Contents

1 Appendix B: Zero-Knowledge Proof Systems - Technical Implementation

1.1 B.1 Cryptographic Foundations

1.1.1 B.1.1 Proof System Definitions

Definition B.1 (Zero-Knowledge Proof System)

A zero-knowledge proof system for an NP language L consists of three probabilistic polynomial-time algorithms (Setup, Prove, Verify):

- **Setup(1^ , C) → (pp, vk)**: Generates public parameters pp and verification key vk for circuit C
- **Prove(pp, x, w) →** : Generates proof that (x, w) satisfies circuit C
- **Verify(vk, x,) → {0,1}**: Verifies proof for public input x

Properties: 1. **Completeness**: For all valid (x, w), Verify accepts with probability 1 2. **Soundness**: No adversary can create accepting proof for invalid x 3. **Zero-Knowledge**: Proof reveals nothing beyond truth of statement

1.1.2 B.1.2 SNARKs (Succinct Non-Interactive Arguments of Knowledge)

Succinctness: | | = poly(, log|C|), independent of witness size **Non-Interactive**: Single message from prover to verifier **Argument**: Computational soundness (not information-theoretic) **of**

Knowledge: Extractor can recover witness from accepting proof

1.2 B.2 Backend Implementations

1.2.1 B.2.1 Groth16 - Optimal Performance

Cryptographic Construction:

Curve: BLS12-381 - **Base field**: _q where q 2^381 - **Scalar field**: _r where r 2^255 - **Embedding degree**: k = 12 - **Security**: 128-bit

Proof Structure:

```
 = ( _A  G , _B  G , _C  G )
```

Verification Equation:

```
e( _A, _B) = e( , ) · e(L, ) · e( _C, )
```

where: - e: G × G → G_T is optimal Ate pairing - , , , : Setup parameters from trusted ceremony - L: Linear combination of public inputs

Proof Size Breakdown:

```
 _A: 48 bytes (compressed G  point)
 _B: 96 bytes (compressed G  point)
 _C: 48 bytes (compressed G  point)
Total: 192 bytes
```

Trusted Setup Protocol:

Phase 1 (Powers of Tau):

```
 , ², ³, ..., ^N  in G , G
```

- Universal: Reusable across all circuits up to size N
- Participants: 1000+ in Perpetual Powers of Tau ceremony
- Security: Need only 1 honest participant

Phase 2 (Circuit-Specific):

```
 , , , , { _i}  for specific circuit
```

- Circuit-specific: Must rerun for circuit modifications
- Multi-party computation: N participants contribute randomness
- Security: 1-of-N honesty assumption

Key Compromise Response:

Indicators of compromise: 1. Invalid proofs that verify 2. Leaked ceremony transcripts 3. Participant compromise acknowledgment

Immediate response (T < 1 hour):

```
# 1. Disable affected circuits
genomevault zk disable-circuit variant_presence --reason "key_compromise"

# 2. Alert downstream systems
```

```bash
genomevault alerts broadcast --level critical --msg "ZK key compromise detected"

# 3. Queue re-generation
genomevault zk queue-regenall --start-after-ceremony
```

Recovery procedure (T = 24-48 hours):

```bash
# 1. New ceremony with vetted participants
snarkjs groth16 setup circuit.r1cs pot28_final.ptau circuit_0000.zkey

# 2-N. Participants contribute
for i in {1..10}; do
    snarkjs zkey contribute circuit_$(($i-1)).zkey circuit_$i.zkey \
        --name "Emergency Contributor $i" \
        --entropy $(openssl rand -hex 32)
done

# Final beacon
snarkjs zkey beacon circuit_10.zkey circuit_final.zkey $(openssl rand -hex 32) 10

# 3. Export new verification key
snarkjs zkey export verificationkey circuit_final.zkey vk_new.json

# 4. Update production
genomevault zk update-vk variant_presence vk_new.json --verify-ceremony
```

1.2.2 B.2.2 PLONK - Universal Setup

Cryptographic Construction:

Polynomial Commitment: KZG (Kate-Zaverucha-Goldberg)

Commit to polynomial f(X) of degree d:

```
C = [f( )]  = f( )·G
```

where   is secret from trusted setup.

Opening Proof: For claimed evaluation f(z) = y:

```
  = [(f(X) - y)/(X - z)]
```

Verification:

```
e(C - [y] , G ) = e( , [ ]  - [z] )
```

Universal SRS (Structured Reference String):

```
{[ ] , [ ]  : i = 0...N}
```

Advantages: - **Reusable**: All circuits up to size N - **Updatable**: Additional ceremonies extend without invalidating - **Available**: Aztec's Ignition ceremony provides SRS up to 2^{2} constraints

Proof Composition:

```
  = (
    a_comm, b_comm, c_comm,     # Wire commitments
    z_comm,                      # Permutation accumulator
    t_lo, t_mid, t_hi,          # Quotient polynomial (chunked)
    a_eval, b_eval, c_eval,      # Wire evaluations
    s_ 1_eval, s_ 2_eval,       # Permutation evaluations
    z_omega_eval,                # Next accumulator eval
    opening_proof,               # Batch opening
    challenge_response           # Fiat-Shamir transcript
)
```

Proof Size: Approximately 1KB (7 G points + 7 scalars)

1.2.3 B.2.3 Halo2 - Trustless Construction

Innovation: Eliminates trusted setup via IPA (Inner Product Argument)

Polynomial Commitment (IPA-based):

Commit to polynomial f(X) with coefficients f = (f , f , …, f_d):

```
C =  f, G  = Σ f ·G
```

where G is deterministic from hash function (no trusted setup).

Opening Proof: Recursive halving protocol

To prove f(z) = y:

```
1. Split f = (f_L, f_R), G = (G_L, G_R)
2. Compute cross terms L, R
3. Verifier sends challenge x
4. Recurse on f' = f_L + x·f_R, G' = G_L + x ¹·G_R
5. After log(d) rounds, verify base case
```

Proof Size: O(log d) group elements   5KB for d = 2^2

Pasta Curves (Pallas/Vesta):

Purpose: Enable efficient recursion without pairing-friendly curves

- **Pallas**: Base field _p, scalar field _q
- **Vesta**: Base field _q, scalar field _p

Cycle property: p's order = q, q's order = p

This allows: - Prove Vesta circuit in Pallas - Prove Pallas circuit in Vesta - Compose recursively without field switching overhead

1.3 B.3 Circuit Implementation

1.3.1 B.3.1 Variant Presence Circuit

Circuit Logic:

```
template VariantPresence(numVariants) {
    // Private inputs
    signal input variants[numVariants];     // Patient's variants
    signal input queryVariant;               // Variant to check

    // Public output
    signal output hasVariant;

    // Intermediate signals
    signal isMatch[numVariants];
    signal accumulator[numVariants];

    // Check each variant
    accumulator[0] <== 0;
    for (var i = 0; i < numVariants; i++) {
        isMatch[i] <== IsEqual()([variants[i], queryVariant]);
        if (i > 0) {
            accumulator[i] <== accumulator[i-1] + isMatch[i];
        }
    }

    // Output 1 if any match found, 0 otherwise
    hasVariant <== GreaterThan(32)([accumulator[numVariants-1], 0]);
}
```

Constraint Count Analysis: - IsEqual: 2 constraints per comparison - GreaterThan: 252 constraints (32-bit comparison) - Total: 2N + 252 15,234 for N=7,500 variants

1.3.2 B.3.2 Polygenic Risk Score Circuit

Circuit Logic:

```
template PolygeneticRisk(numSNPs) {
    // Private inputs
    signal input snps[numSNPs];           // Binary SNP values {0,1,2}
    signal input weights[numSNPs];        // Risk weights (fixed-point)
    signal input salt;                     // Privacy salt

    // Public inputs
    signal input threshold;                // Risk threshold
    signal input commitment;               // Hash commitment to SNPs

    // Public output
    signal output isHighRisk;

    // Constraint: Verify SNP commitment
    component hasher = Poseidon(numSNPs + 1);
    for (var i = 0; i < numSNPs; i++) {
        hasher.inputs[i] <== snps[i];
```

```
    }
    hasher.inputs[numSNPs] <== salt;
    hasher.out === commitment;

    // Compute weighted risk score
    signal partialSums[numSNPs];
    partialSums[0] <== snps[0] * weights[0];
    for (var i = 1; i < numSNPs; i++) {
        partialSums[i] <== partialSums[i-1] + snps[i] * weights[i];
    }
    signal riskScore <== partialSums[numSNPs - 1];

    // Check if risk exceeds threshold
    isHighRisk <== GreaterThan(32)([riskScore, threshold]);
}
```

Constraint Count: O(numSNPs)   1M for comprehensive PRS

1.3.3 B.3.3 Ancestry Estimation Circuit

Circuit Logic: Principal component analysis on genetic markers

```
template AncestryProof(numMarkers, numPCs) {
    // Private inputs
    signal input markers[numMarkers];      // Ancestry-informative markers
    signal input pcWeights[numMarkers][numPCs];  // PCA weights

    // Public outputs
    signal output ancestry;                 // Ancestry category {0,1,2,...}

    // Compute principal components
    signal pcs[numPCs];
    for (var pc = 0; pc < numPCs; pc++) {
        signal partialSum[numMarkers];
        partialSum[0] <== markers[0] * pcWeights[0][pc];
        for (var i = 1; i < numMarkers; i++) {
            partialSum[i] <== partialSum[i-1] + markers[i] * pcWeights[i][pc];
        }
        pcs[pc] <== partialSum[numMarkers - 1];
    }

    // Classification decision tree (simplified)
    signal isEuropean <== GreaterThan(32)([pcs[0], threshold_eur]);
    signal isAfrican <== GreaterThan(32)([pcs[1], threshold_afr]);
    // ... additional logic for multi-way classification

    ancestry <== 0 + isEuropean + 2*isAfrican + ...;
}
```

1.4 B.4 Performance Optimization

1.4.1 B.4.1 Batch Proving

Process multiple proofs in parallel:

```
from genomevault.zk_proofs import BatchProver
from concurrent.futures import ThreadPoolExecutor

prover = BatchProver(backend="halo2", workers=10)

# Queue proofs
proof_ids = []
for i, (public, private) in enumerate(inputs):
    proof_id = prover.queue_proof(
        circuit="variant_presence",
        public_inputs=public,
        private_inputs=private
    )
    proof_ids.append(proof_id)

# Wait for completion
proofs = prover.wait_all(proof_ids, timeout=300)

print(f"Generated {len(proofs)} proofs in batch")
```

Throughput: - Serial: 1.67 proofs/core/sec - Parallel (10 cores): 16.7 proofs/sec - With caching (40% hit rate): 27.8 proofs/sec effective

1.4.2 B.4.2 Proof Caching

Cache proofs for common queries:

```
from genomevault.zk_proofs import ProofCache

cache = ProofCache(
    backend="redis",
    ttl=86400,  # 24 hours
    max_size="10GB"
)

# Check cache before proving
cache_key = hash((circuit, public_inputs))
cached_proof = cache.get(cache_key)

if cached_proof:
    return cached_proof
else:
    proof = prover.prove(circuit, public_inputs, private_inputs)
    cache.set(cache_key, proof)
```

```
    return proof
```

Cache Hit Rates (measured in production): - Variant presence queries: 42% hit rate - PRS calculations: 18% hit rate (more varied) - Ancestry checks: 65% hit rate (limited ancestry groups)

1.5 B.5 Security Analysis

1.5.1 B.5.1 Soundness Analysis

Groth16 Soundness Error:

```
_soundness = 1 / | _r|    2^(-255)   10^(-77)
```

PLONK Soundness Error:

```
_soundness = (d + 6) / | |    10^(-75) for d = 10
```

Halo2 Soundness Error:

```
_soundness = O(d / | _p|)    10^(-74) for d = 10
```

All provide negligible soundness error (« 2^(-128) security).

1.5.2 B.5.2 Zero-Knowledge Analysis

Groth16 ZK Simulator:

Given verification key vk and statement x, simulator can produce indistinguishable proofs without witness:

```python
def simulate_proof(vk, x):
    # Sample random group elements
     _A = random_G1()
     _B = random_G2()

    # Compute  _C to satisfy verification equation
     _C = compute_valid_C(vk, x,  _A,  _B)

    return ( _A,  _B,  _C)
```

Zero-Knowledge Property: No polynomial-time distinguisher can tell real proofs from simulated proofs with advantage > .

1.6 B.6 Production Monitoring

1.6.1 B.6.1 Key Metrics

Proving Metrics:

```yaml
proof_generation_time_ms:
  p50: 603
  p95: 711
  p99: 711

peak_memory_usage_mb:
```

```yaml
  p50: 4200
  p95: 4350
  p99: 4400

proof_queue_depth:
  threshold: 100
  alert: depth > 100 for 5 minutes
```

Verification Metrics:

```yaml
proof_verification_time_ms:
  p50: 20.4
  p95: 23.1
  p99: 23.2

verification_failure_rate:
  threshold: 0.001  # 0.1%
  alert: rate > 0.001 over 1 hour
```

1.6.2 B.6.2 Alerting Rules

```yaml
alerts:
  - name: ProvingTimeAnomaly
    condition: proof_generation_time_ms.p95 > 1000
    severity: warning
    action: Scale up prover pool

  - name: VerificationFailure
    condition: verification_failure_rate > 0.001
    severity: critical
    action: Investigate key compromise, disable circuit

  - name: QueueBacklog
    condition: proof_queue_depth > 100
    severity: warning
    action: Add prover workers

  - name: MemoryExhaustion
    condition: peak_memory_usage_mb > 60000
    severity: critical
    action: Restart prover, upgrade instance
```

1.7 B.7 References

1. Groth, J. (2016). "On the size of pairing-based non-interactive arguments." In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 305-326).

2. Gabizon, A., Williamson, Z. J., & Ciobotaru, O. (2019). "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge." *IACR ePrint*

Archive.

3. Bowe, S., Grigg, J., & Hopwood, D. (2020). "Recursive Proof Composition without a Trusted Setup." *IACR ePrint Archive.*
4. Kate, A., Zaverucha, G. M., & Goldberg, I. (2010). "Constant-size commitments to polynomials and their applications." In *International Conference on the Theory and Application of Cryptology and Information Security* (pp. 177-194).

Implementation: Complete circuits available in `zk_circuits/` directory with compilation instructions and test vectors.