Public VM

Objectives

We need to ensure that the execution environment for public functions has the following properties:

1. The Prover is fairly compensated for their work (gas metering)
2. A dishonest Prover cannot convince the kernel circuit that a public function failed, when an honest Prover would claim it succeeded
3. A public function cannot produce unsatisfiable constraints that would prevent the creation of a valid block proof
4. The published verification key for a public function is accurately represented by the opcode stream Provers will use to construct a function proof

Why we need a VM

Point 2 requires a Virtual Machine architecture. The older model of having ACIR directly produce a circuit + verification key is flawed. For any witness that isnot a public input, a dishonest Prover could assign an incorrect value and create a failing proof.

The kernel circuit cannot prevent this byrequiring the public function proof to succeed, as it is trivial for a malicious contract writer to create unsatisfiable programs (e.g.assert(true == false)).

We cannot require that a Prover/Sequencer simulates the transaction to determine whether it is possible to include it in the block proof, as this amounts to unpaid compute that can be used as a DoS attack vector.

VM Solution

How a VM solves DoS attacks + dishonest Prover/contract writer attacks.

Every VM opcode operates on someinput state and executes 1 of 2 functions:

1. A pure function that produces output state from input state
2. A function that produces no output state and applies assertion checks on input state

VM simulator asserts that functions of type 1 always succeed.

VM simulator allows functions of type 2 to fail while still producing a satisfiable VM proof. (i.e. the VM circuit contains afailed public input flag, that will be set totrue if any assertion checks fail.)

The protocol mandates the VM proof must be valid, ensuring that functions of type 1 are correctly evaluated by the Prover.

The VM architecture must finally ensure thatall witness values are either public inputs or derived from public inputs. This implies that failing assertion checks are due to bad inputs and not a malicious Prover.

Gas metering the VM

Public functions have the following L2 costs associated with them:

1. The compute cost of generating a VM proof for a given program
2. The compute cost of generating a public kernel snark proof
3. The latency cost of reading L2 state
4. The storage cost of writing L2 state

The following L1 costs also apply:

1. Data cost of broadcasting L2 state writes to L1

Under a VM model, the VM simulator can perform opcode-by-opcode gas metering.

(need to define VM architecture for more detail)

VM ABI

Like private functions, public functions do not make state updates; they make state updaterequests that are kicked out as public inputs.

This ensures that all VM witnesses are derived from public inputs (Merkle hash paths have lots of aux data not contained in public inputs)

It also makes gas metering easier as all L1 metering happens in the Kernel circuit, not the VM simulator circuit

MVP Roadmap

The public kernel circuit does not directly verify a public function proof.

It verifies averifier circuit that verifies a public function proof!

Why? Modularity, ease of development, backwards compatibility support.

Proceed with the following development phases:

Phase 0: Full Proverless

There are no proofs. The rollup verifier smart contract returnsyes

Phase 1: Kernel Proverless

Rollup proofs are generated, but public kernel snark verification logic is absent: kernel verifier returnsyes

Phase 2: VM Proverless

Kernel proof is generated, but public function verifier circuit always returnsyes

Phase 3: Honest prover assumption

The verifier circuit verifies a circuit whose verification key is directly computed from ACIR (like the private functions)

In theory we could go into production with this model, but without full decentralization.

Phase 4: Full VM

The verifier circuit verifies that a VM program has been successfully executed.

Additional benefit of having a VM verifier circuit is that we can upgrade the VM while supporting backwards compatibility (Kernel circuit picks one of multiple VM verification keys depending on the version used when contract was deployed).

Also means we can adopt completely different VM architectures without changing existing circuits (e.g. eWasm?)

Draft Aztec VM Architecture

This VM arch design makes the following assumptions:

1. We want the public function developer experience to track private function developer experience as closely as possible
2. Public function efficiency is not a protocol bottleneck (i.e. if pub fns are similar to private fns, and private fns proofs can be computed on consumer hardware, we can afford a significant prover slowdown factor when making public fn proofs).

On the tradeoff space between public function prover efficiency and reduced protocol complexity, I think it's better to bias towards reduced protocol complexity.

Given that private functions compile to ACIR opcodes, it is natural to define a VM architecture that directlysimulates ACIR opcodes.

Consider a crypto backend that produces circuit constraints from ACIR opcodes. Minimal complexity solution requires a VM architecture that will executeidentical constraints within the context of a VM simulator.

i.e. we can repurpose our existing crypto backend and ACIR constraint generation modules to build the VM.

VM architecture overview

ACIR opcodes do not directly map to constraints; they map to subroutines composed of constraints (however the subroutine size can be 1 for a simple opcode e.g. ADD)

The subroutine constraints for a given opcode is described bymicrocode .

Aprogram instruction is a tuple of:

1. instruction value
2. instruction witness context
3. instruction selector context

Theinstruction value is represents a combination of opcode value and microcode value.

Thewitness context describes up to fourwitness indices that are used to look up the witness values used in the microcode constraint.

Theselector context describes custom arithmetic selector values, when the values are not derived directly from the opcode/microcode (e.g. mul gates, add gates, linear combination gates)

An instruction value is used to look up the UltraPlonk constraint selectors (from a fixed lookup table that is shared by all VM programs) required to evaluate a microcode constraint.

The microcode constraint selector values are looked up from opcode/microcode instruction table. For arithmetic selectors, the program-specific selector context values are added in.

We assume that if opcode selector context is nonzero, relevant selector values in the microcode lookup table are zero. Vice-versa applies. This can be validated when publishing a contract. Selector contexts cannot fully define an arithmetic constraint. There must be 1 degree of freedom remaining to prevent VM programs from producing unsatisfiable constraints (e.g.q m , q 1 , q 2 , q 3 , q 4 = 0 , q c = 1 q_m, q_1, q_2, q_3, q_4 = 0, q_c = 1 q m ,

q 1 ,

q 2 ,

q 3 ,

q 4

= 0 ,

q c

= 1). TODO: how do we enforce this? We might need to transpile some ACIR opcodes that run into this issue into sequences of simpler ones that don't.

VM program columns

Witness commitments:

P C PC PC I I I C q⃗ \vec{C_q} C q C⃗ w \vec{C}_{w} C w Q⃗ \vec{Q} Q W⃗ \vec{W} W E E E G G G Program counter Instruction value selector context witness indices (4 columns) selector values witness values (4 columns) Error flag Gas counter Prover-computed lookup tables:

T⃗ W \vec{T}*{W} T W Lookup Relation Maps witness index to witness value (4 columns) W i = T W i [C W i] W_i =*{T}W_i}[C_{W_i}] W i

= T W i [C W i] Program-specific precomputed lookup tables:

T⃗ I \vec{T}*I T I Lookup Relation Maps program counter value to tuple ofinstruction value, witness context, selector context { I , C⃗ q , C⃗ w } = T I [P C] {I, \vec{C}_q, \vec{C}_w } = T*{I}[PC] { I ,

C q ,

C w }

= T I [PC] The[T⃗ I] [\vec{T}_I] [T I] commitments form the program-specific components of the VM verification key.

VM-specific precomputed lookup tables

T⃗ Q \vec{T}_{Q} T Q Lookup Relation Maps instruction value to selector values { q 1 , q 2 , q 3 , q 4 , q m , q c , q s o r t ,

q p l o o k u p , q p l o o k u p _ i n d e x , q a r i t h , q e c c , q a u x }

= T Q [I] { q_1, q_2, q_3, q_4, q_m, q_c, q_{sort}, \ q_{plookup}, q_{plookup_index}, q_{arith}, q_{ecc}, q_{aux} } \ = T_Q[I] { q 1 ,

q 2 ,

q 3 ,

q 4 ,

q m ,

q c ,

q sor t ,

q pl oo k u p ,

q pl oo k u p _ in d e x ,

q a r i t h ,

q ecc ,

q a ux }

= T Q [I] T G T_{G} T G Lookup Relation Defines gas value per instruction G i = G i − 1 + T G [I i] G_{i} = G_{i - 1} + T_G[I_i] G i

= G i − 1

+ T G [I i]

VM Verifier Logic

New arithmetic checks are required in the verifier algorithm.

1. P
2. C
3. i
4. =
5. P
6. C
7. i
8. −
9. 1
10. +
11. 1
12. PC_i = PC_{i-1} + 1
13. P
14. C
15. i
16.
17. =
18. P
19. C
20. i
21. −
22. 1
23.
24. +
25. 1
26. (if we add JUMP/JUMPI instructions this logic will change)
27. {
28. I
29. ,
30. C
31. →
32. q
33. ,
34. C
35. →
36. w
37. }
38. =

39. T
40. i
41. [
42. P
43. C
44.]
45. { I, \vec{C}_q, \vec{C}_w } = T_i[PC]
46. {
47. I
48. ,
49. C
50. q
51.
52. ,
53. C
54. w
55.
56. }
57. =
58. T
59. i
60.
61. [
62. PC
63.]
64. W
65. →
66. =
67. T
68. w
69. [
70. C
71. →
72. w
73.]
74. \vec{W} = T_w[\vec{C}_w]
75. W
76. =
77. T
78. w
79.
80. [
81. C
82. w
83.
84.]
85. Q
86. →
87. =
88. T
89. q
90. [
91. I
92.]
93. +
94. C
95. →
96. q
97. \vec{Q} = T_q[I] + \vec{C}_q
98. Q
99.
100. =
101. T
102. q
103.
104. [
105. I
106.]

107. +
108. C
109. q
110.
111. G
112. i
113. =
114. G
115. i
116. −
117. 1
118. +
119. T
120. G
121. [
122. I
123. i
124.]
125. G_i = G_{i-1} + T_G[I_i]
126. G
127. i
128.
129. =
130. G
131. i
132. −
133. 1
134.
135. +
136. T
137. G
138.
139. [
140. I
141. i
142.
143.]

Standard UltraPlonk arithmetic checks are applied toW⃗ \vec{W} W andQ⃗ \vec{Q} Q , with some changes to the standard arithmetic constraint to accommodate for the error flagE E E

e.g. For some assertion checkP (W , Q) P(W, Q) P (W ,

Q) , we do not requireP (W , Q) = = 0 P(W, Q) == 0 P (W ,

Q)

== 0 but instead:

- CheckE
- E
- E
- is boolean
- IfP
- =
- =
- 0
- ,
- E
- =
- =
- 0
- P == 0, E == 0
- P
- ==
- 0
- ,
- E
- ==

- 0
- IfP
- ≠
- 0
- ,
- E
- =
- =
- 1
- P \ne 0, E == 1
- P
- ❓
- =
- 0
- ,
- E
- ==
- 1

i.e:

IfP ≠ 0 P \ne 0 P

❓

0 letX = 1 P X = \frac{1}{P} X

= P

1 , elseX = = 0 X == 0 X

== 0

- P
- ⋅
- (
- 1
- −
- E
-)
- =
- =
- 0
- P \cdot (1 - E) == 0
- P
- ⋅
- (
- 1
- −
- E
-)
- ==
- 0
- (
- P
- ⋅
- X
- −
- 1
-)
- ⋅
- E
- =
- =
- 0
- (P \cdot X - 1) \cdot E == 0
- (
- P

- ⋅
- X
- −
- 1
-)
- ⋅
- E
- ==
- 0
- E
- 2
- −
- E
- =
- =
- 0
- E^2 - E == 0
- E
- 2
- −
- E
- ==
- 0

Constant selectors

In the VM model, what used to be precomputed selectors are now part of the Prover's witness commitment.

The only selectors where this does not apply are:

1. Existing plookup selectors
2. Permutation selectors

Permutation Arguments

The original Plonk permutation argument is no longer required as all witnesses are looked up via plookup tables.

Additional grand product arguments are required for the multiple new lookups

Set equivalence checks

Current Composers rely on set equivalence checks to perform range checks, RAM read/writes and ROM reads.

This isn't possible in the VM model as set equivalence checks require additional constraints that depend on the size of the set. i.e. if two programs require different numbers of range checks (or different range sizes) they will not produce two uniform circuits.

In the VM arch we need to find another way to evaluate range checks, ROM and RAM. Will leave as a TODO for now, doubt it's too tricky. Range checks also fall under the assertion category i.e. failing a range check doesn't create an unsatisfiable circuit; the proof is valid but the output error flag is set to1 1 1 .

VM Overheads

Why this is a public-function-only abstraction.

The following UltraPlonk/Honk selectors are now witness commitments! (TheQ⃗ \vec{Q} Q columns)

q 1 , q 2 , q 3 , q 4 , q m , q c , q s o r t , q p l o o k u p , q p l o o k u p _ i n d e x , q a r i t h , q e c c , q a u x q_1, q_2, q_3, q_4, q_m, q_c, q_{sort}, q_{plookup}, q_{plookup_index}, q_{arith}, q_{ecc}, q_{aux} q 1 ,

q 2 ,

q 3 ,

q 4 ,

q m ,

q c ,

q sor t ,

q pl oo k u p ,

q pl oo k u p _ in d e x ,

q a r i t h ,

q ecc ,

q a ux

Additional witness commitments required that are not part of UltraPlonk/Honk:

P C , I , C w 1 , C w 2 , C w 3 , C w 4 , E , G , T w 1 , T w 2 , T w 3 , T w 4 PC, I, C_{w1}, C_{w2}, C_{w3}, C_{w4}, E, G, T_{w1}, T_{w2}, T_{w3}, T_{w4} PC ,

I ,

C w 1 ,

C w 2 ,

C w 3 ,

C w 4 ,

E ,

G ,

T w 1 ,

T w 2 ,

T w 3 ,

T w 4

In addition, selector context polynomialsC q C_{q} C q will likely represent at least 4 selectors in order to handle ACIR's linear combination instruction

That's 28 additional witness polynomial commitments on top of the usual Honk commitments.

The additional 4 plookup arguments will also likely require at least 2 additional grand product + sorted list commitments, pushing the number of extra commitments to 32.

To contrast, we are expecting Honk to have 6 Prover commitments:

w 1 , w 2 , w 3 , w 4 , Z , S w1, w2, w3, w4, Z, S w 1 ,

w 2 ,

w 3 ,

w 4 ,

Z ,

S

(Z = combined grand product of permutation argument and plookup argument) (S = sorted list for plookup argument)

Plus 2 opening proofs for KZG.

i.e. the overall number of Prover multi-exponentiations has grown from8 n 8n 8 n to40 n 40n 40 n

TLDR: a VM simulator circuit of an ACIR program is approx. 6 times more expensive to prove than a proof of a natively-compiled ACIR circuit.

We can afford to pay this for public functions, but this is far too great an overhead for locally-generated private function proofs.

Building the VM simulator circuit

The plan is that the existing crypto backend can do the following by re-using the Plonk standard library with a custom composer:

1. build the VM circuit lookup tables
2. build VM verification keys for a given programand
3. generate program-specific VM proofs

i.e. stdlib code is re-used and only composer code changes.

ComputingT

Q , T G T_Q, T_G T Q ,

T G using the stdlib

A program is written that executes every ACIR opcode in sequence.

Composer will track an 'instruction counter'. If the opcode makes a standard library call, the stdlib function is executed as normal, but the Composer will do the following:

- Whenever a constraint is added, selector values are written intoT
- Q
- T_Q
- T
- Q
-
- at the currentinstruction_counter
- value
- Default value written intoT
- G
- T_G
- T
- G
-
- . We'll need to pass special context booleans into a stdlib function if it is performing a state read/write in order to add special values intoT
- G
- T_G
- T
- G
-
- instruction_counter
- is incremented

Computing VM circuits + proofs using the stdlib

In this context, the Composer already possessesT Q T_Q T Q .

Consider the execution of an ACIR opcode stream. If the opcode makes a standard library call, the stdlib function is executed as normal, but the Composer will do the following when a constraint is added:

- Current program counter value added intoP
- C
- PC
- PC
- Composer validates the selector values are present inT
- Q
- T_Q
- T
- Q
-
- and looks up the instruction value
- instruction value added toI
- I
- I
- If not already present, program counter/instruction value mapping is written intoT
- I

- T_I
- T
- I
-
- , along with required witness indices
- Witness index values are written intoC
- w
- C_w
- C
- w
-
- Depending on context, some constraints may write values intoC
- q
- C_q
- C
- q
-
- (e.g.basic arithmetic gates)
- Gas values looked up fromT
- G
- T_G
- T
- G
-
- andG
- G
- G
- updated
- SimilarlyE
- E
- E
- updated if assertion fails

We assume the circuit will evaluate an opcode stream of a fixed size. If the opcode stream is smaller than this limit, the remaining instructions are evaluated as NOP instructions

The Composer will be able to produce the program-specific components of the verification key[$\vec{T}$ I] [\vec{T}_I] [T I] as well as the Prover-specific commitments for a proof given the public inputs used.

Validating VM programs

When adding a contract to Aztec, we want to validate that theT I T_I T I commitments are commitments to a published opcode stream.

We can do this efficiently in a custom circuit.

1. Compute Fiat-Shamir challengesα
2. ,
3. ζ
4. \alpha, \zeta
5. α
6. ,
7. ζ
8. by hashing[
9. T
10. →
11. I
12.]
13. [\vec{T}_I]
14. [
15. T
16. I
17.
18.]
19. EvaluateT
20. →
21. I
22. (
23. ζ

24.)
25. \vec{T}_I(\zeta)
26. T
27. I
28.
29. (
30. ζ
31.)
32. via KZG, usingα
33. \alpha
34. α
35. to create linear combination of all commitments within[
36. T
37. →
38. I
39.]
40. [\vec{T}_I]
41. [
42. T
43. I
44.
45.]
46. Iterate over the opcode stream and manually computeT
47. →
48. I
49. (
50. ζ
51.)
52. \vec{T}_I(\zeta)
53. T
54. I
55.
56. (
57. ζ
58.)
59. in linear-time using field operations. Validate this matches the evaluation of the commitments

In addition, we can perform checks on each opcode to validate conditions on selector contexts withinT I T_I T I and ensure unsatisfiable constraints are not being generated.

Open Questions

No doubt many, but first one on my mind is:

Q: Do we want to support JUMP/JUMPI instructions for public ACIR functions? Would bevery nice to have but creates discontinuity between public/private functions. Edit this page

Previous State Model Next Trees