OmniDork: Technical Implementation Guide

Project Structure

The OmniDork project combines multiple sophisticated components:

```
src/
├── main.rs                    # Main application entry point
├── tokenizer.rs               # Prime-based tokenization for quantum search
├── prime_hilbert.rs           # Hilbert space representations
├── entropy.rs                 # Entropy and persistence calculations
├── engine.rs                  # Core quantum resonant search engine
├── quantum_types.rs           # Quantum mathematical structure definitions
├── crawler.rs                 # Web crawler for content discovery
├── dork_engine.rs             # Google dorking and OSINT automation
├── vulnerability_matcher.rs   # Pattern matching for security issues
├── proxy_scanner.rs           # Proxy discovery and validation
├── bug_bounty.rs              # Bug bounty program integration
├── lib.rs                     # Library exports
```

Key Components

1. Quantum Resonant Search Engine

This is the core search technology that uses quantum-inspired algorithms:

- **Prime Tokenization**: Represents words as prime numbers to create unique mathematical structures
- **Biorthogonal Vectors**: Non-Hermitian quantum mechanics inspired representation with left/right eigenvectors
- **Persistence Theory**: Uses thermodynamic principles to evaluate information stability
- **Document Compression**: Efficiently manages memory with automatic document compression

2. OSINT Automation Framework

The OSINT components handle reconnaissance and vulnerability discovery:

- **DorkEngine**: Generates and executes optimized Google dorks for various targets
- **API Integration**: Connects to security services like Shodan, URLScan.io
- **JavaScript Analysis**: Extracts and analyzes JavaScript files for sensitive information
- **Pattern Matching**: Uses regex patterns to identify security issues

3. Proxy Scanner

The proxy component handles discovery and validation of anonymous proxies:

- **Multi-Source Fetching**: Collects proxies from multiple public sources
- **Concurrent Validation**: Tests proxies efficiently using async processing
- **Anonymity Analysis**: Determines the level of anonymity provided by each proxy
- **Speed Testing**: Measures and ranks proxies by performance

Implementation Details

Quantum-Inspired Algorithms

rust

```rust
// Calculate complex resonance with phase information
pub fn resonance_complex(vec1: &PrimeVector, vec2: &PrimeVector, decay_factor: f64)
    let dot_real = dot_product(vec1, vec2);

    // Use the decay factor as a basis for imaginary component
    Complex::new(dot_real, decay_factor)
}

// Calculate persistence score based on thermodynamic parameters
pub fn persistence_score(
    reversibility: f64,
    entropy_pressure: f64,
    buffering: f64,
    fragility: f64
) -> f64 {
    if buffering <= 0.0 {
        return 0.0; // Avoid division by zero
    }
    ((-fragility) * (1.0 - reversibility) * (entropy_pressure / buffering)).exp()
}
```

Concurrent Proxy Validation

rust

```rust
// Validate proxies concurrently with limited concurrency
stream::iter(proxies)
    .map(|proxy| {
        let validated_proxies = Arc::clone(&validated_proxies);

        async move {
            if let Ok(is_valid) = self.validate_proxy(&proxy).await {
                if is_valid {
                    // Get response time and details
                    if let Ok((response_time, country, anonymity)) =
                        self.measure_proxy_performance(&proxy).await {
                        // Store validated proxy
                        let mut proxies = validated_proxies.lock().await;
                        proxies.push(validated_proxy);
                    }
                }
            }
        }
    })
    .buffer_unordered(self.connection_limit)
    .collect::<Vec<()>>()
    .await;
```

Dork Generation and Execution

rust

```rust
// Generate dorks for a domain based on predefined templates
pub fn generate_dorks_for_domain(&self, domain: &str) -> Vec<String> {
    let mut dorks = Vec::new();

    for (_, dork_templates) in &self.dork_categories {
        for template in dork_templates {
            // Replace placeholders with domain
            let dork = template.replace("{domain}", domain);
            dorks.push(dork);
        }
    }

    dorks
}

// Execute a dork against a target domain
pub async fn execute_dork(&self, dork: &str, domain: &str) -> Result<Vec<DorkResult>
    // Build search query and execute it
    // Process search results
    // Return structured findings
}
```

Integration Between Components

1. Document Flow

```
Input Domain
    ↓
DorkEngine discovers subdomains
    ↓
Crawler indexes all content
    ↓
Engine builds quantum representations
    ↓
VulnerabilityMatcher analyzes content
    ↓
BugBountyManager matches findings to programs
```

2. Data Structures

The project uses several key data structures to represent findings:

```rust
// Finding from vulnerability analysis
struct Finding {
    id: String,
    target_id: String,
    finding_type: String,
    severity: String,
    url: Option<String>,
    description: String,
    discovery_timestamp: u64,
}

// Search result from quantum engine
struct SearchResult {
    title: String,
    resonance: f64,
    delta_entropy: f64,
    score: f64,
    quantum_score: f64,
    persistence_score: f64,
    snippet: String,
    path: String,
}

// Proxy information
struct ProxyInfo {
    ip: String,
    port: u16,
    protocol: String,
    anonymity: String,
    response_time: f64,
    country: String,
    last_checked: u64,
}
```

Advanced Features

1. Quantum Jump Mechanism

rust

```rust
// Apply a quantum jump to update document relevance
pub fn apply_quantum_jump(&mut self, query: &str, importance: f64) {
    // Find documents that match the query
    // Apply quantum jump to update their state
    // This changes future search results based on past queries
}

// Implementation details
pub fn quantum_jump_event(doc_state: &mut MatrixComplex<f64>, jump_operator: MatrixC
    // Apply the jump operator to both sides of the density matrix
    let result = &jump_operator * &(*doc_state) * &jump_operator.adjoint();
    *doc_state = result;

    // Normalize the density matrix by its trace
    let tr = trace(doc_state).re;
    if tr > 0.0 {
        for i in 0..doc_state.nrows() {
            for j in 0..doc_state.ncols() {
                doc_state[(i, j)] = doc_state[(i, j)] * Complex::new(1.0/tr, 0.0);
            }
        }
    }
}
```

2. Biorthogonal Vector Representation

rust

```rust
// Build a biorthogonal representation of a document
pub fn build_biorthogonal_vector(primes: &[u64]) -> BiorthogonalVector {
    let base_vector = build_vector(primes);

    // Create right vector with variations from the left vector
    let mut right_vector = PrimeVector::new();

    for (&prime, &value) in &base_vector {
        // Modify the weights slightly for the right vector
        right_vector.insert(prime, value * (1.0 + 0.1 * (prime % 2) as f64));
    }

    // Normalize the right vector
    let norm: f64 = f64::sqrt(right_vector.values().map(|&v| v * v).sum());
    if norm > 0.0 {
        for val in right_vector.values_mut() {
            *val /= norm;
        }
    }

    BiorthogonalVector {
        left: base_vector,
        right: right_vector,
    }
}
```

3. Document Compression

rust

```rust
// Compress the document text to save memory
fn compress_text(&mut self) {
    if !self.text.is_empty() && self.compressed_text.is_none() {
        let mut encoder = GzEncoder::new(Vec::new(), Compression::default());
        encoder.write_all(self.text.as_bytes()).unwrap_or_default();
        self.compressed_text = encoder.finish().ok();

        // Only clear the text if compression was successful
        if self.compressed_text.is_some() {
            self.text.clear();
        }
    }
}

// Decompress the text when needed
fn decompress_text(&mut self) -> &str {
    if self.text.is_empty() && self.compressed_text.is_some() {
        let compressed = self.compressed_text.as_ref().unwrap();
        let mut decoder = GzDecoder::new(&compressed[..]);
        let mut text = String::new();

        if decoder.read_to_string(&mut text).is_ok() {
            self.text = text;
        }
    }

    &self.text
}
```

Build and Run Instructions

Prerequisites

1. Rust toolchain installed (rustc, cargo)
2. PostgreSQL database for storing results
3. Required external dependencies:
    - OpenSSL development libraries
    - pkg-config

Setup

1. Clone the repository
2. Create the database with the provided schema

3. Build the project:

```bash
cargo build --release
```

Running the Application

```bash
# Run with default settings
cargo run --release

# Or execute the binary directly
./target/release/omnidork
```

Customization Options

1. **Database Connection String**:
   - Edit the `.env` file to set your PostgreSQL connection
2. **Proxy Sources**:
   - Modify the `proxy_sources` list in `proxy_scanner.rs`
3. **Dork Templates**:
   - Add or modify dork templates in `dork_engine.rs`
4. **Quantum Parameters**:
   - Adjust `fragility` and `entropy_weight` in the interactive menu

Future Development Plans

1. Expanded vulnerability detection patterns
2. Deeper integration with cloud APIs
3. Enhanced visualization capabilities
4. Machine learning for false positive reduction
5. Distributed scanning architecture

This guide covers the key technical aspects of the OmniDork implementation, providing a roadmap for understanding how the components work together and how to extend them for future development.