The aim of this post is to describe a "minimum viable abstract VM" which does everything that we require. This could also be understood as an abstract interface specification for Taiga.

Basics

Fix a finite field F

, with:

- Multiplication: *_F

, with type F \to F \to F

- Addition: +_F

, with type F \to F \to F

- Additive identity: 0_F

, with type F

- Multiplicative identity: 1_F

, with type F

- Additive inverse: -_F

, with type F \to F

- Equality: =_F

, with type F \to F \to {0 | 1}

Fix an abstract (possibly infinite) program field P

, with:

- Multiplication: *_P

, with type P \to P \to P

- Addition: +_P

, with type P \to P \to P

- Additive identity: 0_P

, with type P

- Multiplicative identity: 1_P

, with type P

- Additive inverse: -_P

, with type P \to P

- Comparison: <_P

(less than, exclusive), with type P \to P \to {0 | 1}

Fix an abstract proving system

with two functions:

- Prove(C, x, w) ⟶ π
- Verify(C, x, π) ⟶ { 0 | 1}

We say that this proving system is zero-knowledge

if from π

the verifier learns nothing of w

.

(we will omit preprocessing for the sake of simplicity)

Fix a hash function

H

, with type $P \to F$

, such that:

- H

is non-invertible, in that a computationally bounded adversary cannot compute the preimage from the output

- H

is collision-resistant, in that a computationally bounded adversary cannot feasibly find two preimages with the same output

Programs

Now, we will define programs.

Programs PROG

are extended multivariate polynomials over $a_0 ... a_n \in P$

in canonical form, with designated variable b

, for example:

$a_0^3 + 2a_1^2 - 3a_0 + 2 - b = 0$

We extend these polynomials with control flow and reflection.

First, let’s introduce the control flow primitives: comparison and branching.

Comparison

is $lt(a, b)$

, where:

- $lt(a, b) = 0$

if $a <_P b$

- $lt(a, b) = 1$

otherwise

Branching

is $branch(a, b, c)$

, with type $P \to P \to P \to P$

, where:

- $branch(0_P, b, \_) = b$
- $branch(1_P, \_, c) = c$
- $branch(\_, \_, \_) = undefined$

Thus, for example,

$branch((lt(a_0, a_1), a_2, a_3^2) - b = 0$

is a valid program.

Assuming that $a \in \{0, 1\}$

, branch(a, b, c)

is semantically equivalent to ((1 - a)*_P b) + (a*_P c)

, so we aren't changing the expressive power here, just adding explicit information about control flow.

Second, let's introduce reflection:

- reflect_P

, with type P \to PROG

- repr_P

, with type PROG \to P

- eval_P

, with type PROG \to P {...} \to P

, where the arity depends on PROG

Argument order corresponds to numbering of a_0 ... a_n

.

Thus, for example,

eval_P(branch((lt(a_0, a_1), a_2, a_3^2) - a_4 = 0, 3, 4, 5, b) = 0

is a valid program, and so is

eval_P(reflect_P(a_1), 3) + 2a_2 - b = 0

The behavior of applying eval_P

with the wrong number of arguments is undefined.

Note, in particular, that we can freely swap a subprogram PROG

with eval_P(reflect_P(repr_P(PROG)))

, where repr_P(PROG)

can be calculated at compile-time, and thus "defer compilation".

Circuit transformation

Programs may be transformed to a circuit, by:

- factoring the polynomial
- ordering the factors in canonical form
- transforming to a circuit DAG

Stack machine compilation

The circuit DAG may then be compiled to a sequential stack machine in the standard manner. For example, define the stack machine state as (pc, ins, data)

, with:

- pc

the program counter

- ins

the instruction sequence

- data

the data stack

Stack machine operations:

- PUSH a
- NEG
- ADD
- MUL
- COMPARE
- BRANCH pc_1 pc_2
- REPR
- REFLECT
- EVAL

TODO: Figure out any stack consistency issues on branching.

Resources

Define a resource

R

as a seven-tuple (PROG, P, F, P, F, F, 0 | 1)

with fields named as follows:

- R_{logic}

of type PROG

- R_{label}

of type P

- R_{quantity}

of type F

- R_{value}

of type P

- R_{nonce}

of type F

- R_{nc}

of type F

(“nullifier commitment”)

- R_{ephemerality}

of type { 0_P | 1_P }

Resources with R_{ephemerality} = 0_P

are known as ephemeral

, while resources with R_{ephemerality} = 1_P

are known as persistent

.

TODO: Same as the commitment for now.

Define the commitment

of a resource $R_{commitment}$

as $hash(R)$

.

Define the address

of a resource $R_{address}$

as $R_{commitment}$

.

Define the nullifier

of a resource $R_{nullifier}$

as n

such that $hash(n) = R_{nc}$

.

Define the type

of a resource R_{type}

as $hash(R_{logic}, R_{label})$

.

Define the delta

of a resource R_{delta}

as the two-tuple $(R_{type}, R_{quantity})$

.

Partial transactions

Define a partial transaction

PTX

as a six-tuple $(Set\ F, Set\ F, Set\ \pi, D, Set\ P, PROG)$

with fields named as follows:

- $PTX_{created}$

of type $Set\ F$

(commitments to newly created resources)

- $PTX_{consumed}$

of type $Set\ F$

(nullifiers for consumed resources)

- PTX_{proofs}

of type $Set\ \pi$

(proofs for consumed and created resources)

- PTX_{delta}

of type $Set\ (F, F)$

(as resource delta above)

- PTX_{extradata}

of type Set\ P

(extradata possibly used as input by proofs)

- PTX_{executable}

of type PROG

(possibly empty)

A partial transaction is valid

w.r.t. a past commitment set CS

if and only if:

- For each commitment c

in PTX_{created}

, a valid proof π

is included in PTX_{proofs}

such that, for some resource R

with R_{commitment} = c

, * Verify(R_{logic}, (in, out, extradata, 1), π) = 1

, with in \subset PTX_{consumed}

, out \subset PTX_{created}

, extradata \subset PTX_{extradata}

.

- Verify(R_{logic}, (in, out, extradata, 1), π) = 1

, with in \subset PTX_{consumed}

, out \subset PTX_{created}

, extradata \subset PTX_{extradata}

.

- For each nullifier n

in PTX_{consumed}

, a valid proof π

is included in PTX_{proofs}

such that, for some resource R

with R_{nullifier} = n

and R_{commitment} \in CS

. * Verify(R_{logic}, (in, out, extradata, 0), π) = 1

, with in \subset PTX_{consumed}

, out \subset PTX_{created}

, extradata \subset PTX_{extradata}

.

- $\text{Verify}(R_{logic}, (in, out, extradata, 0), \pi) = 1$

, with $in \subset PTX_{consumed}$

, $out \subset PTX_{created}$

, $extradata \subset PTX_{extradata}$

.

- $PTX_{delta} = \sum_{R \in PTX_{consumed}} R_{delta} - \sum_{R \in PTX_{created}} R_{delta}$

A partial transaction is balanced

if and only if $PTX_{delta} = 0$

.

A partial transaction may be executed

by running PROG

, which may additionally:

- read resources by $R_{address}$
- read from and add to $PTX_{extradata}$
- add to $PTX_{created}$

(altering PTX_{delta}

)

- add to $PTX_{consumed}$

(altering PTX_{delta}

)

Partial transactions may be composed

either sequentially

or concurrently

, where sequential

composition uses the output commitment set as input to the next transaction, and concurrent

composition does not. In both cases, nullifier sets may not conflict.

Recursion

Abstractly, recursion may be achieved as follows: define a resource R

with:

- R_{label}

set to the consensus provider identity

- $R_{quantity}$

set to 1

- R_{value}

set to the commitment and nullifier trees

- R_{nonce}

an incrementing counter

- R_{nc} = hash(0_F)
- R_{ephemerality} = 1
- R_{logic} = verify_partial_tx\ \&\&\ verify_signature

where the partial tx is checked with respect to the input and output commitment and nullifier sets in R_{value}

and the signature is checked with respect to the consensus provider identity in R_{label}

.