Ethernity MEV Detector - Algoritmos e Detecção de Padrões

1. Algoritmos de Classificação de Transações

1.1 Análise de Seletores

O `TxNatureTagger` utiliza um mapeamento determinístico de seletores de função (4 bytes) para tags:

```rust
// Seletores conhecidos
0x38ed1739 -> ["swap-v2", "router-call"]     // swapExactTokensForTokens
0x18cbaf95 -> ["swap-v3", "router-call"]     // exactInputSingle
0xa9059cbb -> ["transfer", "token-move"]     // transfer(address,uint256)
```

Algoritmo de Detecção

```rust
fn detect_selector(input: &[u8]) -> Option<Vec<String>> {
    if input.len() < 4 {
        return None;
    }

    let selector = [input[0], input[1], input[2], input[3]];
    SELECTOR_MAP.get(&selector).cloned()
}
```

1.2 Análise de Bytecode

Detecção de padrões no bytecode do contrato para identificar proxies e delegações:

rust

```rust
fn analyze_bytecode(code: &[u8]) -> (Vec<String>, f64) {
    let mut tags = Vec::new();
    let mut confidence = 0.5;

    // Detecta DELEGATECALL (0xf4)
    if code.iter().any(|&b| b == 0xf4) {
        tags.push("proxy-call".to_string());
        confidence = 0.7;
    }

    // Detecta padrões de implementação mínima
    if is_minimal_proxy(code) {
        tags.push("minimal-proxy".to_string());
        confidence = 0.8;
    }

    (tags, confidence)
}
```

1.3 Extração de Caminhos de Token

Algoritmo para extrair endereços de tokens do calldata:

rust

```rust
fn extract_token_paths(input: &[u8]) -> Vec<Address> {
    let mut paths = Vec::new();

    // Pula selector (4 bytes)
    if input.len() <= 4 {
        return paths;
    }

    // Processa chunks de 32 bytes
    for chunk in input[4..].chunks(32) {
        if chunk.len() == 32 {
            // Verifica se parece com endereço (12 zeros + 20 bytes)
            if chunk[0..12].iter().all(|&b| b == 0) {
                let addr = Address::from_slice(&chunk[12..32]);
                if addr != Address::zero() {
                    paths.push(addr);
                }
            }
        }
    }

    // Deduplica mantendo ordem
    paths.dedup();
    paths
}
```

2. Algoritmos de Agregação

2.1 Geração de Chave de Grupo

Utiliza Keccak256 para gerar chave determinística:

rust

```rust
fn group_key(
    token_paths: &[Address],
    targets: &[Address],
    tags: &[String]
) -> H256 {
    let mut bytes = Vec::new();

    // Adiciona tokens (ordem importa)
    for addr in token_paths {
        bytes.extend_from_slice(addr.as_bytes());
    }

    // Adiciona alvos
    for addr in targets {
        bytes.extend_from_slice(addr.as_bytes());
    }

    // Adiciona assinatura de tags
    let tag_sig = tags_signature(tags);
    bytes.extend_from_slice(tag_sig.as_bytes());

    // Hash final
    let mut keccak = Keccak::v256();
    keccak.update(&bytes);
    let mut out = [0u8; 32];
    keccak.finalize(&mut out);
    H256::from(out)
}
```

2.2 Ordenação e Certeza

Algoritmo para calcular certeza de ordenação:

rust

```rust
fn calc_ordering_certainty(group: &TxGroup) -> f64 {
    if group.txs.len() <= 1 {
        return 1.0;
    }

    let first = group.txs.first().unwrap().first_seen as f64;
    let last = group.txs.last().unwrap().first_seen as f64;
    let delta = last - first;

    // Janela de 30 segundos = alta certeza
    if delta <= 30.0 {
        1.0
    } else {
        // Decai linearmente
        (1.0 - (delta - 30.0) / 270.0).max(0.0)
    }
}
```

2.3 Detecção de Contaminação

Identifica grupos com alta variância de confiança:

rust

```rust
fn calc_contamination(group: &TxGroup) -> bool {
    if group.txs.is_empty() {
        return false;
    }

    // Calcula média
    let avg = group.txs.iter()
        .map(|t| t.confidence)
        .sum::<f64>() / group.txs.len() as f64;

    // Calcula desvio padrão
    let variance = group.txs.iter()
        .map(|t| {
            let d = t.confidence - avg;
            d * d
        })
        .sum::<f64>() / group.txs.len() as f64;

    let std_dev = variance.sqrt();

    // Contaminated se desvio > 20%
    std_dev > 0.2
}
```

3. Algoritmos de Impacto Econômico

3.1 Modelo Constant Product (Uniswap V2)

rust

```rust
fn expected_out_v2(
    amount_in: f64,
    reserve_in: f64,
    reserve_out: f64,
    fee_rate: f64,
) -> f64 {
    // Validações de segurança
    if reserve_in <= 0.0 || reserve_out <= 0.0 || amount_in <= 0.0 {
        return 0.0;
    }

    let fee_multiplier = 1.0 - fee_rate;
    let numerator = amount_in * fee_multiplier * reserve_out;
    let denominator = reserve_in + amount_in * fee_multiplier;

    if denominator <= 0.0 || !denominator.is_finite() {
        return 0.0;
    }

    let out = numerator / denominator;
    if out.is_finite() && out >= 0.0 { out } else { 0.0 }
}
```

3.2 Modelo Uniswap V3

rust

```rust
fn expected_out_v3(amount_in: f64, sqrt_price_x96: f64) -> f64 {
    if sqrt_price_x96 <= 0.0 {
        return 0.0;
    }

    // Converte sqrt_price para ratio
    let price_ratio = (sqrt_price_x96 * sqrt_price_x96) / 2_f64.powi(192);

    if !price_ratio.is_finite() || price_ratio <= 0.0 {
        0.0
    } else {
        amount_in * price_ratio
    }
}
```

3.3 Cálculo de Slippage

rust

```rust
fn calculate_slippage(
    expected_out: f64,
    min_out: f64
) -> f64 {
    if expected_out <= 0.0 {
        return 0.0;
    }

    ((expected_out - min_out) / expected_out) * 100.0
}
```

3.4 Histórico Adaptativo de Slippage

rust

```rust
struct SlippageHistory {
    window: usize,
    values: Vec<f64>,
}

impl SlippageHistory {
    fn record(&mut self, value: f64) {
        self.values.push(value);
        if self.values.len() > self.window {
            self.values.remove(0);
        }
    }

    fn moving_average(&self) -> f64 {
        if self.values.is_empty() {
            0.0
        } else {
            self.values.iter().sum::<f64>() / self.values.len() as f64
        }
    }

    fn weighted_average(&self) -> f64 {
        if self.values.is_empty() {
            return 0.0;
        }

        let mut sum = 0.0;
        let mut weight_sum = 0.0;

        for (i, &value) in self.values.iter().enumerate() {
            let weight = (i + 1) as f64; // Peso linear crescente
            sum += value * weight;
            weight_sum += weight;
        }

        sum / weight_sum
    }
}
```

4. Algoritmos de Detecção de Ataques

4.1 Detecção de Frontrun

rust

```rust
fn detect_frontrun(txs: &[(AnnotatedTx, f64)]) -> Option<(Vec<H256>, f64)> {
    for i in 0..txs.len() {
        for j in (i + 1)..txs.len() {
            let (tx_i, priority_i) = &txs[i];
            let (tx_j, priority_j) = &txs[j];

            // Verifica janela temporal
            let dt = tx_j.first_seen.saturating_sub(tx_i.first_seen);
            if dt > ENTROPY_TOLERANCE_WINDOW {
                continue;
            }

            // Calcula dominância de prioridade
            if priority_i > priority_j {
                let dominance = priority_i / (priority_i + priority_j);
                if dominance > 0.65 { // 65% threshold
                    return Some((
                        vec![tx_i.tx_hash, tx_j.tx_hash],
                        dominance
                    ));
                }
            }
        }
    }
    None
}
```

4.2 Detecção de Sandwich

rust

```rust
fn detect_sandwich(txs: &[(AnnotatedTx, f64)]) -> Option<(Vec<H256>, f64)> {
    if txs.len() < 3 {
        return None;
    }

    for i in 0..txs.len() - 2 {
        let a = &txs[i];
        for j in (i + 1)..txs.len() - 1 {
            let b = &txs[j];
            // Verifica janela A-B
            let dt1 = b.0.first_seen.saturating_sub(a.0.first_seen);
            if dt1 > ENTROPY_TOLERANCE_WINDOW {
                continue;
            }

            for k in (j + 1)..txs.len() {
                let c = &txs[k];
                // Verifica janela B-C
                let dt2 = c.0.first_seen.saturating_sub(b.0.first_seen);
                if dt2 > ENTROPY_TOLERANCE_WINDOW {
                    continue;
                }

                // Padrão sandwich: A e C têm prioridade > B
                if a.1 > b.1 && c.1 > b.1 {
                    let dominance = (a.1 + c.1) / (a.1 + b.1 + c.1);
                    if dominance > 0.6 {
                        return Some((
                            vec![a.0.tx_hash, b.0.tx_hash, c.0.tx_hash],
                            dominance
                        ));
                    }
                }
            }
        }
    }
    None
}
```

4.3 Detecção de Spoofing

rust

```rust
fn detect_spoof(txs: &[(AnnotatedTx, f64)]) -> Option<(Vec<H256>, f64)> {
    let avg_gas = txs.iter()
        .map(|(t, _)| t.gas_price)
        .sum::<f64>() / txs.len() as f64;

    for (tx, _) in txs {
        let anomaly_score = calculate_anomaly_score(tx);
        let high_gas = tx.gas_price > avg_gas * 2.0;

        // Combina sinais
        let likelihood = if high_gas { 0.5 } else { 0.0 } + anomaly_score;

        if likelihood >= 0.5 {
            return Some((vec![tx.tx_hash], likelihood));
        }
    }
    None
}

fn calculate_anomaly_score(tx: &AnnotatedTx) -> f64 {
    let mut score = 0.0;

    // Tags anormalmente longas
    let unusual_tags = tx.tags.iter()
        .filter(|t| t.len() > 20)
        .count() as f64;
    score += unusual_tags * 0.1;

    // Múltiplos alvos
    if tx.targets.len() > 3 {
        score += 0.2;
    }

    // Path circular
    if has_circular_path(&tx.token_paths) {
        score += 0.3;
    }

    score.min(1.0)
}
```

4.4 Detecção de Flash Loan

rust

```rust
fn detect_flash_loan(group: &TxGroup) -> Option<(Vec<H256>, f64)> {
    // Busca por tags indicativas
    let has_flash_tag = group.txs.iter()
        .any(|t| t.tags.iter().any(|tag|
            tag.contains("flash-loan") ||
            tag.contains("flash")
        ));

    if !has_flash_tag {
        return None;
    }

    // Características adicionais
    let mut confidence = 0.7;

    // Alta prioridade de gas indica urgência
    let high_priority_count = group.txs.iter()
        .filter(|t| t.max_priority_fee_per_gas
            .map(|p| p > 10.0)
            .unwrap_or(false))
        .count();

    if high_priority_count > group.txs.len() / 2 {
        confidence += 0.1;
    }

    // Múltiplos tokens indica complexidade
    if group.token_paths.len() >= 3 {
        confidence += 0.1;
    }

    Some((
        group.txs.iter().map(|t| t.tx_hash).collect(),
        confidence.min(0.9)
    ))
}
```

5. Algoritmos de Otimização

5.1 Cache LRU com Eviction Policy

rust

```rust
impl<K: Hash + Eq, V> LruCache<K, V> {
    fn get_or_insert_with<F>(&mut self, key: K, f: F) -> &V
    where
        F: FnOnce() -> V,
    {
        if !self.contains(&key) {
            let value = f();
            self.put(key.clone(), value);
        }
        self.get(&key).unwrap()
    }
}
```

5.2 TTL Adaptativo

rust

```rust
fn adaptive_ttl(tx: &AnnotatedTx, mode: OperationalMode) -> Duration {
    match mode {
        OperationalMode::Burst => Duration::from_secs(3),
        OperationalMode::Recovery => Duration::from_secs(7),
        OperationalMode::Normal => {
            if tx.gas_price > 100.0 {
                Duration::from_secs(3) // High priority
            } else {
                Duration::from_secs(5) // Normal
            }
        }
    }
}
```

5.3 Compactação de Grupos

rust

```rust
fn compact_groups(groups: &mut HashMap<H256, TxGroup>, max_size: usize) {
    if groups.len() <= max_size {
        return;
    }

    // Score baseado em idade e tamanho
    let mut scored: Vec<_> = groups.iter()
        .map(|(key, group)| {
            let age = current_time() - group.window_start;
            let size = group.txs.len() as f64;
            let score = age as f64 / size; // Mais velho e menor = maior score
            (*key, score)
        })
        .collect();

    // Remove grupos com maior score
    scored.sort_by(|a, b| b.1.partial_cmp(&a.1).unwrap());
    let to_remove = groups.len() - max_size;

    for (key, _) in scored.iter().take(to_remove) {
        groups.remove(key);
    }
}
```

6. Algoritmos de Sincronização

6.1 Alinhamento de Estado

rust

```rust
fn compute_state_alignment(
    group: &TxGroup,
    current_block: u64
) -> f64 {
    match group.block_number {
        Some(bn) if bn >= current_block.saturating_sub(1) => 1.0,
        Some(bn) => {
            // Decai exponencialmente com distância
            let distance = current_block.saturating_sub(bn) as f64;
            0.5_f64.powf(distance / 10.0)
        }
        None => 0.8, // Sem referência de bloco
    }
}
```

6.2 Cálculo de Jitter

rust

```rust
fn compute_jitter(group: &TxGroup) -> f64 {
    if group.txs.len() <= 1 {
        return 0.0;
    }

    // Timestamps como f64
    let timestamps: Vec<f64> = group.txs.iter()
        .map(|t| t.first_seen as f64)
        .collect();

    // Média
    let mean = timestamps.iter().sum::<f64>() / timestamps.len() as f64;

    // Desvio padrão
    let variance = timestamps.iter()
        .map(|&t| (t - mean).powi(2))
        .sum::<f64>() / timestamps.len() as f64;

    variance.sqrt()
}
```