Contents

AMO-Lean: Ruta de Trabajo hacia zkVM

Documento de Planificacion Tecnica

Version: 1.0 **Fecha**: Enero 2026 **Autores**: Equipo AMO-Lean

---

Resumen Ejecutivo

Este documento presenta la planificacion detallada para evolucionar AMO-Lean desde su estado actual (compilador FRI verificado) hacia una herramienta capaz de generar primitivos criptograficos optimizados para zkVMs de produccion.

Priorizacion basada en impacto para zkVM:

Prioridad	Componente	Tiempo	Impacto
#1 CRITICO	Poseidon/Rescue Hash	6-10 semanas	Habilita recursion eficiente
#2 ALTO	Backend CUDA/GPU	3-6 meses	10-100x speedup en proof generation
#3 MEDIO	Variantes AVX-512	2-3 semanas	2x speedup incremental
#4 BAJO	FRI Query Phase	4-6 semanas	Completa el protocolo

---

1. Poseidon/Rescue Hash Integration

1.1 Contexto y Motivacion

Poseidon es un hash "ZK-friendly" disenado para ser eficiente dentro de circuitos aritmeticos. Es el estandar de facto para: - Recursion de pruebas (una prueba verifica otra) - Arboles de Merkle en zkVMs - Compromisos de estado

Por que es critico: Sin Poseidon optimizado, la recursion de pruebas es prohibitivamente lenta, limitando la escalabilidad de cualquier zkVM.

1.2 Desafio Tecnico Principal

Poseidon combina operaciones lineales y no-lineales:

`Poseidon_round(state) = MDS x (state + round_constants)^alpha`

Donde: - `MDS`: Matriz de difusion (Maximum Distance Separable) – **LINEAL**, compatible con Kronecker - `^alpha`: S-box, tipicamente x^5 – **NO LINEAL**, rompe nuestra representacion actual

El problema: Nuestra arquitectura `MatExpr` esta optimizada para operaciones lineales (productos Kronecker). Las S-boxes requieren extender el IR.

1.3 Plan de Implementacion por Fases

Fase 1.1: Extension de MatExpr (Semanas 1-2) **Objetivo**: Anadir soporte para operaciones element-wise no-lineales.

Entregables:

```
-- Nuevo constructor en MatExpr
inductive MatExpr (a : Type) : Nat -> Nat -> Type where
  | ...existing constructors...
  | elemwise : (Expr a -> Expr a) -> MatExpr a n n
  | diagMap : (Expr a -> Expr a) -> VecExpr a n -> MatExpr a n n
```

Obstaculos tecnicos: 1. **Preservacion de tipos dependientes**: El constructor `elemwise` debe preservar dimensiones - *Tecnica*: Usar indices fantasma (phantom indices) como en Xi & Pfenning - *Bibliografia*: "Dependent Types in Practical Programming" (POPL 1999)

2. **Interaccion con E-Graph**: Las operaciones no-lineales complican el e-matching
   - *Tecnica*: Tratar `elemwise` como nodo opaco, sin expansion
   - *Bibliografia*: Willsey et al. "egg: Fast and Extensible Equality Saturation"

Testing Fase 1.1: - [ ] Unit tests: `elemwise` preserva dimensiones - [ ] Property test: `elemwise id = identity` - [ ] Integration test: Composicion con operaciones lineales

Fase 1.2: Implementacion de Poseidon Round (Semanas 3-4) **Objetivo**: Implementar una ronda de Poseidon usando el nuevo IR.

Entregables:

```
/-- Constantes de Poseidon para estado de tamano t -/
structure PoseidonParams (t : Nat) where
  mds : Matrix t t           -- Matriz MDS
  roundConstants : Array (Vec t)  -- Constantes por ronda
  fullRounds : Nat           -- Rondas completas
  partialRounds : Nat        -- Rondas parciales
  alpha : Nat := 5           -- Exponente S-box

/-- Una ronda completa de Poseidon -/
```

```
def poseidonFullRound (params : PoseidonParams t) (round : Nat)
    (state : MatExpr a t 1) : MatExpr a t 1 :=
  let withConstants := MatExpr.add state (params.roundConstants.get! round)
  let afterSbox := MatExpr.elemwise (fun x => Expr.pow x params.alpha) withConstants
  MatExpr.matmul params.mds afterSbox
```

Obstaculos tecnicos: 1. **Matrices MDS grandes**: Para t=12 (comun en zkVMs), la matriz tiene 144 elementos - *Tecnica*: Representacion sparse si hay estructura, o precomputacion - *Bibliografia*: Grassi et al. "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems"

2. **Overflow en campos finitos**: x^5 puede causar overflow
   - *Tecnica*: Usar representacion Montgomery para aritmetica modular
   - *Bibliografia*: Montgomery "Modular Multiplication Without Trial Division"

Testing Fase 1.2: - [ ] Test vectors oficiales de Poseidon - [ ] Comparacion con implementacion de referencia (Circom) - [ ] Fuzzing diferencial contra poseidon-rs

Fase 1.3: Optimizacion de S-boxes (Semanas 5-6) **Objetivo**: Optimizar x^5 para diferentes backends.

Entregables:

```
/-- Estrategias de exponenciacion -/
inductive SboxStrategy where
  | naive      : SboxStrategy  -- x * x * x * x * x
  | squareChain : SboxStrategy  -- x^2 -> x^4 -> x^5 (3 muls)
  | lookupTable : Nat -> SboxStrategy  -- Tabla precalculada (para campos pequenos)

/-- Seleccionar estrategia optima -/
def selectSboxStrategy (fieldSize : Nat) (simdWidth : Nat) : SboxStrategy :=
  if fieldSize <= 2^16 then SboxStrategy.lookupTable fieldSize
  else SboxStrategy.squareChain
```

Obstaculos tecnicos: 1. **Trade-off memoria vs computacion**: Lookup tables usan memoria pero son O(1) - *Tecnica*: Cost model que considera cache sizes - *Bibliografia*: Agner Fog "Instruction Tables" (latencias reales de CPU)

2. **SIMD para exponenciacion**: AVX2 no tiene instruccion de potencia
   - *Tecnica*: Vectorizar la cadena de cuadrados
   - *Bibliografia*: Intel Intrinsics Guide

Testing Fase 1.3: - [ ] Benchmark: naive vs squareChain vs lookupTable - [ ] Verificar equivalencia numerica entre estrategias - [ ] Profile con perf/vtune para cache misses

Fase 1.4: Full Poseidon + CodeGen (Semanas 7-9) **Objetivo**: Poseidon completo con generacion de codigo C optimizado.

Entregables: - `poseidonHash : PoseidonParams -> Array FieldElement -> FieldElement` - `generatePoseidonC : PoseidonParams -> String` (codigo C) - Proof anchors para cada componente

Obstaculos tecnicos: 1. **Rondas parciales**: Poseidon usa rondas parciales (solo una S-box) para eficiencia - *Tecnica*: Modelar como caso especial de `elemwise` - *Bibliografia*: Poseidon paper, Appendix B

2. **Padding y dominio de separacion**: Hash debe manejar inputs de longitud variable
   - *Tecnica*: Sponge construction estandar
   - *Bibliografia*: NIST SP 800-185 (SHA-3 derived functions)

Testing Fase 1.4: - [ ] Full test suite contra implementaciones canonicas - [ ] Fuzzing con inputs aleatorios de varias longitudes - [ ] Verificar proof anchors contra especificacion

Fase 1.5: Verificacion Formal (Semana 10) **Objetivo**: Teoremas que conectan implementacion con especificacion.

Entregables:

```
theorem poseidon_permutation_correct (params : PoseidonParams t) (input : Vec t) :
  evalPoseidon params input = reference_poseidon params input

theorem poseidon_sponge_absorb (params : PoseidonParams t) (msg : Array FieldElement) :
  spongeAbsorb params msg satisfies_sponge_security
```

Obstaculos tecnicos: 1. **Propiedades criptograficas dificiles de probar**: Resistencia a colisiones requiere asunciones - *Tecnica*: Probar propiedades estructurales, no criptograficas - *Bibliografia*: Barthe et al. "Computer-Aided Cryptographic Proofs"

Testing Fase 1.5: - [ ] Teoremas compilan sin sorry (o con sorry documentados) - [ ] Differential fuzzing: Lean eval == C binary

---

2. Backend CUDA/GPU

2.1 Contexto y Motivacion

La industria de ZK se esta moviendo hacia proof generation en GPUs: - Granjas de GPUs como servicio (proof marketplaces) - Paralelismo masivo: miles de hilos vs decenas en CPU - Potencial de 10-100x speedup

2.2 Desafio Tecnico Principal

Las GPUs tienen una jerarquia de memoria diferente:

```
CPU: RAM <-> L3 <-> L2 <-> L1 <-> Registros

GPU: Global Memory (lenta, grande)
         |
     Shared Memory (rapida, pequena, por bloque)
         |
     Registers (muy rapida, por hilo)
```

Nuestro IR actual no modela movimiento de datos entre niveles de memoria.

2.3 Plan de Implementacion por Fases

Fase 2.1: Diseno de GPUExpr IR (Semanas 1-3) **Objetivo**: Nuevo IR que modela jerarquia de memoria GPU.

Entregables:

```
/-- Niveles de memoria GPU -/
inductive MemLevel where
  | global   : MemLevel  -- DDR, alta latencia
  | shared   : MemLevel  -- Por bloque, baja latencia
  | register : MemLevel  -- Por hilo

/-- Expresiones GPU con anotaciones de memoria -/
inductive GPUExpr where
  | load   : MemLevel -> Address -> GPUExpr
  | store  : MemLevel -> Address -> GPUExpr -> GPUExpr
  | compute : Kernel -> List GPUExpr -> GPUExpr
```

```
  | sync   : GPUExpr  -- __syncthreads()
  | parallel : Nat -> Nat -> GPUExpr -> GPUExpr  -- gridDim, blockDim
```

Obstaculos tecnicos: 1. **Coalesced memory access**: Accesos a global memory deben ser coalescentes - *Tecnica*: Analisis de patrones de acceso en tiempo de compilacion - *Bibliografia*: CUDA C Programming Guide, Chapter 5

2. **Bank conflicts en shared memory**: 32 banks, accesos al mismo bank serializan
   - *Tecnica*: Padding automatico de arrays en shared memory
   - *Bibliografia*: "Optimizing Parallel Reduction in CUDA" (Harris, Nvidia)

Testing Fase 2.1: - [ ] Unit tests: GPUExpr bien formado - [ ] Analisis estatico de coalescing - [ ] Simulacion de bank conflicts

Fase 2.2: Lowering MatExpr -> GPUExpr (Semanas 4-7) **Objetivo**: Transformar operaciones matriciales a kernels GPU.

Entregables:

```
/-- Convertir producto Kronecker a kernel paralelo -/
def lowerKronToGPU (A : MatExpr m n) (B : MatExpr p q) : GPUExpr :=
  GPUExpr.parallel
    (gridSize := m * p)
    (blockSize := 256)
    (GPUExpr.compute kroneckerKernel [lowerToGPU A, lowerToGPU B])
```

Obstaculos tecnicos: 1. **Tiling para shared memory**: Matrices grandes no caben en shared - *Tecnica*: Algoritmos de tiled matrix multiplication - *Bibliografia*: Volkov "Better Performance at Lower Occupancy"

2. **Sincronizacion entre bloques**: GPU no tiene sync global facil
   - *Tecnica*: Multiples kernel launches para fases
   - *Bibliografia*: "Cooperative Groups" (CUDA 9+)

Testing Fase 2.2: - [ ] Correctness: GPU result == CPU result - [ ] Performance: Medir GFLOPS, memory bandwidth - [ ] Occupancy analysis con nvprof

Fase 2.3: Generacion de CUDA C (Semanas 8-12) **Objetivo**: Generar codigo CUDA compilable y eficiente.

Entregables:

```
// Codigo generado por AMO-Lean
__global__ void fri_fold_kernel(
    const uint64_t* __restrict__ input,
    uint64_t* __restrict__ output,
    uint64_t alpha,
    size_t n
) {
    __shared__ uint64_t shared_input[512];

    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Coalesced load to shared memory
    if (idx * 2 + 1 < n * 2) {
        shared_input[threadIdx.x * 2] = input[idx * 2];
        shared_input[threadIdx.x * 2 + 1] = input[idx * 2 + 1];
    }
    __syncthreads();
```

```
    // Compute
    if (idx < n) {
        output[idx] = shared_input[threadIdx.x * 2]
                    + alpha * shared_input[threadIdx.x * 2 + 1];
    }
}
```

Obstaculos tecnicos: 1. **Aritmetica modular en GPU**: GPUs no tienen uint64 mul nativo eficiente - *Tecnica*: PTX assembly para umul.wide - *Bibliografia*: "Montgomery Multiplication on GPUs" (Emmart et al.)

2. **Divergencia de warps**: Branches en kernel causan serializacion
   - *Tecnica*: Predicacion en lugar de branches
   - *Bibliografia*: CUDA Best Practices Guide

Testing Fase 2.3: - [ ] Compilacion exitosa con nvcc - [ ] Correctness vs implementacion CPU - [ ] Benchmark vs implementaciones estado del arte (bellman, plonky3)

Fase 2.4: Optimizacion y Profiling (Semanas 13-18) **Objetivo**: Alcanzar performance competitiva con implementaciones manuales.

Entregables: - Kernels optimizados para Poseidon, FRI fold, Merkle - Dashboard de performance - Documentacion de trade-offs

Obstaculos tecnicos: 1. **Memory-bound vs compute-bound**: Diferentes kernels tienen diferentes bottlenecks - *Tecnica*: Roofline analysis - *Bibliografia*: Williams et al. "Roofline Model"

2. **Multi-GPU**: Scaling a multiples GPUs
   - *Tecnica*: NCCL para comunicacion
   - *Bibliografia*: "Scaling Deep Learning on GPU Clusters"

Testing Fase 2.4: - [ ] Roofline plot para cada kernel - [ ] Comparacion con prover de referencia (Stone, Plonky3) - [ ] Tests de regresion de performance

---

3. Variantes AVX-512

3.1 Contexto y Motivacion

AVX-512 ofrece registros de 512 bits (8 doubles o 8 uint64), potencialmente 2x sobre AVX2.

3.2 Desafio Tecnico Principal

Thermal throttling: Muchos CPUs reducen frecuencia 10-20% cuando usan AVX-512 intensivamente.

3.3 Plan de Implementacion por Fases

Fase 3.1: Infraestructura AVX-512 (Semana 1) **Objetivo**: Detectar soporte y configurar generacion condicional.

Entregables:

```
/-- Detectar capacidades SIMD en runtime -/
structure SIMDCapabilities where
  hasAVX2 : Bool
  hasAVX512F : Bool
  hasAVX512DQ : Bool  -- Double/Quad word
  hasAVX512IFMA : Bool  -- Integer FMA (para Montgomery)
```

```
/-- Seleccionar backend segun capabilities -/
def selectSIMDBackend (caps : SIMDCapabilities) (kernel : Kernel) : CodeGen :=
  if caps.hasAVX512IFMA && kernel.needsMontgomery then avx512IFMAGen
  else if caps.hasAVX512F then avx512Gen
  else if caps.hasAVX2 then avx2Gen
  else scalarGen
```

Obstaculos tecnicos: 1. **Runtime dispatch**: Seleccionar version optima en runtime - *Tecnica*: Function multi-versioning (GCC/Clang **attribute**((target))) - *Bibliografia*: Intel Architecture Optimization Manual

Testing Fase 3.1: - [ ] Deteccion correcta en varios CPUs - [ ] Fallback graceful si no hay soporte

Fase 3.2: Kernels AVX-512 (Semana 2) **Objetivo**: Portar kernels existentes a AVX-512.

Entregables:

```
// FRI fold con AVX-512
void fri_fold_avx512(size_t n, const uint64_t* input,
                     uint64_t* output, uint64_t alpha) {
    __m512i valpha = _mm512_set1_epi64(alpha);

    for (size_t i = 0; i < n; i += 8) {
        // Cargar 16 elementos (8 pares)
        __m512i even = _mm512_loadu_si512(&input[i * 2]);
        __m512i odd = _mm512_loadu_si512(&input[i * 2 + 8]);

        // Deinterleave
        __m512i evens = _mm512_unpacklo_epi64(even, odd);
        __m512i odds = _mm512_unpackhi_epi64(even, odd);

        // output = even + alpha * odd
        __m512i prod = _mm512_mullo_epi64(valpha, odds);
        __m512i result = _mm512_add_epi64(evens, prod);

        _mm512_storeu_si512(&output[i], result);
    }
}
```

Obstaculos tecnicos: 1. **Alineacion 64 bytes**: AVX-512 requiere alineacion estricta para loads/stores alineados - *Tecnica*: `aligned_alloc(64, size)` o pragmas de alineacion - *Bibliografia*: Intel Intrinsics Guide

Testing Fase 3.2: - [ ] Correctness vs version escalar - [ ] Benchmark vs AVX2

Fase 3.3: Cost Model con Thermal Awareness (Semana 3) **Objetivo**: Decidir cuando usar AVX-512 considerando throttling.

Entregables:

```
/-- Cost model que considera thermal throttling -/
structure AVX512CostModel extends CostModel where
  thermalPenalty : Float := 0.85  -- 15% frequency reduction typical
  transitionCost : Nat := 1000   -- Cycles para cambiar de modo

/-- Decidir si usar AVX-512 para un kernel -/
def shouldUseAVX512 (model : AVX512CostModel) (kernel : Kernel) : Bool :=
  let avx512Cost := kernel.computeCost / 2 * model.thermalPenalty + model.transitionCost
  let avx2Cost := kernel.computeCost
  avx512Cost < avx2Cost
```

Obstaculos tecnicos: 1. **Variabilidad entre CPUs**: Throttling varia significativamente - *Tecnica*: Benchmarking adaptativo en warmup - *Bibliografia*: Agner Fog "Microarchitecture of Intel/AMD CPUs"

Testing Fase 3.3: - [ ] Benchmark en varios CPUs (Xeon, desktop) - [ ] Verificar que seleccion automatica es optima

---

4. FRI Query Phase

4.1 Contexto y Motivacion

La Query Phase verifica la proximidad: 1. El verificador envia indices aleatorios 2. El prover responde con valores y Merkle proofs 3. El verificador chequea consistencia

4.2 Desafio Tecnico Principal

Patron de acceso aleatorio: A diferencia de Commit (secuencial, vectorizable), Query hace saltos aleatorios en memoria, causando cache misses.

4.3 Plan de Implementacion por Fases

Fase 4.1: Modelado de Query (Semanas 1-2) **Objetivo**: Definir tipos y operaciones para Query phase.

Entregables:

```
/-- Indices de query (generados por verificador via Fiat-Shamir) -/
structure QueryIndices where
  indices : Array Nat
  round : Nat

/-- Respuesta del prover -/
structure QueryResponse where
  values : Array FieldElement
  merkleProofs : Array MerkleProof
  siblingValues : Array FieldElement

/-- Verificar una query -/
def verifyQuery (commitment : MerkleRoot) (query : QueryIndices)
    (response : QueryResponse) : Bool :=
  -- Verificar Merkle proofs
  response.merkleProofs.all (verifyMerkleProof commitment) &&
  -- Verificar consistencia de fold
  verifyFoldConsistency query.indices response.values response.siblingValues
```

Obstaculos tecnicos: 1. **Generacion de indices deterministica**: Debe ser reproducible - *Tecnica*: Derivar de transcript usando squeeze - *Bibliografia*: FRI paper, Section 4

Testing Fase 4.1: - [ ] Roundtrip: commit -> query -> verify - [ ] Verificar que indices son deterministicos

Fase 4.2: Optimizacion de Merkle Proofs (Semanas 3-4) **Objetivo**: Minimizar accesos a memoria en verificacion.

Entregables:

```
/-- Batch multiple Merkle proofs para amortizar accesos -/
def batchMerkleProofs (root : MerkleRoot) (indices : Array Nat)
    (tree : FlatMerkle) : Array MerkleProof :=
```

```
-- Ordenar indices para localidad de acceso
let sortedIndices := indices.qsort (. < .)
-- Extraer proofs compartiendo nodos comunes
extractBatchedProofs root sortedIndices tree
```

Obstaculos tecnicos: 1. **Prefetching**: Indicar al CPU que memoria necesitaremos - *Tecnica*: `__builtin_prefetch` basado en patron de queries - *Bibliografia*: "What Every Programmer Should Know About Memory" (Drepper)

Testing Fase 4.2: - [ ] Benchmark: batched vs naive - [ ] Profile cache misses con perf

Fase 4.3: CodeGen para Query (Semanas 5-6) **Objetivo**: Generar codigo C optimizado para verificacion.

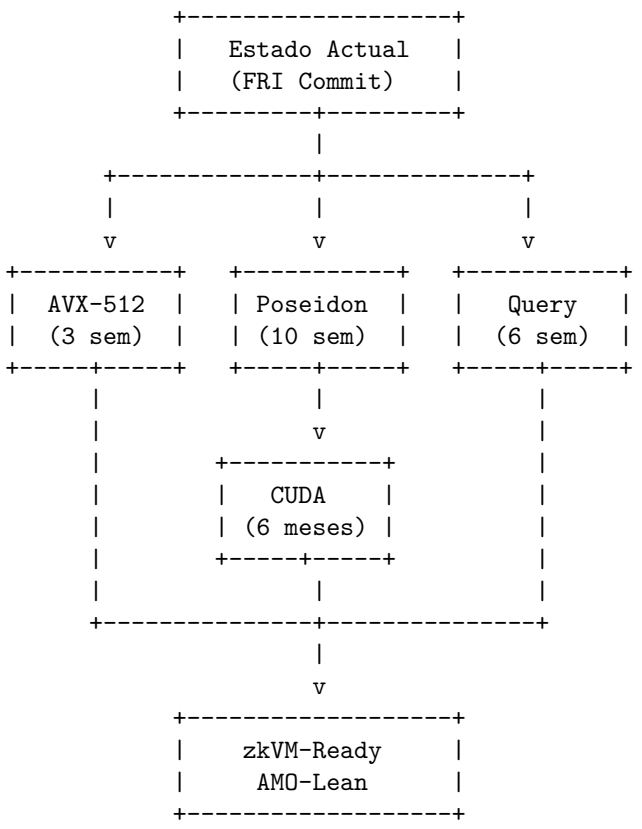
Entregables: - `generateQueryVerifier : QueryParams -> String` - Codigo con prefetching hints - Proof anchors para verificacion

Obstaculos tecnicos: 1. **Branch prediction**: Verificacion tiene muchos branches - *Tecnica*: Branchless comparisons donde sea posible - *Bibliografia*: "Branch Prediction" (Patterson & Hennessy)

Testing Fase 4.3: - [ ] Differential fuzzing vs implementacion Lean - [ ] Benchmark en diferentes tamanos de arbol

---

Dependencias y Orden de Ejecucion



Camino critico: Poseidon -> CUDA (maxima duracion)

Paralelizable: AVX-512 y Query pueden desarrollarse en paralelo con Poseidon.

Metricas de Exito

Hito	Metrica	Target
Poseidon	Throughput	>= 1M hashes/segundo (CPU)
AVX-512	Speedup vs AVX2	>= 1.5x
CUDA	Speedup vs CPU	>= 10x
Query	Verificacion	< 100ms para arbol de 2^20 hojas
Correctness	Tests	100% passing, differential fuzzing
Verificacion	Sorry count	Minimo, documentados

Bibliografia Completa

Criptografia ZK

1. Grassi et al. "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems" (USENIX 2021)
2. Ben-Sasson et al. "Fast Reed-Solomon Interactive Oracle Proofs of Proximity" (ICALP 2018)
3. Ames et al. "Ligero: Lightweight Sublinear Arguments Without a Trusted Setup" (CCS 2017)

Optimizacion SIMD/GPU

4. Intel Architecture Optimization Reference Manual
5. CUDA C Programming Guide (Nvidia)
6. Harris "Optimizing Parallel Reduction in CUDA"
7. Volkov "Better Performance at Lower Occupancy"
8. Emmart et al. "Montgomery Multiplication on GPUs"

Compiladores y Verificacion

9. Willsey et al. "egg: Fast and Extensible Equality Saturation" (POPL 2021)
10. Gross et al. "Accelerating Verified-Compiler Development with a Verified Rewriting Engine" (ITP 2022)
11. Xi & Pfenning "Dependent Types in Practical Programming" (POPL 1999)

Performance

12. Drepper "What Every Programmer Should Know About Memory"
13. Agner Fog "Instruction Tables" y "Microarchitecture"
14. Williams et al. "Roofline: An Insightful Visual Performance Model"

Documento generado: Enero 2026 Proxima revision: Al completar Fase 1 (Poseidon)