Vision Document: Dimensional Game Theory Integration with Quantum Threat Modeling

Version 1.0 | September 2, 2025

Author: Nnamdi Okpala (OBINexus)

Toolchain: riftlang.exe → .so.a → rift.exe → gosilang

Executive Summary

This vision document outlines the integration of Dimensional Game Theory (DGT) with Quantum Field Theory (QFT) threat modeling, creating a unified framework for analyzing side-channel attacks, multi-stage threats, and real-time mitigation through Node Zero's ZKP system. By treating attack vectors as coherent mathematical objects in a variadic dimensional space, we achieve provable safety guarantees and computational tractability.

1. Theoretical Foundation

1.1 Dimensional Game Theory Mapping

In DGT, we define a game as:

```
G = (N, A, u, D)
```

where:

- **N**: Set of agents (threats, defenders, systems)
- **A**: Action space (exploit attempts, mitigations)
- **u**: Utility functions (attack success probability)
- **D**: Active dimensions based on threat context

1.2 Quantum Threat Integration

The threat amplitude in dimensional space becomes:

```
M_DGT(threat) = ∑_i δ(x_i, D_i) × M_Feynman(i)
```

where $\delta(x_i, D_i)$ is the dimensional activation function.

1.3 Side-Channel Attack Formalism

Side-channel attacks are modeled as:

```
|SC⟩ = ∫ dω Ψ_leak(ω) |ω⟩ ⊗ |target⟩
```

where Ψ_leak(ω) represents the leakage spectrum.

2. Extended Threat Scenarios

2.1 Quantum Side-Channel Attack

Scenario: Power analysis of quantum gate operations

Feynman Diagram:

```
Q_gate → [Power_leak] → Classical_observer
      ↓
  [Decoherence]
      ↓
  State_collapse
```

Amplitude Calculation:

```
M_QSC = g_leak × ⟨ω|H_quantum|ω⟩ × exp(-λt) × S
```

Dimensional Analysis:

- D1: Power consumption axis
- D2: Quantum state fidelity
- D3: Time correlation
- Only activate D1,D2 if correlation > threshold

2.2 Multi-Stage Supply Chain Attack

Scenario: NPM → Docker → Kubernetes → Production

Multi-vertex Diagram:

```
Developer → NPM_registry → Docker_build → K8s_deploy → Production
   ↓         ↓             ↓            ↓           ↓
[Commit]  [Package]     [Image]      [Pod]       [RCE]
   ↓         ↓             ↓            ↓           ↓
  G_x=?     G_x=?         G_y=?        G_z=?       S=?
```

Cascading Amplitude:

```
M_cascade = ∏_{i=1}^n g_i × P_i × S_i
```

where each stage has its own coupling and safety function.

2.3 RAF (Regulation As Firmware) Attack

Scenario: Firmware-level persistence with AuraSeal bypass

Diagram:

```
Firmware_vuln → RAF_layer → AuraSeal_check
     ↓           ↓           ↓
  [Persist]   [Bypass]    [Validate]
     ↓           ↓           ↓
    G_x=0      G_y=0       G_z=?
```

Truth Weight Calculation:

```
TW = ∑_i w_i × verified_i ≥ 3.0 (safety threshold)
```

2.4 Coherence Manipulation Attack

Scenario: Exploiting quantum coherence for unauthorized computation

Advanced Diagram:

```
|Ψ_coherent⟩ → [Manipulation] → |Ψ_exploit⟩
     ↓              ↓              ↓
[Entangle]     [Measure]      [Extract]
     ↓              ↓              ↓
  Node Zero     Challenge       Proof
```

3. Implementation Architecture

3.1 Core Components

```rust
```

```
// riftlang.exe - Threat Propagator Generator
pub struct ThreatPropagator {
    mass: f64,          // Threat complexity
    momentum: Vector3,  // Attack direction
    coupling: f64,      // System susceptibility
    dimensions: Vec<Dimension>,
}

// .so.a - Lattice Verification Library
pub struct LatticeVerifier {
    gates: [Gate; 3],   // Gx, Gy, Gz
    threshold: f64,     // Safety threshold
    zkp_state: NodeZeroState,
}

// rift.exe - Policy Enforcer
pub struct PolicyEngine {
    rules: HashMap<ThreatType, Policy>,
    dimensional_constraints: DGTConstraints,
    truth_weight: f64,
}

// gosilang - Polyglot Coordinator
type ThreatAnalyzer interface {
    CalculateAmplitude(threat Threat) Amplitude
    VerifyDimensions(game Game) bool
    CoordinateResponse(agents []Agent) Response
}
```

3.2 Amplitude Calculation Engine

```
python
```

```python
# Python implementation for rapid prototyping
import numpy as np
from typing import List, Tuple, Dict

class QuantumThreatAmplitude:
    def __init__(self, coupling_constants: Dict[str, float]):
        self.g_sys = coupling_constants.get('system', 0.1)
        self.g_leak = coupling_constants.get('leakage', 0.05)
        self.g_quantum = coupling_constants.get('quantum', 0.3)

    def calculate_propagator(self,
                             momentum: np.ndarray,
                             mass: float,
                             prop_type: str) -> complex:
        """Calculate Feynman propagator for threat/vulnerability"""
        p_squared = np.dot(momentum, momentum)

        if prop_type == 'threat':
            return 1j / (p_squared - mass**2 + 1e-10j)
        elif prop_type == 'vulnerability':
            delta_v = self._persistence_factor(momentum)
            return 1j * delta_v / (p_squared - mass**2 + 1e-10j)
        elif prop_type == 'quantum':
            eta = np.eye(4)  # Minkowski metric
            return 1j * eta / p_squared
        else:  # information
            return 1j / (p_squared - mass**2)

    def calculate_vertex(self,
                         incoming: List[Tuple[str, np.ndarray]],
                         safety: float) -> complex:
        """Calculate interaction vertex amplitude"""
        vertex_factor = 1j * self.g_sys * safety

        # Apply dimensional game theory constraints
        if len(incoming) > 3:  # Dimensional reduction
            vertex_factor *= self._dimensional_reduction(incoming)

        return vertex_factor

    def _persistence_factor(self, momentum: np.ndarray) -> float:
        """Vulnerability persistence based on momentum"""
        return np.exp(-np.linalg.norm(momentum) / 10.0)

    def _dimensional_reduction(self, particles: List) -> float:
        """Apply DGT dimensional reduction"""
```

```
        active_dims = len([p for p in particles if p[0] != 'virtual'])
        return 1.0 / (1.0 + active_dims - 3)

class SideChannelAnalyzer(QuantumThreatAmplitude):
    def __init__(self, coupling_constants: Dict[str, float]):
        super().__init__(coupling_constants)
        self.decoherence_rate = 0.01  # 1/s

    def quantum_side_channel_amplitude(self,
                                       leak_spectrum: np.ndarray,
                                       time: float,
                                       gates: Tuple[float, float, float]) -> float:
        """Calculate amplitude for quantum side-channel attack"""
        # Leakage coupling
        leak_amp = self.g_leak * np.sum(leak_spectrum)

        # Decoherence factor
        decoherence = np.exp(-self.decoherence_rate * time)

        # Safety function
        safety = gates[0] * gates[1] * gates[2]

        return leak_amp * decoherence * safety
```

3.3 Node Zero Real-Time Integration

typescript

```typescript
// Node Zero ZKP integration for real-time gate updates
import { z0 } from '@obinexus/node-zero';

interface GateState {
  x: number;  // Software QA
  y: number;  // Quantum integration
  z: number;  // Blockchain verification
}

class NodeZeroGateController {
  private gates: GateState = { x: 0, y: 0, z: 0 };
  private threatMonitor: ThreatMonitor;

  async initializeVerification(): Promise<void> {
    // Create lattice-based identities
    await z0.create('system_identity.json');
    await z0.create('verifier_identity.json');

    // Start continuous verification loop
    this.startVerificationLoop();
  }

  private async startVerificationLoop(): Promise<void> {
    setInterval(async () => {
      const threats = this.threatMonitor.getActiveThreats();

      for (const threat of threats) {
        // Generate challenge for each axis
        const challenge = await z0.challenge({
          from: 'verifier_identity.json',
          to: 'system_identity.json',
          threat_context: threat
        });

        // System generates proof
        const proof = await z0.proof({
          challenge: challenge.data,
          identity: 'system_identity.json'
        });

        // Verify and update gates
        const result = await z0.verify({
          proof: proof.data,
          challenger: 'verifier_identity.json'
        });
```

```
            this.updateGates(threat.axis, result.valid);
          }
        }, 1000); // Check every second
      }

      private updateGates(axis: string, verified: boolean): void {
        const newValue = verified ? 1 : 0;

        switch(axis) {
          case 'X':
            this.gates.x = newValue;
            console.log(`QA Gate (Gx): ${newValue}`);
            break;
          case 'Y':
            this.gates.y = newValue;
            console.log(`Quantum Gate (Gy): ${newValue}`);
            break;
          case 'Z':
            this.gates.z = newValue;
            console.log(`Blockchain Gate (Gz): ${newValue}`);
            break;
        }

        // Calculate safety function
        const safety = this.gates.x * this.gates.y * this.gates.z;
        if (safety === 0) {
          console.warn('SYSTEM IN FAIL-SAFE MODE');
          this.triggerFailSafe();
        }
      }

      private triggerFailSafe(): void {
        // Implement fail-safe protocol
        // All operations blocked until gates restored
      }
    }
```

4. Visualization Tool Architecture

4.1 Feynman Diagram Generator

```javascript
```

```javascript
// D3.js-based visualization for threat diagrams
class FeynmanThreatVisualizer {
  constructor(containerId) {
    this.svg = d3.select(containerId)
      .append('svg')
      .attr('width', 800)
      .attr('height', 600);

    this.simulation = d3.forceSimulation()
      .force('link', d3.forceLink().id(d => d.id))
      .force('charge', d3.forceManyBody().strength(-300))
      .force('center', d3.forceCenter(400, 300));
  }

  renderThreatDiagram(threatData) {
    // Clear previous diagram
    this.svg.selectAll('*').remove();

    // Define arrow markers for different line types
    this.defineMarkers();

    // Render nodes (threats, vulnerabilities, systems)
    const nodes = this.svg.selectAll('.node')
      .data(threatData.nodes)
      .enter().append('g')
      .attr('class', d => `node ${d.type}`);

    nodes.append('circle')
      .attr('r', d => d.type === 'threat' ? 20 : 15)
      .attr('fill', d => this.getNodeColor(d.type));

    nodes.append('text')
      .text(d => d.label)
      .attr('text-anchor', 'middle')
      .attr('dy', '.35em');

    // Render edges (propagators)
    const edges = this.svg.selectAll('.edge')
      .data(threatData.edges)
      .enter().append('path')
      .attr('class', d => `edge ${d.type}`)
      .attr('stroke', d => this.getEdgeColor(d.type))
      .attr('stroke-dasharray', d => this.getDashArray(d.type))
      .attr('marker-end', d => `url(#arrow-${d.type})`);

    // Apply force simulation
```

```
    this.simulation
      .nodes(threatData.nodes)
      .on('tick', () => this.tick(nodes, edges));

    this.simulation.force('link')
      .links(threatData.edges);
  }

  defineMarkers() {
    const defs = this.svg.append('defs');

    // Define arrow markers for different propagator types
    ['threat', 'vulnerability', 'quantum', 'information'].forEach(type => {
      defs.append('marker')
        .attr('id', `arrow-${type}`)
        .attr('viewBox', '0 -5 10 10')
        .attr('refX', 15)
        .attr('refY', 0)
        .attr('markerWidth', 6)
        .attr('markerHeight', 6)
        .attr('orient', 'auto')
        .append('path')
        .attr('d', 'M0,-5L10,0L0,5')
        .attr('fill', this.getEdgeColor(type));
    });
  }

  getNodeColor(type) {
    const colors = {
      'threat': '#dc3545',
      'vulnerability': '#007bff',
      'system': '#28a745',
      'quantum': '#6f42c1',
      'zkp': '#17a2b8'
    };
    return colors[type] || '#6c757d';
  }

  getEdgeColor(type) {
    const colors = {
      'threat': '#dc3545',
      'vulnerability': '#007bff',
      'quantum': '#6f42c1',
      'information': '#ffc107',
      'zkp': '#17a2b8'
    };
    return colors[type] || '#000000';
```

```javascript
    }

    getDashArray(type) {
        const patterns = {
            'threat': '5,5',      // Dashed
            'vulnerability': '0', // Solid
            'quantum': '2,2',     // Dotted
            'zkp': '10,5,2,5'    // Dash-dot
        };
        return patterns[type] || '0';
    }

    tick(nodes, edges) {
        nodes.attr('transform', d => `translate(${d.x},${d.y})`);

        edges.attr('d', d => {
            const dx = d.target.x - d.source.x;
            const dy = d.target.y - d.source.y;
            const dr = Math.sqrt(dx * dx + dy * dy);

            // Curved paths for better visibility
            return `M${d.source.x},${d.source.y}A${dr},${dr} 0 0,1 ${d.target.x},${d.target.y}`;
        });
    }
}

// Usage example
const visualizer = new FeynmanThreatVisualizer('#threat-diagram');

const sideChannelAttack = {
    nodes: [
        { id: 'quantum_gate', type: 'system', label: 'Q-Gate' },
        { id: 'power_monitor', type: 'threat', label: 'Power Analysis' },
        { id: 'leak', type: 'vulnerability', label: 'EM Leak' },
        { id: 'classical_observer', type: 'system', label: 'Observer' },
        { id: 'node_zero', type: 'zkp', label: 'Node Zero' }
    ],
    edges: [
        { source: 'quantum_gate', target: 'leak', type: 'quantum' },
        { source: 'leak', target: 'power_monitor', type: 'vulnerability' },
        { source: 'power_monitor', target: 'classical_observer', type: 'threat' },
        { source: 'node_zero', target: 'quantum_gate', type: 'zkp' }
    ]
};

visualizer.renderThreatDiagram(sideChannelAttack);
```

4.2 Real-Time Threat Dashboard

```html
```

```html
<!DOCTYPE html>
<html>
<head>
  <title>Quantum Threat Monitor - OBINexus</title>
  <style>
    .threat-panel {
      display: grid;
      grid-template-columns: 1fr 1fr 1fr;
      gap: 20px;
      padding: 20px;
    }
    .gate-indicator {
      width: 50px;
      height: 50px;
      border-radius: 50%;
      display: inline-block;
      margin: 10px;
    }
    .gate-active { background: #28a745; }
    .gate-inactive { background: #dc3545; }
    .threat-score {
      font-size: 48px;
      font-weight: bold;
      text-align: center;
    }
    .amplitude-chart {
      height: 300px;
      background: #f8f9fa;
      border: 1px solid #dee2e6;
    }
  </style>
</head>
<body>
  <h1>OBINexus Quantum Threat Monitor</h1>

  <div class="threat-panel">
    <div class="gate-status">
      <h2>Gate Status</h2>
      <div>
        <label>Gx (QA):</label>
        <span class="gate-indicator" id="gate-x"></span>
      </div>
      <div>
        <label>Gy (Quantum):</label>
        <span class="gate-indicator" id="gate-y"></span>
      </div>
```

```
        <div>
            <label>Gz (Blockchain):</label>
            <span class="gate-indicator" id="gate-z"></span>
        </div>
    </div>

    <div class="threat-analysis">
        <h2>Threat Score</h2>
        <div class="threat-score" id="threat-score">0.0</div>
        <div id="threat-function">T(x,y,z) = 0</div>
    </div>

    <div class="amplitude-display">
        <h2>Attack Amplitude</h2>
        <canvas id="amplitude-chart" class="amplitude-chart"></canvas>
    </div>
</div>

<div id="threat-diagram"></div>

<script src="https://d3js.org/d3.v7.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
    // Real-time monitoring implementation
    class ThreatMonitor {
        constructor() {
            this.gates = { x: 0, y: 0, z: 0 };
            this.amplitudeHistory = [];
            this.initCharts();
            this.startMonitoring();
        }

        initCharts() {
            const ctx = document.getElementById('amplitude-chart').getContext('2d');
            this.amplitudeChart = new Chart(ctx, {
                type: 'line',
                data: {
                    labels: [],
                    datasets: [{
                        label: 'Threat Amplitude',
                        data: [],
                        borderColor: 'rgb(220, 53, 69)',
                        tension: 0.1
                    }]
                },
                options: {
                    responsive: true,
```

```
            maintainAspectRatio: false,
            scales: {
                y: {
                    beginAtZero: true,
                    max: 1.0
                }
            }
        }
    });
}

async startMonitoring() {
    setInterval(async () => {
        // Simulate threat detection and gate updates
        const threats = await this.detectThreats();
        const gates = await this.verifyGates();

        this.updateDisplay(threats, gates);
    }, 1000);
}

async detectThreats() {
    // Simulate threat detection
    return {
        x: Math.random() * 24 - 12,
        y: Math.random() * 24 - 12,
        z: Math.random() * 24 - 12
    };
}

async verifyGates() {
    // Simulate Node Zero verification
    return {
        x: Math.random() > 0.3 ? 1 : 0,
        y: Math.random() > 0.2 ? 1 : 0,
        z: Math.random() > 0.1 ? 1 : 0
    };
}

updateDisplay(threats, gates) {
    // Update gate indicators
    ['x', 'y', 'z'].forEach(axis => {
        const indicator = document.getElementById(`gate-${axis}`);
        indicator.className = `gate-indicator ${gates[axis] ? 'gate-active' : 'gate-inactive'}`;
    });

    // Calculate threat score
```

```
            const T = 0.4 * threats.x + 0.3 * threats.y + 0.3 * threats.z;
            document.getElementById('threat-score').textContent = T.toFixed(2);
            document.getElementById('threat-function').textContent =
              `T(${threats.x.toFixed(1)}, ${threats.y.toFixed(1)}, ${threats.z.toFixed(1)}) = ${T.toFixed(2)}`;

            // Calculate amplitude
            const S = gates.x * gates.y * gates.z;
            const amplitude = Math.abs(T) * S / 12; // Normalized

            // Update chart
            const now = new Date().toLocaleTimeString();
            this.amplitudeChart.data.labels.push(now);
            this.amplitudeChart.data.datasets[0].data.push(amplitude);

            // Keep only last 20 points
            if (this.amplitudeChart.data.labels.length > 20) {
              this.amplitudeChart.data.labels.shift();
              this.amplitudeChart.data.datasets[0].data.shift();
            }

            this.amplitudeChart.update();
          }
        }

        // Initialize monitor
        const monitor = new ThreatMonitor();
      </script>
    </body>
    </html>
```

5. Advanced Side-Channel Attack Scenarios

5.1 Quantum Timing Side-Channel

```python
```

```python
class QuantumTimingSideChannel:
    """Models timing attacks on quantum gate operations"""

    def __init__(self):
        self.gate_timings = {
            'H': 1e-9,      # Hadamard gate
            'CNOT': 3e-9,   # Controlled-NOT
            'T': 2e-9,      # T gate
            'measure': 5e-9 # Measurement
        }

    def extract_circuit_info(self, timing_trace: List[float]) -> Dict:
        """Extract quantum circuit structure from timing measurements"""

        # Differential timing analysis
        deltas = np.diff(timing_trace)

        # Match against known gate timings
        identified_gates = []
        for delta in deltas:
            for gate, timing in self.gate_timings.items():
                if abs(delta - timing) < 1e-10:  # 10ps resolution
                    identified_gates.append(gate)
                    break

        # Calculate leakage amplitude
        information_gain = len(identified_gates) / len(deltas)

        return {
            'gates': identified_gates,
            'information_gain': information_gain,
            'threat_level': -12 * information_gain  # Map to threat scale
        }

    def mitigation_strategy(self) -> Dict:
        """Node Zero mitigation for timing attacks"""
        return {
            'randomize_gate_order': True,
            'insert_dummy_operations': True,
            'constant_time_execution': True,
            'zkp_verification_interval': 100e-9  # 100ns
        }
```

5.2 Electromagnetic Emanation Analysis

rust

```rust
// Rust implementation for EM side-channel detection
use nalgebra::{Vector3, DMatrix};
use rustfft::{FftPlanner, num_complex::Complex};

pub struct EMSideChannelDetector {
    fft_planner: FftPlanner<f64>,
    baseline_spectrum: Vec<Complex<f64>>,
    threat_threshold: f64,
}

impl EMSideChannelDetector {
    pub fn new(baseline: Vec<f64>) -> Self {
        let mut planner = FftPlanner::new();
        let fft = planner.plan_fft_forward(baseline.len());

        let mut spectrum: Vec<Complex<f64>> = baseline
            .iter()
            .map(|&x| Complex::new(x, 0.0))
            .collect();

        fft.process(&mut spectrum);

        Self {
            fft_planner: planner,
            baseline_spectrum: spectrum,
            threat_threshold: 0.1,
        }
    }

    pub fn analyze_em_trace(&mut self, trace: Vec<f64>) -> ThreatAssessment {
        // Convert to frequency domain
        let mut spectrum: Vec<Complex<f64>> = trace
            .iter()
            .map(|&x| Complex::new(x, 0.0))
            .collect();

        let fft = self.fft_planner.plan_fft_forward(spectrum.len());
        fft.process(&mut spectrum);

        // Compare with baseline
        let deviation = self.calculate_spectral_deviation(&spectrum);

        // Identify leaked information
        let leaked_freqs = self.identify_information_bearing_frequencies(&spectrum);

        ThreatAssessment {
```

```rust
            threat_level: self.map_to_threat_scale(deviation),
            leaked_frequencies: leaked_freqs,
            recommended_action: self.determine_mitigation(deviation),
        }
    }

    fn calculate_spectral_deviation(&self, spectrum: &[Complex<f64>]) -> f64 {
        spectrum.iter()
            .zip(self.baseline_spectrum.iter())
            .map(|(a, b)| (a - b).norm())
            .sum::<f64>() / spectrum.len() as f64
    }

    fn identify_information_bearing_frequencies(&self, spectrum: &[Complex<f64>]) -> Vec<f64> {
        let mut peaks = Vec::new();

        for (i, val) in spectrum.iter().enumerate() {
            if val.norm() > self.threat_threshold {
                let freq = i as f64 * 1e9 / spectrum.len() as f64; // Assuming 1GHz sampling
                peaks.push(freq);
            }
        }

        peaks
    }

    fn map_to_threat_scale(&self, deviation: f64) -> i8 {
        match deviation {
            d if d < 0.01 => 0,    // Neutral
            d if d < 0.05 => -2,   // Low threat
            d if d < 0.1 => -5,    // Medium threat
            d if d < 0.2 => -8,    // High threat
            _ => -11,              // Critical threat
        }
    }

    fn determine_mitigation(&self, deviation: f64) -> MitigationAction {
        if deviation > 0.1 {
            MitigationAction::EmergencyShielding
        } else if deviation > 0.05 {
            MitigationAction::IncreaseNoiseFloor
        } else {
            MitigationAction::ContinueMonitoring
        }
    }
}
```

```rust
#[derive(Debug)]
pub struct ThreatAssessment {
    pub threat_level: i8,
    pub leaked_frequencies: Vec<f64>,
    pub recommended_action: MitigationAction,
}

#[derive(Debug)]
pub enum MitigationAction {
    ContinueMonitoring,
    IncreaseNoiseFloor,
    EmergencyShielding,
}
```

6. Integration with OBINexus Toolchain

6.1 Build Pipeline

```bash
```

```bash
#!/bin/bash
# build.sh - OBINexus Quantum Threat Analyzer Build Script

echo "Building Quantum Threat Analyzer with OBINexus toolchain..."

# Step 1: Generate threat propagators with riftlang
echo "[1/5] Generating threat propagators..."
riftlang.exe \
  --input src/threats.rift \
  --output build/propagators.so.a \
  --mode quantum-threat \
  --dimensional-game-theory enabled

# Step 2: Compile lattice verification libraries
echo "[2/5] Compiling lattice verification..."
gcc -shared -fPIC \
  -o build/lattice_verify.so.a \
  src/lattice/*.c \
  -lm -lpthread

# Step 3: Build policy enforcement with rift.exe
echo "[3/5] Building policy engine..."
rift.exe \
  --propagators build/propagators.so.a \
  --lattice build/lattice_verify.so.a \
  --output build/policy_engine.exe \
  --gates "Gx,Gy,Gz" \
  --safety-threshold 3.0

# Step 4: Compile polyglot coordinator with gosilang
echo "[4/5] Building polyglot coordinator..."
gosilang build \
  --target threat-analyzer \
  --languages "rust,python,typescript,c" \
  --output build/coordinator \
  src/coordinator/*.go

# Step 5: Link everything with nlink
echo "[5/5] Linking with nlink..."
nlink \
  --inputs "build/*.so.a,build/*.exe" \
  --output dist/quantum-threat-analyzer \
  --polybuild-config polybuild.toml \
  --enable-node-zero \
  --enable-dimensional-reduction
```

```
echo "Build complete! Output: dist/quantum-threat-analyzer"
```

6.2 Configuration Files

```toml
# polybuild.toml - Polyglot build configuration
[project]
name = "quantum-threat-analyzer"
version = "1.0.0"
toolchain = "riftlang.exe → .so.a → rift.exe → gosilang"

[languages]
rust = { version = "1.70", features = ["nalgebra", "rustfft"] }
python = { version = "3.11", packages = ["numpy", "scipy", "qiskit"] }
typescript = { version = "5.0", packages = ["@obinexus/node-zero"] }
c = { version = "c11", libs = ["openssl", "pthread"] }

[threat-model]
dimensions = ["power", "timing", "electromagnetic", "acoustic"]
max_active_dimensions = 3
safety_threshold = 3.0

[node-zero]
enabled = true
verification_interval_ms = 100
challenge_timeout_ms = 5000
proof_size_bytes = 2048

[gates]
Gx = { name = "Software QA", default = 0 }
Gy = { name = "Quantum Integration", default = 0 }
Gz = { name = "Blockchain Verification", default = 0 }

[dimensional-game-theory]
enabled = true
reduction_threshold = 4
activation_function = "sigmoid"
```

7. Conclusion and Future Work

This vision document establishes a comprehensive framework for integrating Dimensional Game Theory with Quantum Threat Modeling. Key achievements:

1. **Mathematical Foundation**: Unified DGT and QFT for threat analysis

2. **Extended Scenarios**: Side-channel, multi-stage, and quantum attacks
3. **Implementation**: Complete toolchain integration with OBINexus
4. **Visualization**: Real-time monitoring and Feynman diagram generation
5. **Node Zero Integration**: Continuous ZKP verification for gate updates

Future Enhancements

1. **Machine Learning Integration**: Train models on threat patterns
2. **Automated Response**: AI-driven mitigation strategies
3. **Quantum Simulator**: Test attacks on simulated quantum systems
4. **Formal Verification**: Prove safety properties using Coq/Isabelle
5. **Hardware Integration**: Direct interface with quantum processors

Patents Integration

The framework leverages existing OBINexus patents:

- Dimensional Game Theory for strategic threat reduction
- Fault-Tolerant Cryptographic Integration with AuraSeal
- Quantum Field Theory Decision Framework
- RIFT Architecture for governed computation
- Unified Quantum-Classical Bridge Protocol

All components maintain #NoGhosting compliance and support the milestone-based investment model of the OBINexus Legal Policy.