The objective of this topic is to discuss how to properly abstract specific virtual machines such as Nock, Cairo, and RISC0 in the Anoma specification. At the moment, a virtual machine, which we will take here as simply a way to represent arbitrary classically-computable functions, is required in a few places:

- in the protocol where we need canonical commitments to and evaluatable representations of functions, such as external identities
- in the resource machine, for resource logics
- in the resource machine, for transaction functions
- in application definitions, for projection functions

In general, we do not need to use the same virtual machine in all of these places, or even a specific virtual machine for a specific place - we can support interoperability between multiple representations; we just need a way to demarcate what representation is in use so we can interpret data as the correct code.

In certain cases, however - such as external identities - we do need a way to understand when two representations are equal

, i.e. extensional function equality - and to compute a canonical succinct commitment upon the basis of which equality will be judged, we do need a canonical representation for the preimage. It may be possible to work with a weaker notion (commit a = commit b

entails a = b

, but a = b

does not necessarily entail commit a = commit b

), and use runtime equality judgements (a la “signs for” and “reads for”, which we need anyways), I’m not sure yet.

First, however, what exactly is

a virtual machine? Let’s try to capture a definition in math:

1. A virtual machine

is an encoding of arbitrary classically-computable functions defined by two (type-parameterized) functions: * deserialise : bytes -> Maybe (Repr T)

which attempts to deserialise a bytestring into an internal representation of a function of type T

.

- serialise : Repr T -> bytes

which serialises an internal representation of a function of a type T

into a bytestring.

- These functions must be inverses, i.e. deserialise . serialise = Just . id

. and fmap serialise . deserialise = id

.

1. deserialise : bytes -> Maybe (Repr T)

which attempts to deserialise a bytestring into an internal representation of a function of type T

.

1. serialise : Repr T -> bytes

which serialises an internal representation of a function of a type T

into a bytestring.

1. These functions must be inverses, i.e. deserialise . serialise = Just . id

. and fmap serialise . deserialise = id

.

1. An evaluatable virtual machine

is a virtual machine

with an additional function evaluate : (Repr (T0 -> T1)) -> Repr T0 -> Repr T1

, which simply calls a function on the provided argument (in an internal representation). evaluate

is required to be deterministic.

1. A provable virtual machine

is a virtual machine

with two additional functions parameterized over a proof type P

: * prove : Repr (T0 -> T1 -> T2 -> boolean) -> Repr T0 -> Repr T1 -> P

generates a proof, where Repr T0

is understood as the private input, and Repr T1

as the public input.

- verify : Repr (T0 -> T1 -> T2) -> Repr T1 -> P -> boolean

verifies a proof for a given program and public input.

- Should P

not reveal any information about Repr T0

, the provable virtual machine can be said to be zero-knowledge

.

1. prove : Repr (T0 -> T1 -> T2 -> boolean) -> Repr T0 -> Repr T1 -> P

generates a proof, where Repr T0

is understood as the private input, and Repr T1

as the public input.

1. verify : Repr (T0 -> T1 -> T2) -> Repr T1 -> P -> boolean

verifies a proof for a given program and public input.

1. Should P

not reveal any information about Repr T0

, the provable virtual machine can be said to be zero-knowledge

.

I think these definitions should be (roughly) sufficient for our purposes. They do not answer, for now, the question of how to convert

between representations in a semantics-preserving way (i.e., compilation), which may be useful to introduce into the protocol structure in the future, but I think we do not need it yet.

Following this route, in the specs we would then define how each of Nock, Cairo, and RISC0 instantiate these types. Before doing that, I wanted to write this up and seek input from @vveiln @jonathan @degregat @mariari @Moonchild @ray .