmvm.** shared libraries and *link_*.go* in the wasmvm directory. Critically, on non-muslc platforms the shared library *is not embedded* in the final Go binary, meaning you need to use Alpine to make static builds, or ensure a copy of the proper dll is in your library path on the production system. This is especially important on

systems where you compile on one machine and copy binaries on the actual nodes that are running the code.

This issue should be known by all node operators on CosmWasm chains, but needs to be communicated to any other non-CosmWasm chain rolling out Wasm Light Clients.

2.d. x/wasm

This is commonly referred to as *wasmd*; however, wasmd is the repository and the sample Cosmos SDK binary that launches a minimal chain including CosmWasm. No one uses *wasmd* in production outside of testnets. Rather, the wasmd repository includes the *x/wasm* module, which provides all the functionality. Typically a chain will import *x/wasm* and then wire it up properly in their *app.go* file.

x/wasm defines three core concepts around contracts – code hash, code id, and contract address. The code hash is a unique identifier tied to the actual wasm code that can be used as a universal identifier. This is used in all communication with Wasmvm. However, it is difficult for users to parse such long hex strings, so we introduced code ids, which are an auto-incrementing integer counter that is mapped to the actual code hash, as well as storing a bit of metadata.

While code hashes and code ids identify the actual byte code, they don't yet make a contract. A contract must have a unique address and account with which to interact with the rest of the blockchain modules, as well as some internal storage which it is fully in control of. As the cost of uploading new wasm code is relatively high, we create and encourage a 1-to-N mapping between code ids and contracts. Each contract maintains a reference to the code id (and thus code hash), as well as a unique address. A unique storage space under the wasm substore is then reserved for the exclusive use of that contract. The prefix *0x03<address bytes>* is used, and under which the contract can write any key/value pairs it chooses.

Since we need to create a new address and account for each contract, there arose an issue about address squatting. Basically, if you could predict the address that would be generated when instantiating a contract, you could send a few tokens to that address first to create a BaseAccount or VestingAccount. Previously, the contract instantiation would fail if the address was in use, but this was since patched with more robust handling. We informed the IBC-go team of the same issue in ICA address generation, which was also patched in the ICA implementation (but not rolled out on the hub fast enough and did create some issues with the initial QuickSilver launch). If any accounts are created in IBC Wasm Light Client, they must use a similar algorithm to be robust.

Although it won't be used in the Wasm light client, the permissioned/permissionless dichotomy of *x/wasm* is worth understanding to inform other design decisions. There are two main places where governance permission can be applied. The first is the actual uploading of the code, the second is the instantiation of the contract. One use cases requires a governance vote to actually execute StoreCode, which also sets an allow-list of who can instantate that code hash (which could be everybody). This is the most powerful block, but

makes a lot of workflows difficult, and even voting on a proposal with the entire wasm bytecode inside of it becomes extremely gas intensive.

A lighter workflow which fits most needs is to allow anyone to upload wasm bytecode (paying the large gas costs themselves for the compilation). However, by default no one may instantiate that code. They can either raise a governance proposal for a single instantiation of the contract, or to set the allow list on this contract (to their team or to open up to everyone). This approach still requires governance to explicitly opt-in to the contract code before it can be *run* on the blockchain, but uses much smaller governance proposals. If you are interested, it is worth reading the documentation of how to configure permissioned CosmWasm.

Cross-Contract Calls

Another feature baked into wasmd, which should most likely be prohibited in the IBC Wasm Light Client (IBC-Go team: please confirm or correct this), is the ability to call other contracts or indeed native modules like bank and staking. To avoid the possibility of reentrancy attacks, we don't allow direct calls to other modules, like with the read-only *Querier*, but rather contracts return a list of serialized messages that should be dispatched. They can also register for a *reply* callback on the successful or unsuccessful execution of that message in order to build longer workflows that depend on parsing the results of those messages.

However, this is a return value and a contract implicitly assumes this code is executed. We cannot simply ignore such messages, but rather if the IBC Wasm Light Client implementation will not execute such messages, it must return an error and abort execution if any messages are returned from the contract. Without that guarantees, the proper functioning of the contract cannot be guaranteed.

If you do wish to allow cross-contract calls from IBC Wasm Light Client, then the handling of such messages should be implemented in accordance with the specified semantics.

Events

In addition, contracts can set and return several custom events. This is likely an important feature to maintain in the IBC Wasm Light Client. The standard events as well as how custom events are constructed are well defined. Beyond the standard events which can be used to track all contract calls, contracts may return arbitrary attributes that will be appended to a standard *wasm* type event.

Contracts can also return entire events with custom types. However, the event type is auto-prefixed with *wasm-* so *stake* becomes *wasm-stake*. Beyond that, a *_contract_address* attribute is added, and no custom attributes starting with _ are allowed (they error the transaction if returned).

3. Architecture Review: IBC-Go Stack

3.a. Versioning and Dependencies

Even before the integration of the wasm light client, there is a relatively tight coupling between Cosmos SDK, Wasmd, and IBC Go packages, which has made upgrading difficult. Now that the Cosmos SDK team has picked up serious velocity and plans to release breaking SDK versions at least twice a year, with expectation these downstream packages are quickly updated, we need to pay special attention to enabling more agile development cycles, as well as maintaining support for multiple Cosmos SDK versions.

Currently, a given IBC Go version depends on a specific minor version of the SDK. It must explicitly choose 0.45, 0.47 or 0.50 and will only run in app chains that are using the same version of the SDK. Likewise, Wasmd must target a specific version of Cosmos SDK, but it must also target a specific version of IBC Go. When IBC Go moved from 1.0 to 3.0 still on Cosmos SDK 0.45, wasmd was able to easily upgrade. However, when there is a new version of the SDK and often some alpha branch of IBC Go with untested support for the new Cosmos SDK version, this becomes a lot trickier.

Additionally, wasmd depends on Wasmvm, which we upgrade independently of the Cosmos SDK version. That is when we go from Wasmd 0.x to wasmd 0.x+1 we may bump *either* the Cosmos SDK version or the Wasmvm version, but not both.

If the IBC Go repo were to import wasmd directly, we would have circular dependencies. If the IBC Go repo imports Wasmvm directly, we now have another shared dependency, requiring wasmd and IBC-go teams to coordinate upgrades not only for Cosmos SDK upgrades but also for Wasmvm upgrades.

3.b. Light Client Interfaces

According to the IBC Specification, the light client implementation MUST store the client and consensus data as opaque bytes under very specific paths. The prefix used to store these is not clarified but left to the implementation of the IBC handler on each chain.

> *"These functions MUST be permissioned to the IBC handler module (the implementation of which is described in separate standards) only, so only the IBC handler module can* ***set*** *or* ***delete*** *the paths that can be read by* ***get****. This can possibly be implemented as a sub-store (prefixed key-space) of a larger key/value store used by the entire state machine."*

Given that this IBC handler stores information for multiple clients in the following paths *clients/{identifier}/clientState*, *clients/{identifier}/consensusStates/{height},* and *clients/{identifier}/connections*, we must assume the same prefix must be used over all clients exposed by the IBC handler on one state machine.

We thus need to ensure that a given Wasm light client contract has access to write to all data after the prefix *clients/{identifier}/* under the canonical IBC handler.

3.c. Integration Requirements

As we saw above, the *x/wasm* module imposes a lot of restrictions on the contract usage in order to sandbox them, as well as a number of functionalities for cross-contract composition. However, much of this composition functionality (like dispatching messages) is undesirable for the Wasm light client; and the storage isolation pattern is incompatible with the storage prefix required by the IBC specification.

Given that, direct integration with *wasmvm* is the highest-level connection that can still be customized to the particular needs of this project. However, special care must be taken both to ensure proper usage (rolling back storage writes, gas metering, etc) as well as integration with a potential *x/wasm* module on the same chain. The issues of storage, gas, etc are covered below in this document under "Specific Questions Answered". **TODO: recommended usage**

The point of proper compatibility with an existing *x/wasm* module is critical to the successful and safe adoption of the IBC Wasm Light Client module. At a minimum, it means that the module must be versioned to maintain releases with the same functionality over multiple versions of wasmvm (at least the last two), so as to allow easy upgrades of the wasmd chains. This means they link to the same DLL and Rust code.

Beyond that, it would be ideal if both the x/wasm module and the ibc light client shared the same *wasmvm.VM* to ensure compatibility. This would mean the same code hashes could be accessed by both *x/wasm* as well as IBC Wasm Light Client, but because they store data in different address spaces, there would be no overlap.

4. General Architecture Recommendations

1. Recommendation: The wasm client should live in its own Go mod, so it can safely import both wasmd/wasmvm, ibc-go and cosmos sdk, but not block their upgrades. Additionally, it should have a unique versioning system as you may want to maintain multiple versions of the wasm light client for one version of the ibc-go repo. (eg. Wasmd 0.41 and 0.42 would both target Cosmos SDK 0.47 and ibc-go 7.0 but wasmd 0.42 will use wasmvm 1.3, while wasmd 0.41 used wasmvm 1.2. It would be beneficial to be able to tag two releases of ibc wasm light client (with different major versions) that both use ibc-go 7.0, but targeting different wasmd/wasmvm releases.)

2. Change Requested: IBC Wasm Light Client **should** directly use Wasmvm. However, it must ensure it uses the same Wasmvm.VM instance as the x/wasm module if present. The wasm Keeper instantiates a VM in the constructor, which can be overridden with an option argument. In order to share them, the recommendation is that IBC wasm light client construct Wasmvm with the same arguments as *x/wasm* and pass this into the x/wasm keeper if the chain supports both IBC Wasm Light Client. I have an submitted a PR to ensure this use-case is properly handled: https://github.com/CosmWasm/wasmd/pull/1494 which will require wasmd 0.41.

3. Possible Improvement: It would make cleaner abstractions to avoid the global singleton WasmVm. However, in practice, we do want one WasmVM instance called by every location (including x/wasm keeper). If a future ibc-go version makes it possible to safely embed this in the keeper, it would be nice to do so.

4. Change Requested: Assuming it is desired behavior not to allow IBC Wasm Light Client contracts to execute sub messages, the proper behavior would be to check if *len(resp.Messages) > 0* in callContract return and error if so, which would abort execution.

5. Recommendation: Assuming it is desired behavior to allow contracts to emit custom events, the proper behavior would be to check the callContract return value and emit something like the defined format. It may be desirable to make some changes here, such as no longer prefixing custom event types with *wasm-* to allow a contract to emit any event it desired. Potentially, you could emit the standard execute/sudo events as well, but that only makes sense if there is a unique "contract address" for each light client.

6. Change Required: Do not use the Execute entry point in callContract, rather the Sudo entry point. This eliminates the need for the fake MessageInfo and is guaranteed to be uncallable by anything except Go modules. In wasmd one can enable a chain governance proposal to call this entry point (if you have a x/wasm address, which we don't have), but no user message nor contract call can access it.

5. Specific Questions Answered

Securely interacting with Contracts

At the moment, when we makes calls to the contract the Sender in MessageInfo is am empty string, but we were wondering if it would be more secure to pass in a 08-wasm module address or something else that contracts can check before executing any calls to make sure that they are being called by the 08-wasm module and not by anybody else

Do not use the Execute entry point, rather the Sudo entry point. That doesn't require any sender info and is guaranteed to be uncallable by anything except Go modules. In wasmd one can enable a chain governance proposal to call this entry point, but not user message nor contract call can access it.

We want to make sure that only the 08-wasm module is able to call the contracts. Do you see any security holes in the design that could lead to other actors calling the light client contracts?

Again, I would ensure that you use the Sudo entry point, which provides a high level of design security. Furthermore, if the wasm clients are stored in another kv namespace than

the normal x/wasm contracts, then it is impossible for any call via that module to interact with the contract's state.

Is gas consumption logic when storing a contract (here and here), and when instantiating the contract (here), calling the contract (here) and querying the contract (here) correct? Do you have any concerns about gas consumption logic related to the Wasm VM?

The gas registry is copied from x/wasm and the usage does appear to be correct. I recommend actually importing that code (not copying) for maintainability. However, the current gas pricing does seem to be correct, including the instantiation cost.

Wasm Storage and Error Handling

Can contracts write to the KV store passed to the VM?

Contracts have full read-write access to the *Prefixed KVStore* passed into the VM. On queries, they only have read access (unlike Cosmos SDK, where eg. the staking keeper writes during queries, in CosmWasm VM writes during queries are prevented both during compile time and error during runtime if someone compiles malicious code)

The contract-specific prefixing is set here

If any of the calls to Wasm VM return an error, is the contract's state reverted? Is there any additional handling we need to perform?

If the entire transaction fails, then all contract writes are reverted, as we are just writing all data to the same (Cached) KVStore that all the Go modules use and have the same semantics.

However, if you wish to revert state changes inside the contract, while the SDK message (and transaction) succeed and commit other state, then you need to implement this logic in your code. We provide this in wasmd for calling submessages. If you use wasmvm directly, you must implement the same logic in your code.

You can see how we CacheWrap the storage before dispatching a submessage, as well providing optional gas limits. This cache is only committed if no error is returned.

In practice, I believe any time a contract returns an error you do return a top-level error, aborting the transaction, and thus can safely rely on the normal Cosmos SDK semantics to discard any (uncommitted) state changes.

If the call to StoreCode returns an error, should we execute RemoveCode to delete the contract from disk storage before returning?

There is no need to do that, and we don't do that in wasmd either. The actual Rust code managing the Wasmer engine only writes the wasm code as well as the compiled version to disk when the whole process succeeds.

Contract Capabilities

Can a contract do protobuf marshaling? The reason to ask is that we were wondering if a contract would be able to proto-marshall the client state and consensus state and store it in the KV store. This is the expected encoding format for client state and consensus state proofs during connection handshake.

This is totally possible. Both marshaling and unmarshalling via prost. In fact, this is the main form that Osmosis uses to allow contracts to interact with the native modules, including the DEX. You can view an example contract forming a protobuf-based "StargateMsg". You can view the auto-generated Rust messages here, which uses prost in runtime. You can view the whole osmosis-std README for a better understanding. The bindings are compiled from the original **.proto* files via prost-build.

The payload that is passed to the contract is wrapped in a struct. For example, when querying the status of a light client you can see that there are 2 structs: payload and inner payload. In the contract the status query message is also wrapped in the Rust code. Would it be possible to get rid of this wrapping?

The particular format you refer to is how we represent enums in Go, which plays nicer with JSON serialization than the format protoc uses for oneof. The case you highlight is rather confusing because they only show one variant of the enum. The wrapping is needed as we use the top-level keyword of the map to identify the method (here "status"), and the inner struct to hold the arguments (an empty struct for no arguments).

Generally, one would define one struct in Go code with all possible variants and then set exactly one of them to be non-nil. Since you need some placeholder to differentiate between unset and empty, we use **struct{}*, so we get a serialized format like *{"status":{}}*. Most of the other cases would have actual arguments, so they would look like *{"validator":{"address":"cosmos1foobar"}}*.

The code developers can refer to the patterns used in the CosmWasm/wasmvm repo as best practices for handling nested JSON structs in communication with the contracts.

6. Implementation Review

1. Recommendation: It is unnecessary and wasteful to write the original wasm byte code to the IAVL store. The wasm is stored outside of the IAVL in *wasmvm* and you can query it back from the wasmVM if needed. As long as the hash is stored in the

chain, you are guaranteed that all nodes are running the same code.

2. Change Requested: As the Wasmvm.VM data is stored outside of the IAVL tree, you will need a Snapshotter extension to ensure that wasm dir is properly synced. This is implemented in wasmd already and must be registered in app.go. Since the IBC Wasm Light Client will share the same wasm cache dir as x/wasm, this must only be registered once and will work "out of the box" for existing CosmWasm chains, only required when adding the functionality to non-CosmWasm chains.

3. Recommendation: Many of the utils around code storing, gunziping it, etc are copied from x/wasm. And other code (like snapshotter, and gas registry) also needs to be copied or imported. It would be better to import such common logic, to ensure any upstream bug fixes or improvements are available. However, I also understand the desire to avoid a direct import of *wasmd* here.

   The goal should be only to make the minimal changes to wasmd as needed for the IBC integration and import / inherit as much functionality as possible. The CosmWasm maintainers monitor any security issues and do make patches when possible issues arise. It would be much safer to share this code, as it is unlikely the IBC Wasm Light Client maintainers have the resources to independently do such monitoring of CosmWasm integration issues.

   It would be good for the IBC Go maintainers to work closely with the Confio team to create a new go package with such shared utils that can be used by both projects without a full import of *wasmd* (eg. make a new *go.mod* in a subdirectory of the *wasmd* repo). Until that point, my suggestion is to make such a hard dependency and import those files from *wasmd*. Those imports should be isolated to one go package, so it is clearly defined what needs to be shared.

4. Change Required: The current version uses wasmvm 1.2.1, you should definitely bump it to the latest release 1.2.4 or higher, as there are some security issues patched there.

5. Recommendation: There is a way to pin code to the wasmvm cache. This makes a tradeoff to consume some memory in order for faster runtime and lower gas usage. This is typically hidden behind a governance proposal in x/wasm. However, since we have governance gating on registering the wasm clients, it would make sense to pin the code hash upon approval of the code to be a wasm light client. This is what is done when storing code with governance proposal in *x/wasm* as well.

6. Recommendation: Use Query for non-state changing operations. *Execute/Sudo* is intended for *state-changing operations*. This includes verifying a new block header, which will not only return success, but also update the internal consensusState. Functions that do not modify state should use the *Query* interface, which is better designed to return result data to the caller, and also provides a compile-time assurance that these methods only read state and will not modify it, which is what one would expect from eg. *VerifyMembership*. The actual code uses *call* not *query*,

which I would change, both in the IBC-Go client as well as the contracts.

7. Possible Improvement: Define one message type to encapsulate all variants for calls. Currently, there is a different payload wrapper for each variant. Like here for VerifyMembership and here for VerifyNonMembership. In our example wasmd code, we use one wrapping struct that includes all variants. Something like:

```
contractExecMsg struct {
  VerifyMembership *verifyMembershipPayload
`json:"verify_membership"`
  VerifyNonMembership *verifyNonMembershipPayload
`json:"verify_non_membership"`
}
```

8. Possible Improvement: Unify the RPC interfaces with x/wasm. Ideally using the same types when providing the same information to increase usability. *StoreCodeMsg* and the *Code* and *Codes* queries are all available on *x/wasm*, and tooling exists to use them, which is understood by contract developers. Ideally this would support the same proto types for the functionality that is also present in *x/wasm*, even if *x/wasm* is not there, making them look more unified in the interfaces.

9. Change Requested: Adjust the configuration options for the Wasmvm cache and make them more similar to *x/wasm,* especially allowing node-level configations. If architecture recommendation 2 is implemented, then these are the same Cache and should have reasonable configuration. *wasmDataDir*, *wasmSupportedFeatures*, and *wasmCacheSizeMb* should be passed in from *app.go*. *wasmCacheSizeMb* is not consensus-critical and should be defined on a per-node basis, often 100-1000MB. *wasmPrintDebug* must be false on all production nodes, and only enabled in test environments or debug non-validating nodes. *wasmMemoryLimitMb* defines the maximum memory allowed *per contract call*. This is hardcoded to 32 MB in the VM created by *x/wasm*, and could be raised to 128MB without much impact on recursion and such, but should not be raised significantly beyond that point. If there is a reason, it is likely a contract error, and the authors should discuss this more with the Confio core team.

7. Open Questions

There are several sections of the code that were outside the scope of the audit, but adjacent to it, and which may raise architecture issues, or may be correctly designed. As part of the response to this review, it is requested that the Interchain GmbH team provide explanations to the intended function of these sections and their usage in the larger system.

1. CheckSubstituteAndUpdateState seems to provide some sort of migration, but it is unclear where and why this is called. Please provide an explanation to see if this overlaps with standard *x/wasm* workflows. It also appears to be the only usage of wrappedStore, which is an odd data structure when we attempt to transparently

sandbox contracts.

2. Searching through *light-clients/08-wasm*, I only see NewClientState constructor used in test code. Since this is a key element to isolating different clients, code hashes, and client state, I would like to understand how this is used and constructed in the larger system.
   It seems like the IBC infrastructure detects the client id and provides a prefixed storage with *clients/{indentifier}/* prefix, which can be used to sandbox the state, as defined here. Then the serialized data under `ClientState` key maintains all the information that the 08-wasm handler needs to run the proper code.

3. It seems the when Go-level interface is called (like Status) it is passed read-write *clientStore sdk.KVStore* with this *clients/{indentifier}/* prefix as the only way to identify the currently running client; as well as the global *ctx sdk.Context*, which could allow access to any data. However, *sdk.Context* requires a StoreKey to access any substore, which is not available as it is called outside a keeper, and the logic may therefore not access any keeper data in these functions, just the local contract state. Is this a correct assumption as to the information available to the Go module when processing these calls?

4. How do you create a new client in the first place? Can a relayer pass any serialized ClientState bytes for the initial state? Or is there a place for the 08-wasm module to make any checks on those bytes during initialization, to eg. limit which code hashes may be used?

5. Where is the mapping between client type (eg 08-wasm) and the code executed for that type? Does this have to be updated at compile time? Or could it be registered dynamically at runtime based on the database state?

6. Why (either business reason or technical reason) was it decided the light-clients were permissioned not permissionless (eg. why the code hash must be approved by governance before a client can be instantiated with that code)?

7. What interfaces are available to detect the client data associated with a given client identifier? I assume someone off-chain could somehow detect a given identifier is run with *08-wasm* and then query the *clients/{identifier}/clientState* raw key and protobuf decode it. This would provide the *codeID* (*codeHash)* of the actual contract being run, which could be referenced in a registry or something. My question is what interfaces exist to make this process somewhat more ergonomic. I assume the 08-wasm keeper wouldn't have access to the ibc client state store to provide a custom query? Or how is this implemented in IBC Go.

8. Missing Features

These are new features that should be addressed in a future version. At the very least, all Change Requested and Recommendation issues should be properly closed before beginning work on these items.

TODO

Topics like safe isolation of contracts, (off-chain) client side interactions, as well as migrations require an understanding of some more design considerations covered in Open Questions. Once answers to those questions have been received in written form by Interchain GmbH, those answers will be included in this report, and the recommendation for these future features can be written on that basis.