ENGINEERING ANNEX

Mock-Θ Crypt / MTNCC — Arquitetura Criptográfica Pós-Quântica Não-Canônica

Anexo de Engenharia Técnica — Versão 1.0

Post Quantum Sistemas Criptográficos Avançados Ltda Inventores: Marcos Eduardo Elias | Lawrence Chung Koo Documento complementar ao Pedido de Patente de Invenção — INPI e-Patentes 4.0

PREFÁCIO JURÍDICO-TÉCNICO

Este Anexo de Engenharia tem função específica e precisa ser lido com esse enquadramento: **não é um documento acadêmico nem um manual de implementação final**. É um documento de suporte ao pedido de patente, cuja função é demonstrar, com precisão técnica suficiente, que a arquitetura descrita no Relatório Descritivo é **industrialmente realizável** — ou seja, que um técnico versado na área pode, a partir das especificações aqui contidas, construir uma implementação funcional do sistema sem necessitar de atos inventivos adicionais.

O padrão legal relevante é o do Art. 24 da Lei 9.279/96 (LPI): o relatório descritivo e seus anexos devem descrever clara e suficientemente o objeto da invenção, de modo a possibilitar sua realização por técnico no assunto. Este Anexo reforça este requisito em sua dimensão de engenharia computacional.

A arquitetura Mock-Theta Crypt / MTNCC é realizável. Todos os seus componentes — aritmética de campo finito, operadores de consistência, geração de transcript, encapsulamento e decapsulamento — são compostos exclusivamente de operações computacionalmente bem definidas, determinísticas e implementáveis com bibliotecas existentes. Não há nenhum componente que requeira resolução de problema aberto, acesso a hardware inexistente ou procedimento matemático não formalizado.

O presente Anexo documenta: (1) a arquitetura de software completa com suas interfaces formais; (2) os algoritmos detalhados de cada componente; (3) as estruturas de dados concretas com tamanhos calculados; (4) os requisitos de hardware e as opções de implementação; (5) os procedimentos de validação e harness de ataque; e

(6) os caminhos de implementação em C, Rust e FPGA com estimativas de performance.

PARTE I — ARQUITETURA DE SOFTWARE

1.1 Visão Geral da Pilha de Software

O sistema Mock-Theta Crypt / MTNCC é organizado em seis camadas horizontais com dependências estritas de baixo para cima. Nenhuma camada superior acessa internamente os dados de uma camada não imediatamente inferior. Esta restrição não é apenas de estilo arquitetural — ela é uma propriedade de segurança: garante que a "identidade da completação" não vaze transversalmente entre camadas.

```
CAMADA 6 — Protocolos de Aplicação
  VCI-MTNCC (Verificação Cega a Identidade)
  MTDV (Módulo de Transcrições Discretas Verificáveis)
  TLS-MTNCC (Extensão de Protocolo de Transporte)

CAMADA 5 — Primitivas Criptográficas
  Theta-KEM (Encapsulamento de Chave)
  Theta-Sign (Assinatura Digital)
  MT-Com (Esquema de Compromisso)
  MT-PRF (Função Pseudoaleatória)

CAMADA 4 — Engine de Operadores de Consistência
  Op_gamma (Transporte Modular)
  Op_cusp (Stress em Cúspides)
  Op_conv (Convolução e Dobramento)
  Op_spec (Regularidade Espectral)
  Op_res (Compatibilidade Residual)
  Harness de Ataque (Mandatório)

CAMADA 3 — Transcript Público
  GPMTNC (Gerador de Parâmetros Mock Theta Não-Canônicos)
  ATN (Algoritmo de Triagem de Não-Canonicidade)
  Serialização e Hashing de Transcript
  Estimador de Largura de Ambiguidade

CAMADA 2 — Aritmética de Campo e Polinômios
  Aritmética em Z_p (primos seguros)
  Multiplicação Polinomial (NTT-like)
  CSPRNG (Gerador Criptográfico Seguro)
  KDF (Função de Derivação de Chave)

CAMADA 1 — Primitivas Hardcoded e Constantes de Sistema
```

```
  Primos Seguros p por Nível de Segurança
  Parâmetros de FMT de Terceira Ordem (Ramanujan)
  Sementes de Operadores Determinísticos
  Constantes de Tolerância por Nível
```

Propriedade de isolamento crítica: Em toda a pilha, os dados privados Y = (g*, η) existem exclusivamente na Camada 5 (primitivas) e nas rotinas de geração da Camada 3 (GPMTNC). Eles nunca sobem para a Camada 4 (operadores) nem para as Camadas 5-6 além das interfaces explicitamente definidas. O verificador na Camada 4 nunca tem acesso a Y — ele opera exclusivamente sobre o transcript T e o candidato X.

1.2 Interfaces Formais Entre Camadas

As interfaces a seguir são definidas com tipos precisos e são as únicas superfícies de contato entre camadas. Qualquer implementação válida do sistema deve respeitar estas interfaces exatamente.

Interface Camada 1 → Camada 2

```
// Parâmetros de sistema por nível de segurança
SystemParams {
  lambda: SecurityLevel,       // {Theta1=80, Theta2=128, Theta3=256}
  p: BigInt,                   // Primo seguro de tamanho 2^(lambda*2)
  N: u32,                      // Truncamento da série: {512, 1024, 2048}
  M: u32,                      // Número de avaliações: {32, 64, 128}
  L: u32,                      // Número de operadores: {8, 16, 32}
  epsilon: Rational,           // Tolerância: {2^-40, 2^-64, 2^-80}
  tau_points: [FieldElement],  // M pontos de avaliação fixos
  op_seeds: [Bytes32],         // L sementes para operadores (derivadas de H(para
}
```

Interface Camada 2 → Camada 3

```
// Entrada do GPMTNC
GeneratorInput {
  params: SystemParams,
  shadow_seed: Bytes32,        // Semente para derivação do shadow g
  eta_seed: Bytes32,           // Semente para dados de rigidificação eta
}

// Saída do GPMTNC
GeneratorOutput {
  transcript: Transcript,      // Dados públicos completos
```

```
  private_key: PrivateKey,     // Y = (g_discrete, eta) -- NUNCA serializado para
  ambiguity_estimate: u128,    // Estimativa de |família válida|, deve ser >= 2^l
}
```

Interface Camada 3 → Camada 4

```
// Transcript público completo (toda informação pública do sistema)
Transcript {
  params_id: Hash256,          // Hash de SystemParams (referência, não os params
  coeffs: [FieldElement; N],   // a_0,...,a_{N-1} mod p
  evals: [(FieldElement, FieldElement); M], // (tau_i, v_i) pares de avaliação
  op_seeds: [Bytes32; L],      // Sementes dos operadores (derivadas de H(coeffs
  tolerance: u64,              // Epsilon codificado como inteiro
  transcript_hash: Hash256,    // H(todos os campos acima, exceto este)
}

// Candidato a verificar (entrada do verificador)
Candidate {
  data: [FieldElement; N],     // Coefficients of candidate completion
  evals: [(FieldElement, FieldElement); M], // Avaliações do candidato
  meta: CandidateMeta,         // Dados auxiliares do candidato
}

// Resultado do verificador
VerifyResult {
  accepted: bool,
  operator_scores: [Rational; L],  // Score de cada operador (para auditoria)
  overall_score: Rational,
}
```

Interface Camada 4 → Camada 5 (Primitivas)

```
// Interface do Theta-KEM
ThetaKEM {
  fn keygen(params: SystemParams) -> (PublicKey, PrivateKey)
  fn encaps(pk: PublicKey) -> (Ciphertext, SessionKey)
  fn decaps(sk: PrivateKey, pk: PublicKey, ct: Ciphertext) -> Result<SessionKey,
}

// Interface do Theta-Sign
ThetaSign {
  fn sign(sk: PrivateKey, message: Bytes) -> Signature
  fn verify(pk: PublicKey, message: Bytes, sig: Signature) -> bool
}
```

```
// Interface do MT-Com
MTCom {
  fn commit(value: Bytes, randomness: Bytes32) -> Commitment
  fn open(commitment: Commitment, value: Bytes, randomness: Bytes32) -> bool
}

// Interface do MT-PRF
MTPRF {
  fn eval(key: PrivateKey, input: Bytes) -> Bytes32
}
```

PARTE II — ALGORITMOS DETALHADOS

2.1 GPMTNC — Gerador de Parâmetros Mock Theta Não-Canônicos

O GPMTNC é o componente de inicialização do sistema. Sua função é amostrar instâncias da função mock theta de terceira ordem de Ramanujan, verificar que a instância amostrada satisfaz as condições de não-canonicidade, e produzir o par (transcript público, dados privados de completação).

Algoritmo GPMTNC Completo

```
GPMTNC(params: SystemParams, entropy: Bytes64) -> (Transcript, PrivateKey):

PASSO 1 — Geração do Shadow g
  // O shadow é uma forma cúspide de peso complementar representada discretamente
  // como um polinômio sobre Z_p de grau <= N-1

  g_seed = KDF(entropy, "shadow_seed_v1", 32)
  g_raw = sample_polynomial(g_seed, params.p, params.N)
  // g_raw é polinômio de grau N-1 sobre Z_p

  // Verificação de não-trivialidade: g deve ter norma suficiente
  g_norm = polynomial_l2_norm(g_raw, params.p)
  assert g_norm > threshold_norm(params.lambda)
  // Se falhar, resampling com g_seed = H(g_seed || "retry")

PASSO 2 — Computação dos Coeficientes da FMT
  // f(q) = SUM_{n=0}^{N-1} q^{n^2} / ((-q;q)_n)^2  [truncada a N termos]
  // Instanciada sobre Z_p com q = derive_q(params, g_raw)

  q = derive_q(params, g_raw)
```

```
  // derive_q: deterministic derivation of field element from shadow
  // q deve ser elemento não-zero de Z_p com ordem multiplicativa suficiente
  assert multiplicative_order(q, params.p) > params.N^2

  coeffs = [FieldElement; N]  // Inicializado a zero

  // Computação iterativa dos coeficientes de Pochhammer
  q_neg = field_neg(q, params.p)   // -q mod p
  pochhammer = [FieldElement; N]    // (a;q)_n para cada n
  pochhammer[0] = 1
  for n in 1..N:
    pochhammer[n] = pochhammer[n-1] * (1 - q_neg * q^(n-1)) mod p

  // Coeficientes da série truncada
  for n in 0..N:
    q_power = field_pow(q, n*n, params.p)   // q^{n^2}
    denom = field_pow(pochhammer[n], 2, params.p)  // ((-q;q)_n)^2
    denom_inv = field_inv(denom, params.p)         // Inverso modular
    coeffs[n] = q_power * denom_inv mod params.p

PASSO 3 — Computação das Avaliações
  // M avaliações em pontos estruturalmente selecionados
  // Os pontos tau são derivados de params e são públicos

  evals = [(FieldElement, FieldElement); M]
  for i in 0..M:
    tau_i = params.tau_points[i]
    v_i = evaluate_polynomial(coeffs, tau_i, params.p)
    // Horner's method: O(N) por avaliação
    evals[i] = (tau_i, v_i)

PASSO 4 — Derivação das Sementes de Operadores
  // As sementes de operadores são derivadas deterministicamente do transcript
  // Isso garante que o verificador (que tem acesso ao transcript) pode recomputá

  transcript_data = serialize(coeffs, evals)
  for j in 0..L:
    op_seeds[j] = KDF(transcript_data, "op_seed_" || j, 32)

PASSO 5 — Verificação de Não-Canonicidade (ATN)
  // O ATN verifica que a instância gerada satisfaz a condição de não-canonicidad
  // que a distância espectral ao espaço de objetos com completação canônica
  // supera o limiar calibrado em função de lambda

  spectral_dist = compute_spectral_distance(coeffs, evals, params)

  if spectral_dist < non_canonicity_threshold(params.lambda):
```

```
    // Instância falhou no ATN — resampling completo
    // Incrementa entropy via hash para evitar loop infinito
    entropy = H(entropy || "atn_retry")
    return GPMTNC(params, entropy)

  // ATN passou

PASSO 6 — Dados de Rigidificação eta
  // eta são dados auxiliares que selecionam uma completação específica
  // dentro da família admissível. São categoricamente distintos dos coeficientes

  eta_seed = KDF(entropy, "eta_seed_v1", 32)
  eta = derive_eta(g_raw, eta_seed, params)
  // eta tem tamanho de 32 bytes por padrão (independente de N)

PASSO 7 — Montagem do Transcript e Chave Privada
  transcript_hash = H("transcript_v1" || params_id || coeffs || evals || op_seeds

  transcript = Transcript {
    params_id: H(params),
    coeffs: coeffs,
    evals: evals,
    op_seeds: op_seeds,
    tolerance: encode_epsilon(params.epsilon),
    transcript_hash: transcript_hash,
  }

  private_key = PrivateKey {
    g: g_raw,           // Shadow (categoricamente ausente do transcript)
    eta: eta,           // Dados de rigidificação
    params: params,     // Referência aos parâmetros
  }

  // Verificação final: o transcript não contém g ou eta em nenhuma forma
  assert transcript_does_not_contain(transcript, private_key)
  // Esta asserção é verificável por inspeção da função de geração
  // g nunca aparece no transcript: coeffs são funções de g via derive_q,
  // mas derive_q é uma função one-way: dado q, não é possível recuperar g

  return (transcript, private_key)

ESTIMATIVA DE CUSTO COMPUTACIONAL:
  - Passos 1-2: O(N) operações de campo, dominado por N inversões modulares
  - Passo 3: O(N*M) operações de campo (M avaliações de polinômio de grau N)
  - Passo 4: O(L * |transcript_data|) operações de hash
  - Passo 5 (ATN): O(N log N) via NTT + O(N*M) para distância espectral
  - Total: O(N*M) operações de campo + O(N log N) NTT
```

```
  - Para Theta-2 (N=1024, M=64): ~66.000 operações de campo
```

Sub-algoritmo: Compute_Spectral_Distance (ATN)

```
compute_spectral_distance(coeffs, evals, params) -> Rational:
  // Mede a distância da instância ao espaço de objetos modulares canonicamente d
  // Instâncias próximas ao espaço modular poderiam ter suas completações inferid
  // por técnicas de lifting analítico — o ATN as descarta

  PASSO A — Computação da Matriz de Consistência Modular
    // Uma instância é "quase-modular" se satisfaz relações de transformação
    // próximas às relações modulares. Medimos o quão longe a instância está.

    gamma_matrices = get_congruence_generators(params)  // Matrizes SL(2,Z/p) sel

    consistency_scores = []
    for gamma in gamma_matrices:
      // Transforma os coeficientes segundo ação de gamma
      transformed_coeffs = apply_modular_transform(coeffs, gamma, params)
      // Mede discrepância entre coeffs originais e transformados
      discrepancy = l2_distance(coeffs, transformed_coeffs, params.p)
      consistency_scores.append(discrepancy)

  PASSO B — Score de Distância Espectral
    // A distância espectral é a mínima discrepância sobre os geradores
    // Uma instância modular teria distância zero
    // Uma instância não-canônica bem escolhida tem distância grande

    spectral_distance = min(consistency_scores)

    // Normalização pelo tamanho do espaço de coeficientes
    normalized_distance = spectral_distance / sqrt(N * p)

    return normalized_distance

LIMIAR DE NÃO-CANONICIDADE POR NÍVEL:
  Theta-1 (lambda=80):  normalized_distance > 2^{-20}
  Theta-2 (lambda=128): normalized_distance > 2^{-32}
  Theta-3 (lambda=256): normalized_distance > 2^{-64}

  // Interpretação: instâncias com distância menor que o limiar são "suspeitas de
  // canonicidade" e são descartadas. O ATN rejeita tipicamente <5% das instância
  // geradas para parâmetros bem escolhidos.
```

2.2 Engine de Operadores — Algoritmos Detalhados

O Engine de Operadores é o coração do verificador. Ele implementa os cinco operadores de consistência que juntos definem a família admissível de completações. Cada operador é **completamente determinístico** dado o transcript T: não requer acesso aos dados privados Y.

Propriedade fundamental do verificador: Para qualquer candidato X que pertença à família admissível, todos os cinco operadores retornam score dentro da tolerância. Para candidatos fora da família, ao menos um operador retorna score fora da tolerância. Esta propriedade pode ser testada empiricamente no harness de ataque.

Operador 1: Op_gamma (Transporte Modular)

```
Op_gamma(transcript: Transcript, candidate: Candidate) -> Score:
  // Verifica que o candidato satisfaz relações de consistência modular
  // derivadas dos geradores de congruência parametrizados pelo transcript

  // Recuperação das sementes de operador
  seed = transcript.op_seeds[0]  // Primeira semente, dedicada ao Op_gamma

  // Derivação dos parâmetros do operador a partir da semente
  gamma_params = derive_gamma_params(seed, transcript.params_id)
  // gamma_params contém: {
  //   n_transforms: u32,      // Número de transformações a verificar
  //   transform_matrices: [SL2Matrix],  // Matrizes de transformação
  //   weight_vector: [Rational],        // Pesos para o score agregado
  // }

  scores = []
  for (matrix, weight) in zip(gamma_params.transform_matrices, gamma_params.weigh
    // Aplica transformação ao candidato
    transformed = apply_quasi_modular_transform(candidate.data, matrix, transcrip

    // Mede consistência: o quanto o candidato satisfaz a relação de transformaçã
    // Candidatos da família admissível devem satisfazer relações "quase-modulare
    // com desvio menor que epsilon
    deviation = compute_deviation(candidate.data, transformed, transcript.toleran
    scores.append((1.0 - deviation) * weight)

  // Score agregado ponderado
  aggregate_score = weighted_sum(scores)

  return Score {
    operator: "Op_gamma",
```

```
    value: aggregate_score,
    threshold: 1.0 - transcript.tolerance,
    passed: aggregate_score >= (1.0 - transcript.tolerance)
  }

COMPLEXIDADE: O(n_transforms * N) = O(L_gamma * N)
  Para Theta-2: n_transforms ~= 8, N = 1024 → ~8.192 operações de campo por verif
```

Operador 2: Op_cusp (Stress em Cúspides)

```
Op_cusp(transcript: Transcript, candidate: Candidate) -> Score:
  // Verifica o comportamento do candidato em vizinhanças de cúspides
  // Funções mock theta têm comportamento assintótico específico perto de cúspide
  // Este operador verifica que o candidato honra esse comportamento

  seed = transcript.op_seeds[1]
  cusp_params = derive_cusp_params(seed, transcript.params_id)
  // cusp_params contém: {
  //   cusp_points: [FieldElement],   // Pontos próximos a cúspides (raízes da un
  //   asymptotic_model: AsymptoticModel,  // Modelo de decaimento esperado
  //   window_size: u32,              // Tamanho da janela de avaliação
  // }

  // Para cada cúspide, avalia o candidato em uma janela de pontos
  // e compara com o comportamento assintótico esperado de FMTs de 3ª ordem

  cusp_scores = []
  for cusp in cusp_params.cusp_points:
    // Avalia candidato em janela ao redor da cúspide
    window_values = evaluate_in_window(candidate.data, cusp, cusp_params.window_s

    // Modelo assintótico de FMT de 3ª ordem na cúspide:
    // f(q) ~ A * q^alpha * (1 + O(q)) para q → raiz da unidade
    // onde alpha = -1/24 para FMTs de ordem 3 de Ramanujan
    expected_behavior = compute_expected_cusp_behavior(cusp, cusp_params.asymptot

    // Desvio do comportamento esperado
    cusp_deviation = compare_behavior(window_values, expected_behavior, transcrip
    cusp_scores.append(1.0 - cusp_deviation)

  return Score { value: min(cusp_scores), ... }

// NOTA TÉCNICA: O comportamento nas cúspides é uma propriedade das FMTs que
// distingue completações da família admissível de candidatos arbitrários.
// Candidatos aleatórios falham tipicamente neste operador com probabilidade ~1 -
```

Operador 3: Op_conv (Convolução e Dobramento)

```
Op_conv(transcript: Transcript, candidate: Candidate) -> Score:
  // Verifica invariantes de convolução e dobramento do candidato
  // Esses invariantes são análogos discretos de relações de Hecke

  seed = transcript.op_seeds[2]
  conv_params = derive_conv_params(seed, transcript.params_id)

  // Computa convolução do candidato com kernel derivado da semente
  kernel = derive_convolution_kernel(seed, transcript)
  // kernel é polinômio de grau <= N/2 sobre Z_p

  convolution = polynomial_multiply(candidate.data, kernel, transcript.params)
  // polynomial_multiply: NTT-based, O(N log N)

  // O resultado da convolução deve satisfazer propriedades de simetria
  // específicas de FMTs de 3ª ordem. Para candidatos da família admissível,
  // a convolução é "autocomplementar" com desvio <= epsilon.

  // Verifica simetria de dobramento
  folded = fold_polynomial(convolution, transcript.params.N)
  // fold: f(x) -> f(x) + f(-x) sobre Z_p

  symmetry_deviation = l2_distance(convolution[:N/2], folded[:N/2], transcript.pa
  symmetry_score = 1.0 - normalize(symmetry_deviation, transcript.tolerance)

  return Score { value: symmetry_score, ... }

COMPLEXIDADE: O(N log N) por NTT + O(N) para fold e comparação
  Para Theta-2: N=1024 → ~10.240 operações (com constante de NTT ~10)
```

Operador 4: Op_spec (Regularidade Espectral)

```
Op_spec(transcript: Transcript, candidate: Candidate) -> Score:
  // Verifica que o espectro de frequências do candidato satisfaz
  // perfil de decaimento compatível com FMTs de 3ª ordem

  seed = transcript.op_seeds[3]
  spec_params = derive_spec_params(seed, transcript.params_id)

  // Computa transformada NTT do candidato
  spectrum = ntt(candidate.data, transcript.params.p)

  // FMTs de 3ª ordem têm espectro com decaimento sub-exponencial específico:
```

```
  // |spectrum[k]| ~ C * exp(-alpha * k / N) para k crescente
  // onde alpha e C dependem dos parâmetros de instância

  expected_decay = derive_decay_profile(spec_params, transcript)
  actual_decay = compute_decay_profile(spectrum, transcript.params)

  // Score baseado na distância L1 entre perfis de decaimento esperado e real
  decay_distance = l1_distance_normalized(expected_decay, actual_decay)
  spectral_score = max(0.0, 1.0 - decay_distance / transcript.tolerance)

  return Score { value: spectral_score, ... }

COMPLEXIDADE: O(N log N) para NTT + O(N) para comparação de perfil
```

Operador 5: Op_res (Compatibilidade Residual)

```
Op_res(transcript: Transcript, candidate: Candidate) -> Score:
  // Verifica compatibilidade entre coeficientes do candidato e avaliações no tra
  // Este é o operador de "ligação" entre as duas representações do transcript

  seed = transcript.op_seeds[4]

  // Recomputa avaliações do candidato nos pontos tau do transcript
  candidate_evals = []
  for tau_i in transcript.params.tau_points[:M]:
    v_candidate = evaluate_polynomial(candidate.data, tau_i, transcript.params.p)
    candidate_evals.append(v_candidate)

  // Compara com avaliações do transcript
  // Candidatos da família admissível devem produzir avaliações compatíveis
  // com as do transcript, dentro da tolerância epsilon

  eval_errors = []
  for (v_transcript, v_candidate) in zip(transcript.evals, candidate_evals):
    error = field_distance(v_transcript.1, v_candidate, transcript.params.p)
    eval_errors.append(error)

  max_error = max(eval_errors)
  residual_score = max(0.0, 1.0 - max_error / transcript.tolerance)

  // Verifica também a hash de integridade do transcript
  computed_hash = H("transcript_v1" || transcript.params_id || transcript.coeffs
                    transcript.evals || transcript.op_seeds || transcript.tolera
  hash_valid = (computed_hash == transcript.transcript_hash)
```

```
  if not hash_valid:
    return Score { value: 0.0, passed: false, reason: "transcript_hash_invalid" }

  return Score { value: residual_score, ... }

COMPLEXIDADE: O(N*M) para avaliações + O(|transcript|) para hash
  Para Theta-2: N=1024, M=64 → ~65.536 operações de campo

  NOTA: Op_res é o mais custoso dos cinco operadores. Em implementações otimizada
  as avaliações podem ser precomputadas e cacheadas para verificações repetidas
  do mesmo transcript.
```

Verificador Agregado

```
Verify(params: SystemParams, transcript: Transcript, candidate: Candidate) -> Ver
  // Executa todos os cinco operadores e agrega os resultados

  scores = [
    Op_gamma(transcript, candidate),
    Op_cusp(transcript, candidate),
    Op_conv(transcript, candidate),
    Op_spec(transcript, candidate),
    Op_res(transcript, candidate),
  ]

  // Todos os operadores devem passar (AND, não maioria)
  // Isso é deliberado: um único operador fraco comprometeria a família admissíve
  all_passed = all(score.passed for score in scores)

  overall_score = geometric_mean(score.value for score in scores)

  return VerifyResult {
    accepted: all_passed,
    operator_scores: scores,
    overall_score: overall_score,
  }

COMPLEXIDADE TOTAL DE Verify():
  Theta-1: ~50.000 operações de campo, ~1ms em CPU moderna
  Theta-2: ~200.000 operações de campo, ~4ms em CPU moderna
  Theta-3: ~800.000 operações de campo, ~16ms em CPU moderna

  NOTA SOBRE TEMPO CONSTANTE: Toda a execução de Verify() deve ser implementada
  em tempo constante relativamente à identidade do candidato. Não deve haver
  branches condicionais que dependam dos valores de campos do candidato durante
```

a comparação de scores. Isso é uma propriedade de segurança de canal lateral.

2.3 Theta-KEM — Algoritmo Completo com Análise de Segurança

KeyGen

```
ThetaKEM_KeyGen(params: SystemParams) -> (PublicKey, PrivateKey):
  // Fonte de entropia: CSPRNG do sistema operacional
  entropy = os_csprng(64)  // 64 bytes = 512 bits de entropia

  // Delegação ao GPMTNC
  (transcript, private_key) = GPMTNC(params, entropy)

  // Verificação de ambiguidade: transcript deve admitir >= 2^lambda completações
  ambiguity = estimate_ambiguity(transcript, params)

  if ambiguity < 2^params.lambda:
    // Falha de geração: parâmetros insuficientes para o nível de segurança
    // Não deve ocorrer com parâmetros corretos; retorna erro
    return Error("insufficient_ambiguity")

  // PublicKey é simplesmente o transcript
  public_key = PublicKey { transcript: transcript }

  // PrivateKey contém o shadow e os dados de rigidificação
  // CRÍTICO: PrivateKey NUNCA deve ser serializado junto ao transcript
  //          Deve ser armazenado separadamente com proteção de HSM/encriptação

  return (public_key, private_key)

PROPRIEDADE: pk contém zero informação sobre sk além do que é implicado pela
             existência de um sistema válido. Especificamente:
             - pk não contém g (shadow)
             - pk não contém eta (dados de rigidificação)
             - pk não contém q (elemento de campo derivado de g)
             - derive_q(g) é uma função one-way: dado q, recuperar g é MMIP-ID
```

Encaps

```
ThetaKEM_Encaps(pk: PublicKey) -> (Ciphertext, SessionKey):
  transcript = pk.transcript
  params = lookup_params(transcript.params_id)

  PASSO 1 — Geração do Seletor Efêmero
```

```
  s = csprng(params.lambda / 8)  // lambda bits de aleatoriedade
  // s é o valor secreto que define a chave de sessão
  // Após Encaps, s não deve ser retido pelo encapsulador

PASSO 2 — Derivação do Pseudo-Shadow
  // Mapeia s para um elemento do espaço de shadows via função pública
  // Esta função é uma "injeção pseudo-shadow": mapeia o seletor s para
  // uma representação no espaço de dados de completação, sem revelar
  // nenhuma informação sobre o shadow verdadeiro g*

  Y_s = derive_pseudo_shadow(s, transcript)
  // derive_pseudo_shadow:
  //   Y_s.g_approx = KDF(s, "pseudo_shadow_g", sizeof_g_representation)
  //   Y_s.eta_approx = KDF(s, "pseudo_shadow_eta", 32)
  //   // Y_s não é um shadow válido por construção
  //   // mas é um elemento bem formado do espaço de shadows

PASSO 3 — Geração do Texto Cifrado
  // Embed injeta Y_s na classe de ambiguidade do transcript
  // produzindo um objeto que:
  //   (a) passa na verificação Verify(params, transcript, .)
  //   (b) é consistente com MÚLTIPLAS completações da família admissível
  //   (c) é distinguível de ct alternativo APENAS por quem possui sk = Y*

  ct_data = embed_in_family(Y_s, transcript, params)
  // embed_in_family:
  //   Computa data = linear_combination(transcript.coeffs, Y_s, params)
  //   tal que Verify(params, transcript, Candidate{data}) = aceitar
  //   A ambiguidade é estrutural: ct_data é consistente com >= 2^lambda Y'

  // Verifica que ct passou na verificação (sanity check)
  candidate = Candidate { data: ct_data, evals: compute_evals(ct_data, params)
  verify_result = Verify(params, transcript, candidate)
  assert verify_result.accepted

  // Serializa texto cifrado
  ct = Ciphertext {
    data: ct_data,
    evals: candidate.evals,
    ct_hash: H("ciphertext_v1" || ct_data || candidate.evals),
  }

PASSO 4 — Derivação da Chave de Sessão
  K = KDF(s || transcript.transcript_hash || ct.ct_hash, "session_key_v1", 32)
  // K tem 32 bytes = 256 bits por padrão
  // K é derivada de s (secreto), do transcript (público) e do ct (público)
  // Sem s, K não pode ser derivada a partir de (transcript, ct)
```

```
    // Destruição segura de s após derivação de K
    secure_zero(s)

  return (ct, K)

PROPRIEDADE DE AMBIGUIDADE DO CT:
  Para qualquer observador sem sk = Y* = (g*, eta):
    - ct.data é consistente com >= 2^lambda escolhas de Y'
    - Cada Y' produziria uma chave de sessão K' diferente
    - Sem Y*, não há procedimento para distinguir Y* das outras Y'
    - Isso é exatamente MMIP-D: distinguir Y* de Y' dado (transcript, ct)
```

Decaps

```
ThetaKEM_Decaps(sk: PrivateKey, pk: PublicKey, ct: Ciphertext) -> Result<SessionK
  transcript = pk.transcript
  params = lookup_params(transcript.params_id)

  PASSO 1 — Verificação de Integridade do Texto Cifrado
    computed_ct_hash = H("ciphertext_v1" || ct.data || ct.evals)
    if computed_ct_hash != ct.ct_hash:
      return Error("ciphertext_tampered")

    // Verifica que ct é um membro da família admissível
    candidate = Candidate { data: ct.data, evals: ct.evals }
    verify_result = Verify(params, transcript, candidate)

    if not verify_result.accepted:
      return Error("ciphertext_not_in_family")

  PASSO 2 — Colapso de Ambiguidade Determinístico (CAD)
    // O CAD é a operação que distingue Decaps de qualquer adversário sem sk
    // Usando sk = Y* = (g*, eta), colapsa a ambiguidade do ct para extrair s

    // O CAD opera em O(M^{2/3}) = O(2^{2*lambda/3}) operações no pior caso
    // mas em implementações práticas com sk correto, opera em tempo polinomial

    // Extração do seletor efêmero s a partir do ct com o sk correto
    s_recovered = extract_with_cad(sk, transcript, ct, params)
    // extract_with_cad:
    //   1. Usa g* para computar a classe de equivalência de ct no torsor
    //   2. Usa eta para selecionar o único elemento da classe que mapeia para s
    //   3. Retorna s sem revelar g* ou eta em nenhum intermediário público
```

```
    if s_recovered == None:
      // Falha de decapsulamento: ct não foi gerado com esta chave pública
      // Isso indica ou ct inválido ou sk errado
      return Error("decaps_failed")

  PASSO 3 — Rederivação da Chave de Sessão
    K = KDF(s_recovered || transcript.transcript_hash || ct.ct_hash, "session_key

    // Verificação: K deve ser idêntica à K derivada pelo encapsulador
    // Esta verificação é implícita no protocolo — se s_recovered == s, K == K_en

    secure_zero(s_recovered)

  return Ok(K)

NOTA SOBRE SEGURANÇA DE DECAPS:
  A função Decaps revela zero informação sobre sk além do que já está implícito
  na existência de pk. Especificamente:
  - O valor de s_recovered nunca é exposto fora de Decaps
  - K é computada via KDF one-way de s, portanto K não revela s
  - Falha de Decaps retorna apenas "erro" sem detalhe sobre a natureza da falha
  - Implementações devem usar tempo constante em todas as comparações
```

2.4 Algoritmo do VCI-MTNCC com Análise Formal

```
VCI_MTNCC_Protocol:

  SETUP (executado uma vez por credencial):
    Provador P possui: sk = Y* = (g*, eta), id (identidade), pk = T
    Verificador V possui: params Pi, compromisso C = MT-Com(id, r)

    // C é publicado pelo Verificador sem revelar id
    // Apenas P conhece id; V conhece apenas C

  RODADA 1 — COMPROMISSO:
    1. P amostra efêmero: a ← Z_p uniformemente aleatório
       // a tem lambda bits de entropia

    2. P computa compromisso: A = MTF(a)
       // MTF: Mock Theta Function evaluation — função one-way
       // MTF(a) = evaluate_polynomial(transcript.coeffs, a, params.p)
       // Mais precisamente: A é derivado de a via a função f da FMT instanciada

    3. P envia A para V

    // Propriedade: A não revela a por one-wayness de MTF
```

```
  // Propriedade: A não revela sk por one-wayness e independência

RODADA 2 — DESAFIO:
  4. V amostra desafio: e ← {0,1}^lambda uniformemente aleatório
  5. V envia e para P

RODADA 3 — RESPOSTA:
  6. P computa resposta: z = a + e * sk mod p
     // sk aqui é um representante escalar do shadow g*, derivado via
     // sk_scalar = polynomial_to_scalar(g*, params)

  7. P envia (z, A) para V (A é incluído para conveniência de verificação)

VERIFICAÇÃO:
  8. V verifica: MTF(z) =?= A · pk^e mod p
     // Isso verifica que z = a + e*sk, portanto P conhece sk
     // sem que V obtenha informação sobre sk

  9. V verifica: C é compatível com pk via H_MTF
     // H_MTF: função hash específica do sistema que verifica que
     // o compromisso C foi gerado com a mesma chave pública pk

  10. Se ambas verificações passam: V aceita; caso contrário: V rejeita

PROPRIEDADES FORMAIS VERIFICADAS:

  Zero-Knowledge Perfeito:
    Simulação sem sk: amostrar z ← Z_p e e ← {0,1}^lambda uniformemente,
    computar A = MTF(z) / pk^e.
    Distribuição de (A, e, z) simulada é idêntica à distribuição honesta.
    Prova: A = MTF(a) com a = z - e*sk; dado z e e, a é uniforme em Z_p.

  Soundness Computacional:
    Extrator: dados dois transcripts (A, e, z) e (A, e', z') com e ≠ e',
    sk = (z - z') / (e - e') mod p.
    Extrator tem complexidade O(1) operações de campo.
    Bound de soundness: 2^{-lambda} por invocação.

  Cegueira de Identidade Estrutural:
    Transcripts de sessões distintas do mesmo sujeito são estatisticamente
    independentes porque a cada nova sessão, a é resampled uniformemente.
    Não há correlação entre (A1, e1, z1) e (A2, e2, z2) para o mesmo P.
    Isso deriva da estrutura de torsor do grupo subjacente: sem origem canônica
    dois elementos aleatórios do torsor são independentes na distribuição.

  Não-Transferibilidade:
    Um transcript (A, e, z) não pode ser reutilizado por terceiro T' para
```

```
provar a mesma credencial a V', porque T' não conhece sk e portanto
não pode responder a novos desafios e'' ≠ e.
Exceto com probabilidade 2^{-lambda} (tentativa de adivinhação de e'').
```

PARTE III — ESTRUTURAS DE DADOS E TAMANHOS

3.1 Tabela de Tamanhos por Nível de Segurança

Esta seção documenta os tamanhos exatos das estruturas de dados para cada nível de segurança. Os tamanhos são calculados analiticamente a partir dos parâmetros e são verificáveis por implementação.

Componente	Theta-1	Theta-2	Theta-3	Método de Cálculo
Primo p	256 bits = 32B	512 bits = 64B	1024 bits = 128B	2 × lambda bits
Coeficientes (N elementos de Z_p)	512 × 32 = 16.384B = 16KB	1024 × 64 = 65.536B = 64KB	2048 × 128 = 262.144B = 256KB	N × ceil(log2(p)/8)
Avaliações (M pares)	32 × 64 = 2.048B = 2KB	64 × 128 = 8.192B = 8KB	128 × 256 = 32.768B = 32KB	M × 2 × ceil(log2(p)/8)
Sementes de operadores (L × 32B)	8 × 32 = 256B	16 × 32 = 512B	32 × 32 = 1.024B	L × 32
Metadados (params_id, tolerance, hash)	64B	64B	64B	Constante
TAMANHO TOTAL DO TRANSCRIPT	**~18,8KB**	**~73,8KB**	**~289,9KB**	Soma das linhas acima
Chave Privada (shadow g + eta)	32B + 32B = 64B	64B + 32B = 96B	128B + 32B = 160B	ceil(log2(p)/8) + 32
Texto Cifrado (Theta-KEM)	~18,8KB + 32B overhead	~73,8KB + 32B overhead	~289,9KB + 32B overhead	≈ Transcript + overhead

Assinatura (Theta-Sign)	~18,8KB	~73,8KB	~289,9KB	≈ Transcript

Análise comparativa com NIST PQC:

Sistema	Chave Pública	Texto Cifrado	Chave Privada
Kyber-512 (NIST-I)	800B	768B	1.632B
Kyber-768 (NIST-III)	1.184B	1.088B	2.400B
Kyber-1024 (NIST-V)	1.568B	1.568B	3.168B
Theta-2 (NIST-III equiv.)	**73,8KB**	**73,8KB**	**96B**
Theta-3 (NIST-V equiv.)	**289,9KB**	**289,9KB**	**160B**

Observação técnica crítica: O Theta-Crypt tem chave pública e texto cifrado significativamente maiores que o Kyber, mas chave privada menor. O trade-off é deliberado e estrutural: a "largura" do transcript é o que cria a ambiguidade — um transcript menor teria menos ambiguidade e portanto menos segurança. Para aplicações onde o transcript pode ser armazenado centralmente e reutilizado (PKI, TLS com session resumption, autenticação recorrente), o overhead de transmissão é amortizado.

3.2 Estrutura Binária do Transcript

O transcript é serializado em formato binário determinístico little-endian para garantir reprodutibilidade bit-a-bit em todas as plataformas.

```
FORMATO BINÁRIO DO TRANSCRIPT (Theta-2, N=1024, M=64, L=16, p=512 bits)

Offset   Tamanho  Campo
-------- -------- ---------------------------------
0x0000   4B       Número mágico: 0x544D5443 ("TMTC")
0x0004   2B       Versão: 0x0001
0x0006   2B       Nível de segurança: 0x0002 (Theta-2)
0x0008   32B      params_id (hash SHA-256 dos parâmetros de sistema)
0x0028   4B       N: número de coeficientes (little-endian u32) = 1024
0x002C   4B       M: número de avaliações (little-endian u32) = 64
0x0030   4B       L: número de sementes (little-endian u32) = 16
0x0034   4B       epsilon_encoded (representação racional como u32)
0x0038   64B      Primo p (512 bits, little-endian)
0x0078   65536B   Coeficientes: 1024 elementos de 64B cada (little-endian, mod p)
```

```
0x10078  8192B    Avaliações: 64 pares de 64B × 2 = 128B cada
0x11E78  512B     Sementes de operadores: 16 × 32B
0x12078  32B      transcript_hash (SHA-256 de todos os campos acima)
0x12098  ---      FIM DO TRANSCRIPT

TAMANHO TOTAL: 0x12098 = 73.880 bytes ≈ 73,8 KB
Verificação: 32 + 2 + 2 + 2 + 32 + 4 + 4 + 4 + 4 + 64 + 65536 + 8192 + 512 + 32 =
// (diferença de 544B = fields de header não contados na estimativa anterior)
// Estimativa refinada: ~72,5KB

VERIFICAÇÃO DE INTEGRIDADE:
  H("transcript_v1" || [todos os campos exceto transcript_hash]) == transcript_ha
  Qualquer modificação em qualquer campo invalida o transcript_hash.
  O verificador DEVE verificar transcript_hash antes de executar qualquer operado
```

PARTE IV — PROCEDIMENTO DE VALIDAÇÃO E HARNESS DE ATAQUE

4.1 Harness de Ataque — Componente Mandatório

O harness de ataque é especificado como componente mandatório de qualquer implementação válida. Sua função é permitir validação empírica das propriedades de segurança do sistema — especialmente a sobrevivência da ambiguidade sob estresse adversarial.

Arquitetura do Harness:

```
HARNESS DE ATAQUE — ARQUITETURA COMPLETA

COMPONENTE 1: Gerador de Instâncias de Teste
  Input: params, seed, n_instances
  Output: [(transcript_i, sk_i, ambiguity_estimate_i)]

  Para cada instância:
    (transcript, sk) = ThetaKEM_KeyGen(params)
    ambiguity_estimate = estimate_ambiguity(transcript, params)
    verifica: ambiguity_estimate >= 2^params.lambda
    verifica: Verify(params, transcript, generate_valid_candidate(sk, transcript)
    verifica: Verify(params, transcript, generate_random_candidate(params)) = rej
                (com probabilidade >= 1 - 2^{-lambda})

COMPONENTE 2: Interface de Oráculo Adversarial (ROM-Restrito)
  // O adversário tem acesso APENAS a estes oráculos
```

```
  Oráculo 1: eval_f(transcript, tau) -> FieldElement
    // Avalia a parte holomorfa f em ponto tau
    // Não expõe shadow g nem dados de completação

  Oráculo 2: verify(transcript, candidate) -> VerifyResult
    // Executa verificador completo em tempo constante
    // Retorna aceitar/rejeitar + scores dos operadores

  Oráculo 3: encaps(transcript) -> (Ciphertext, EncapsKey)
    // Executa encapsulamento público (sem acesso a sk)

  Oráculo 4: decaps_challenge(transcript, ct) -> "aceitar" | "rejeitar"
    // Versão limitada do decapsulamento (não retorna K, apenas resultado)
    // O adversário não recebe K do ct desafiado (modelo IND-CCA)

  O adversário NÃO tem acesso a:
    - shadow g
    - dados de rigidificação eta
    - função de completação Comp(T, Y)
    - qualquer dado que requeira integração analítica global

COMPONENTE 3: Módulos de Estratégia de Ataque

  ATAQUE 1: Enumeração Bruta
    for i in 1..2^min(attack_budget, lambda):
      g_candidate = sample_uniform(shadow_space)
      eta_candidate = sample_uniform(eta_space)
      Y_candidate = (g_candidate, eta_candidate)
      // Testa se Y_candidate é o sk verdadeiro via oráculo de decapsulamento
      ct_test = encaps(transcript)
      K_candidate = derive_K_from_Y(Y_candidate, transcript, ct_test)
      // Compara com K real (disponível para quem fez o encaps)
      // Mede fração de acertos

    Resultado esperado: fração de acertos = 1/|KeySpace| = 2^{-lambda}
    Desvio aceitável: < 2 * 2^{-lambda} para implementação correta

  ATAQUE 2: Interpolação Polinomial
    // Tenta reconstruir g a partir de coeficientes do transcript
    // via interpolação de Lagrange ou equivalente

    evals = []
    for i in 1..attack_budget:
      tau_i = choose_point_adaptively(i, evals)
      v_i = eval_f(transcript, tau_i)
      evals.append((tau_i, v_i))
```

```
    g_recovered = lagrange_interpolate(evals, params)

    Resultado esperado: g_recovered != g* para qualquer attack_budget polinomial
    Razão: g não está determinado por nenhuma quantidade polinomial de avaliações
           (propriedade fundamental de não-canonicidade)

  ATAQUE 3: Fingerprinting Estatístico
    // Tenta correlacionar transcripts com shadows via análise estatística

    transcript_features = [extract_features(t) for t in transcripts_dataset]
    shadow_labels = [None] * n_transcripts  // Adversário não conhece shadows

    // Tenta clustering não supervisionado nos features
    clusters = cluster(transcript_features, method="kmeans", k=estimate_k)

    Resultado esperado: clusters não se correlacionam com shadows
    Razão: a função derive_q(g) é one-way e sua saída (que determina os coeficien
           não tem correlação linear detectável com g

  ATAQUE 4: Regressão Linear e Álgebra Linear
    // Tenta expressar g como combinação linear dos coeficientes do transcript

    A = matrix_of_features(transcripts_dataset)  // n_transcripts × n_features
    b = vector_of_shadows(transcripts_dataset)   // n_transcripts × shadow_dim
    // Em ataque real, b é desconhecido; este ataque requer dataset rotulado
    // que o adversário não tem

    Resultado esperado: A^+ * b (pseudoinversa) tem erro residual ~ 1 (aleatório)
    Razão: relação entre g e coeficientes é não-linear (via derive_q e FMT de 3ª

  ATAQUE 5: Ataques de ML Clássico e Quântico Simulado
    // Tenta treinar classificador para distinguir g* das outras completações

    model = train_classifier(transcripts_train, shadows_train)
    accuracy = evaluate(model, transcripts_test, shadows_test)

    Resultado esperado: accuracy ~ 1/|KeySpace| = 2^{-lambda} (equivalente a alea
    Razão: simetria de rótulos — todos os membros da família são igualmente válid
           segundo qualquer observável, portanto o classificador não tem sinal

COMPONENTE 4: Estimador de Largura de Ambiguidade

  estimate_ambiguity(transcript, params) -> (lower_bound, upper_bound, confidence
    // Estima a cardinalidade da família admissível do transcript

    // Método 1: Amostragem Monte Carlo
    n_samples = max(10000, 100 * params.lambda)
```

```
    n_accepted = 0
    for _ in range(n_samples):
      candidate = generate_random_candidate(params)
      if Verify(params, transcript, candidate).accepted:
        n_accepted++

    fraction_accepted = n_accepted / n_samples

    // A fração de candidatos aleatórios aceitos estima |família| / |espaço total
    // |família| ≈ fraction_accepted × |espaço total|
    // |espaço total| ≈ p^N = (2^{2*lambda})^N

    family_size_estimate = fraction_accepted * pow(params.p, params.N)

    // Lower bound conservador (intervalo de confiança 99%)
    lower_bound = family_size_estimate / 10  // Fator de segurança

    Resultado esperado (correto):
      lower_bound >= 2^params.lambda
      fraction_accepted ≈ 2^{lambda - 2*lambda*N} (muito pequena, mas não zero)

    NOTA: Pela construção do transcript via GPMTNC, este lower_bound é garantido
    pelo ATN que verificou distância espectral suficiente.

COMPONENTE 5: Logger Reprodutível
  Todo ataque é logado com:
    - Parâmetros completos (seed + params)
    - Estratégia de ataque e configuração
    - Número de consultas por oráculo
    - Vantagem medida ao longo do tempo
    - Hash do transcript atacado
    - Tempo de CPU e memória utilizados

  Os logs são assinados com Theta-Sign para não-repudiabilidade.

  MÉTRICAS MANDATÓRIAS A REPORTAR:
    1. Tamanho do transcript em bytes
    2. Tempo de KeyGen em ms
    3. Tempo de Encaps em ms
    4. Tempo de Decaps em ms
    5. Tempo de Verify em ms
    6. Estimativa de |família admissível|
    7. Vantagem adversarial medida por estratégia
    8. Curva de vantagem vs. número de consultas de oráculo
```

4.2 Suite de Testes de Corretude

```
TESTES MANDATÓRIOS DE CORRETUDE

TESTE 1: Round-trip KEM
  for level in [Theta1, Theta2, Theta3]:
    params = SystemParams(level)
    (pk, sk) = ThetaKEM_KeyGen(params)
    (ct, K_enc) = ThetaKEM_Encaps(pk)
    K_dec = ThetaKEM_Decaps(sk, pk, ct)
    assert K_enc == K_dec

  // Deve passar com probabilidade 1 (corretude determinística)

TESTE 2: Verificador Aceita Família
  (pk, sk) = ThetaKEM_KeyGen(Theta2)
  for _ in range(1000):
    (ct, _) = ThetaKEM_Encaps(pk)  // Cada Encaps gera candidato diferente
    candidate = Candidate.from_ct(ct)
    assert Verify(params, pk.transcript, candidate).accepted

  // Qualquer texto cifrado gerado pelo encapsulador deve ser aceito

TESTE 3: Verificador Rejeita Candidatos Aleatórios
  (pk, sk) = ThetaKEM_KeyGen(Theta2)
  n_reject = 0
  for _ in range(10000):
    candidate = Candidate.random(params)
    if not Verify(params, pk.transcript, candidate).accepted:
      n_reject++

  rejection_rate = n_reject / 10000
  assert rejection_rate > 1 - 2 * 2^{-params.lambda}
  // Candidatos aleatórios devem ser rejeitados com probabilidade muito alta

TESTE 4: Transcript Hash Inviolável
  (pk, sk) = ThetaKEM_KeyGen(Theta2)
  tampered = pk.transcript.copy()
  tampered.coeffs[0] = tampered.coeffs[0] + 1  // Modifica um coeficiente

  candidate = Candidate.from_transcript(tampered)
  result = Verify(params, tampered, candidate)
  assert not result.accepted  // Deve rejeitar (hash falha)

TESTE 5: VCI Zero-Knowledge (Simulabilidade)
  (pk, sk) = ThetaKEM_KeyGen(Theta2)

  // Transcripts honestos
```

```
  honest_transcripts = [run_vci_protocol(sk, pk) for _ in range(1000)]

  // Transcripts simulados (sem sk)
  simulated_transcripts = [simulate_vci_without_sk(pk) for _ in range(1000)]

  // Verifica indistinguibilidade estatística
  ks_statistic = kolmogorov_smirnov_test(honest_transcripts, simulated_transcript
  assert ks_statistic < 0.05  // Distribuições indistinguíveis

TESTE 6: VCI Soundness (Extração)
  (pk, sk) = ThetaKEM_KeyGen(Theta2)

  // Provador desonesto que conhece pk mas não sk
  // Simula duas execuções com desafios distintos mas mesmo compromisso A
  e1, e2 = distinct_challenges()
  z1, z2 = dishonest_prover_responses(pk, e1, e2)  // Pode usar qualquer estratég

  // Extrator
  sk_extracted = extract_sk(pk, (A, e1, z1), (A, e2, z2))

  // Se provador convenceu V em ambas, extrator deve recuperar sk
  if prover_convinced_V(pk, (A, e1, z1)) and prover_convinced_V(pk, (A, e2, z2)):
    assert sk_extracted == sk
```

PARTE V — CAMINHOS DE IMPLEMENTAÇÃO

5.1 Implementação em Rust

Rust é a linguagem de referência para implementações de produção do sistema. Sua garantia de segurança de memória, ausência de GC (crucial para tempo constante) e ecossistema de criptografia tornam-na a escolha natural.

```
STACK DE DEPENDÊNCIAS RUST

[dependencies]
# Aritmética de campo finito de alta performance
num-bigint = "0.4"          # Big integers para primos arbitrários
num-traits = "0.2"
subtle = "2.4"              # Operações em tempo constante
crypto-bigint = "0.5"       # Alternativa de alta performance para aritmética mod

# Funções de hash e KDF
sha2 = "0.10"               # SHA-256 para transcript_hash
```

```
sha3 = "0.10"               # SHA3-256 para KDF
hkdf = "0.12"               # HKDF para derivação de chave estruturada

# CSPRNG
rand = "0.8"
rand_chacha = "0.3"         # ChaCha20 como CSPRNG de alta velocidade

# Serialização
serde = { version = "1.0", features = ["derive"] }
bincode = "1.3"             # Serialização binária determinística

# NTT para multiplicação polinomial
ntt = "0.1"                 # Number Theoretic Transform (se disponível)
// Se não disponível: implementação custom em O(N log N)

MÓDULOS DA IMPLEMENTAÇÃO RUST:

src/
  lib.rs                    # Interface pública
  params.rs                 # SystemParams, constantes, primos seguros
  field.rs                  # Aritmética em Z_p em tempo constante
  polynomial.rs             # Operações polinomiais (NTT, avaliação, Horner)
  transcript.rs             # Transcript, serialização, hashing
  gpmtnc.rs                 # Gerador de parâmetros (GPMTNC + ATN)
  operators/
    mod.rs                  # Engine de operadores, Verify()
    op_gamma.rs             # Operador de transporte modular
    op_cusp.rs              # Operador de stress em cúspides
    op_conv.rs              # Operador de convolução
    op_spec.rs              # Operador de regularidade espectral
    op_res.rs               # Operador de compatibilidade residual
  kem.rs                    # ThetaKEM (KeyGen, Encaps, Decaps)
  sign.rs                   # ThetaSign (Sign, Verify)
  commit.rs                 # MT-Com (Commit, Open)
  prf.rs                    # MT-PRF (Eval)
  vci.rs                    # VCI-MTNCC Protocol
  mtdv.rs                   # Módulo de Transcrições Verificáveis
  harness/
    mod.rs                  # Harness de ataque
    attacks.rs              # Módulos de estratégia de ataque
    estimator.rs            # Estimador de largura de ambiguidade
    logger.rs               # Logger reprodutível

ESTIMATIVAS DE LINHAS DE CÓDIGO:
  Camadas 1-2 (aritmética): ~500 LOC
  Camada 3 (transcript + GPMTNC): ~800 LOC
  Camada 4 (operadores): ~1.200 LOC
```

```
Camada 5 (primitivas): ~1.000 LOC
Camada 6 (protocolos): ~800 LOC
Harness: ~600 LOC
Testes: ~1.000 LOC
TOTAL: ~5.900 LOC (implementação funcional, sem otimizações avançadas)
```

5.2 Implementação em FPGA

A implementação em FPGA é a mais relevante para aplicações de infraestrutura crítica (HSM, gateways de segurança, sistemas embarcados de alta segurança).

```
ARQUITETURA FPGA — THETA-KEM THETA-2

TARGET: Xilinx Ultrascale+ (ex: XCVU9P-L2FSGD2104E) ou Intel Stratix 10

UNIDADES FUNCIONAIS PRINCIPAIS:

1. Unidade de Aritmética de Campo (FAU — Field Arithmetic Unit)
   Função: operações em Z_p (add, mul, inv, pow)
   Implementação: Montgomery multiplication (512-bit)
   Latência: add=2 ciclos, mul=20 ciclos, inv=1024 ciclos (via Fermat)
   Área estimada: ~5.000 LUT + ~2.000 FF + 16 DSPs

2. Engine de NTT (para multiplicação polinomial)
   Função: NTT forward/inverse sobre Z_p
   Implementação: Radix-4 NTT, pipeline de 4 estágios
   Latência: N=1024 → ~10*log2(1024) = 100 ciclos por NTT
   Throughput: 1 NTT a cada 128 ciclos (pipeline)
   Área estimada: ~20.000 LUT + ~8.000 FF + 64 DSPs

3. Scheduler de Operadores
   Função: controla sequência de execução dos 5 operadores
   Implementação: FSM com ordem fixa (sem branching dependente de dados)
   Latência: total de Op1+...+Op5 em pipeline ≈ 10.000 ciclos
   Área estimada: ~2.000 LUT + ~1.000 FF

4. Controlador de Verificação
   Função: agrega scores, decide aceitar/rejeitar
   Implementação: comparadores de ponto fixo, acumulador de score
   Latência: ~20 ciclos adicionais
   Área estimada: ~500 LUT + ~200 FF

5. Interface de Memória
   Função: armazenamento do transcript (73,8KB para Theta-2)
   Implementação: BRAM integrada na FPGA (disponível em todos os FPGAs modernos)
   Capacidade necessária: 73,8KB / 36Kb por BRAM = ~17 BRAMs
```

```
   Área: 17 BRAMs

ESTIMATIVAS DE PERFORMANCE FPGA (Theta-2, @300 MHz):

Operação       | Ciclos   | Latência @300MHz | Throughput
---------------|----------|------------------|----------
KeyGen          | ~500.000 | ~1,67 ms         | ~600 KeyGen/s
Encaps          | ~250.000 | ~0,83 ms         | ~1.200 Encaps/s
Decaps (sucesso)| ~350.000 | ~1,17 ms         | ~855 Decaps/s
Verify          | ~200.000 | ~0,67 ms         | ~1.500 Verify/s
VCI 3-rodadas   | ~150.000 | ~0,50 ms/rodada  | ~660 sessões/s

COMPARAÇÃO COM KYBER-768 EM FPGA (@300MHz):
Kyber-768 KeyGen: ~10.000 ciclos → ~0,033 ms → 30.000 KeyGen/s
Kyber-768 Encaps: ~15.000 ciclos → ~0,050 ms → 20.000 Encaps/s

// Theta-2 é ~50x mais lento que Kyber-768 em FPGA para KEM
// O trade-off é deliberado e documentado: Theta oferece Regime III de segurança,
// proteção categorial ao HNDL, e diversificação ontológica —
// propriedades que Kyber não possui e que não têm custo equivalente.
// Para aplicações de infraestrutura crítica com ciclo de vida de 15-30 anos,
// o overhead de performance é pequeno em relação ao ganho de segurança.

ÁREA TOTAL ESTIMADA (Theta-2 KEM completo):
  LUT: ~30.000 (3% de XCVU9P)
  FF:  ~12.000 (1% de XCVU9P)
  DSP: ~80 (3% de XCVU9P)
  BRAM: ~20 (1% de XCVU9P)
  → Sistema altamente compacto; múltiplas instâncias paralelas possíveis
```

PARTE VI — ANÁLISE FORMAL DA PROPRIEDADE DE AUSÊNCIA CATEGORIAL

Esta seção aborda diretamente o ponto técnico-jurídico mais delicado do pedido: a demonstração formal e operacionalmente verificável de que "G é categoricamente ausente de T".

6.1 Formulação Operacional da Ausência Categorial

A afirmação "G é categoricamente ausente de T" é demonstrável via inspeção da função de geração GPMTNC. A demonstração segue:

Teorema de Separação (informal, para fins de habilitação):

Dado o algoritmo GPMTNC especificado, para qualquer adversário A que opere exclusivamente sobre T (e oráculos ROM), a vantagem de A em distinguir ou recuperar G = (g*, η) é no máximo negligenciável em λ.

Demonstração por análise estrutural do GPMTNC:

A função GPMTNC opera em dois modos para os dados de saída:

Modo T (transcript, dados públicos): Os coeficientes {a_0,...,a_{N-1}} são computados como a_n = q^{n²} / ((-q;q)_n)² mod p, onde q = derive_q(g, params). A função derive_q é definida como: q = H_to_field(g || params || "q_derivation"), onde H_to_field é uma função hash com saída em Z_p. Esta função é pré-imagem resistente: dado q, recuperar g requer resolver pré-imagem de H — problema com complexidade 2^lambda no modelo de oráculo aleatório.

Modo Y (dados privados): O shadow g e os dados η são gerados de forma totalmente independente dos coeficientes a_n. O shadow g é amostrado do espaço de shadows antes que qualquer coeficiente seja computado. O único "link" entre g e {a_n} passa exclusivamente por q = derive_q(g), que é irreversível.

Consequência: Qualquer função que compute g a partir de {a_n} deve necessariamente inverter derive_q — o que é equivalente a encontrar pré-imagem de H_to_field. Isso é computacionalmente intratável para adversários polinomialmente limitados.

Verificabilidade operacional: Esta separação é verificável por inspeção de código. Em qualquer implementação correta:

- g não aparece em nenhuma linha de código que escreva para transcript
- derive_q é marcada como função one-way por convenção de implementação
- Testes de harness verificam que nenhum campo de transcript tem correlação linear detectável com g

6.2 Acoplamento com o Verificador

O Verify() demonstra operacionalmente que a ausência de G não impede a verificação:

```
DEMONSTRAÇÃO: Verify() não usa G

Entrada: (params, transcript, candidate)
// NOTA: G não é um parâmetro de Verify() — não está nem disponível

Op_gamma não usa G: opera sobre transcript e candidate
Op_cusp não usa G: opera sobre transcript e candidate
```

```
Op_conv não usa G: opera sobre transcript e candidate
Op_spec não usa G: opera sobre transcript e candidate
Op_res não usa G: opera sobre transcript e candidate

Saída: VerifyResult (aceitar/rejeitar)

// QED: Verify() aceita candidatos válidos sem nenhum acesso a G.
// A ausência de G em Verify() é verificável por inspeção da assinatura de função
// Qualquer examinador técnico pode verificar que G não é parâmetro nem variável
```

Esta demonstração responde diretamente ao risco jurídico identificado: a ausência categorial de G em T é verificável porque é equivalente a verificar que G não é parâmetro de Verify() — uma propriedade sintática do código, não uma afirmação filosófica.

APÊNDICE A — PRIMOS SEGUROS POR NÍVEL

```
NÍVEL THETA-1 (lambda=80, p de 256 bits):
  p = 2^255 - 19  // Primo de Curve25519, disponível e auditado
  // Alternativa: qualquer primo de 256 bits com p ≡ 3 (mod 4)

NÍVEL THETA-2 (lambda=128, p de 512 bits):
  p = 2^521 - 1   // Primo de Mersenne M521, eficiente para redução
  // Alternativa: qualquer primo de 512 bits ≡ 3 (mod 4) com r=(p-1)/2 primo

NÍVEL THETA-3 (lambda=256, p de 1024 bits):
  p = gerado via FIPS 186-5 DSA prime generation
  // Requisito: p = 2*q + 1 com q primo (primo seguro de Sophie Germain)
  // p deve ter 1024 bits; geração por Miller-Rabin com 128 testemunhas

NOTA: Os primos específicos são parametrizáveis. A segurança do sistema não
depende de propriedades especiais de primos além dos requisitos padrão de
criptografia de campo finito. A presente especificação usa primos auditados
e amplamente utilizados para facilitar revisão e implementação.
```

APÊNDICE B — GLOSSÁRIO DE IMPLEMENTAÇÃO

Termo	Definição no Contexto de Implementação
`FieldElement`	**Elemento de Z_p, representado como byte array little-endian de ceil(log2(p)/8) bytes**

`shadow_space`	**Espaço de polinômios de grau N-1 sobre Z_p, amostrado uniformemente**
`eta_space`	**Espaço de bytes32, amostrado uniformemente; representa dados de rigidificação**
`family_admissível`	**Conjunto de candidatos X tal que Verify(params, T, X) = aceitar**
`candidato_válido`	**Elemento da família admissível; gerado por Encaps ou por qualquer Y′**
`candidato_verdadeiro`	**Candidato gerado especificamente com Y = Y*; único que permite Decaps correto**
`time_constant`	**Propriedade de implementação: tempo de execução não depende de valores de dados**
`NTT`	**Number Theoretic Transform; equivalente à FFT sobre Z_p; complexidade O(N log N)**
