Ethernity MEV Detector - Requisitos, Limitações e Extensões

1. Requisitos do Sistema

1.1 Requisitos de Hardware

Mínimos

- **CPU**: 2 cores @ 2.4GHz
- **RAM**: 4GB
- **Disco**: 10GB SSD (para banco de dados)
- **Rede**: 100 Mbps estável

Recomendados

- **CPU**: 4+ cores @ 3.0GHz
- **RAM**: 16GB
- **Disco**: 50GB NVMe SSD
- **Rede**: 1 Gbps com baixa latência

Produção

- **CPU**: 8+ cores @ 3.5GHz
- **RAM**: 32GB+
- **Disco**: 500GB+ NVMe RAID
- **Rede**: 10 Gbps dedicada

1.2 Requisitos de Software

toml

```toml
[dependencies]
# Core
ethernity-core = "0.1"
ethers = "2.0"
ethereum-types = "0.14"

# Async runtime
tokio = { version = "1.35", features = ["full"] }
async-trait = "0.1"

# Storage
redb = "2.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"

# Utilities
lru = "0.12"
parking_lot = "0.12"
dashmap = "5.5"
tiny-keccak = { version = "2.0", features = ["keccak"] }
chrono = "0.4"

# Dev dependencies
tempfile = "3"
hex = "0.4"
anyhow = "1.0"
```

1.3 Requisitos de RPC

Endpoints Necessários

rust

```rust
// Obrigatórios
eth_call              // Para leitura de estado
eth_getCode           // Para análise de bytecode
eth_getBlockNumber    // Para sincronização
eth_getBlockByNumber  // Para recuperar blocos

// Opcionais (melhor performance)
eth_getStorageAt      // Para leitura direta de slots
eth_getLogs           // Para eventos históricos
eth_subscribe         // Para WebSocket updates
```

Provedores Testados

- **Infura**: Totalmente compatível
- **Alchemy**: Totalmente compatível
- **QuickNode**: Totalmente compatível
- **Geth**: Requer `--rpcapi "eth,net,web3"`
- **Erigon**: Performance superior
- **BSC**: Compatível com ajustes de timing

1.4 Requisitos de Rede

```yaml
Conectividade:
  - Latência RPC: < 50ms (ideal < 20ms)
  - Banda: 10+ Mbps sustentados
  - Perda de pacotes: < 0.1%

Portas:
  - HTTPS (443): Para RPC
  - WSS (443/8546): Para WebSocket

Firewall:
  - Outbound: Liberado para endpoints RPC
  - Inbound: Não necessário (cliente apenas)
```

2. Limitações Conhecidas

2.1 Limitações de Design

Detecção Passiva

```rust
// ❌ Não suportado
- Simulação completa de transações
- Execução de bytecode EVM
- Análise de traces detalhados
- Detecção de reentrancy complexa

// ✅ Suportado
- Inferência baseada em padrões
- Análise estática de calldata
- Heurísticas de comportamento
- Detecção probabilística
```

Cobertura de Protocolos

```rust
// Protocolos bem suportados
- Uniswap V2 e forks
- Uniswap V3 básico
- SushiSwap
- Tokens ERC20 padrão

// Suporte limitado
- Curve Finance (stableswap)
- Balancer (weighted pools)
- Protocolos de lending complexos
- Bridges cross-chain

// Não suportado
- Order books on-chain
- Protocolos privados/obscuros
- MEV em L2 nativo
```

2.2 Limitações de Performance

Throughput

```rust
// Capacidade típica por instância
const MAX_TPS: usize = 1000;          // Transações/segundo
const MAX_GROUPS: usize = 1000;       // Grupos simultâneos
const MAX_CACHE_SIZE: usize = 50;     // MB de RAM para cache

// Latências esperadas
const CLASSIFICATION_P99: Duration = Duration::from_millis(5);
const AGGREGATION_P99: Duration = Duration::from_micros(500);
const DETECTION_P99: Duration = Duration::from_millis(10);
```

Escalabilidade

```rust
// Gargalos identificados
1. RPC calls são síncronos por grupo
2. Banco de dados é single-writer
3. Cache LRU tem contenção em alta carga
4. Detecção é O(n²) para sandwich

// Mitigações
- Pool de conexões RPC
- Batch de escritas no DB
- Sharding de cache por CPU
- Early-exit em detecção
```

2.3 Limitações de Precisão

Falsos Positivos

```rust
// Causas comuns
- Coincidência temporal em transações
- Padrões similares não-maliciosos
- Agregação incorreta de grupos
- Inferência incorreta de tokens

// Taxa típica: 5-10% dependendo dos thresholds
```

Falsos Negativos

```rust
// Causas comuns
- Ataques sofisticados/novos
- Obfuscação intencional
- Transações via contratos proxy
- MEV privado/dark pools

// Taxa estimada: 15-20% dos ataques reais
```

2.4 Limitações de Estado

```rust
// Snapshots
- Lag de 1-2 blocos é comum
- Forks podem invalidar cache
- Estado parcial apenas (reserves)

// Histórico
- Limitado a 3 snapshots por endereço
- Sem análise de tendências longas
- Volatilidade calculada localmente
```

3. Problemas Conhecidos e Soluções

3.1 RPC Rate Limiting

Problema: Providers limitam requisições

Solução:

```rust
// Implementar retry com backoff
async fn call_with_retry<T>(
    provider: &impl RpcProvider,
    call: impl Fn() -> Future<Output = Result<T>>,
    max_retries: u32,
) -> Result<T> {
    let mut delay = Duration::from_millis(100);

    for attempt in 0..max_retries {
        match call().await {
            Ok(result) => return Ok(result),
            Err(Error::RpcError(e)) if e.contains("rate") => {
                tokio::time::sleep(delay).await;
                delay *= 2; // Exponential backoff
            }
            Err(e) => return Err(e),
        }
    }

    Err(Error::RpcError("Max retries exceeded".into()))
}

// Usar múltiplos providers
let providers = vec![primary, secondary, tertiary];
let provider = LoadBalancedProvider::new(providers);
```

3.2 Memory Pressure

Problema: Alto uso de memória em burst

Solução:

```rust
// Limitar grupos ativos
impl TxAggregator {
    fn enforce_memory_limit(&mut self) {
        const MAX_MEMORY_MB: usize = 100;

        let estimated_usage = self.estimate_memory_usage();
        if estimated_usage > MAX_MEMORY_MB * 1024 * 1024 {
            self.evict_oldest_groups(estimated_usage / 10);
        }
    }
}

// Usar arena allocator
use typed_arena::Arena;
let arena = Arena::new();
let groups = arena.alloc_many(initial_groups);
```

3.3 Fork Handling

Problema: Reorganizações invalidam estado

Solução:

rust

```rust
// Validar hashes em cada uso
async fn validate_and_refresh(
    &self,
    address: Address,
    block: u64,
) -> Result<StateSnapshot> {
    let stored = self.get_state(address, block)?;
    let current_hash = self.provider.get_block_hash(block).await?;

    if stored.block_hash != current_hash {
        // Fork detectado, invalida e re-fetch
        self.invalidate_block(block)?;
        self.fetch_fresh_state(address, block).await
    } else {
        Ok(stored)
    }
}
```

4. Extensões Futuras

4.1 Suporte a Novos Protocolos

rust

```rust
// Interface para protocolo customizado
pub trait ProtocolAdapter: Send + Sync {
    /// Identifica se transação pertence ao protocolo
    fn matches(&self, to: Address, selector: [u8; 4]) -> bool;

    /// Extrai informações específicas
    fn extract_swap_params(&self, input: &[u8]) -> Result<SwapParams>;

    /// Calcula impacto esperado
    fn calculate_impact(&self, params: &SwapParams, state: &dyn StateProvider) -> Resul
}

// Exemplo: Adapter para Curve
struct CurveAdapter {
    pool_registry: HashMap<Address, CurvePoolInfo>,
}

impl ProtocolAdapter for CurveAdapter {
    fn matches(&self, to: Address, selector: [u8; 4]) -> bool {
        self.pool_registry.contains_key(&to) &&
        CURVE_SELECTORS.contains(&selector)
    }

    fn extract_swap_params(&self, input: &[u8]) -> Result<SwapParams> {
        // Parse específico do Curve
        // exchange(i, j, dx, min_dy)
        todo!()
    }

    fn calculate_impact(&self, params: &SwapParams, state: &dyn StateProvider) -> Resul
        // StableSwap math
        todo!()
    }
}
```

4.2 Machine Learning para Detecção

rust

```rust
// Feature extraction
struct MevFeatures {
    gas_price_ratio: f64,
    time_delta: f64,
    position_in_block: f64,
    token_liquidity: f64,
    historical_activity: f64,
}

// Modelo treinado offline
struct MevClassifier {
    model: XGBoostModel,
    feature_pipeline: FeaturePipeline,
}

impl MevClassifier {
    fn predict(&self, group: &TxGroup) -> MevPrediction {
        let features = self.feature_pipeline.extract(group);
        let probabilities = self.model.predict(&features);

        MevPrediction {
            is_mev: probabilities[0] > 0.5,
            confidence: probabilities[0],
            attack_type: self.classify_attack_type(&probabilities),
        }
    }
}
```

4.3 Simulação Lightweight

rust

```rust
// Mini-EVM para casos específicos
struct LightweightSimulator {
    opcodes: HashMap<u8, OpHandler>,
    gas_limit: u64,
}

impl LightweightSimulator {
    fn simulate_critical_path(
        &mut self,
        code: &[u8],
        calldata: &[u8],
        state: &mut MockState,
    ) -> Result<SimulationResult> {
        // Simula apenas paths críticos
        // Foca em SLOAD, SSTORE, CALL
        todo!()
    }
}
```

4.4 Integração L2

rust

```rust
// Abstração para diferentes L2s
pub trait Layer2Provider: RpcProvider {
    /// Tempo de finalidade esperado
    fn finality_time(&self) -> Duration;

    /// Custo de bridge
    fn bridge_cost(&self) -> Result<U256>;

    /// Detecção MEV específica
    fn detect_l2_mev(&self, tx: &Transaction) -> Result<L2MevType>;
}

// Implementações
struct OptimismProvider { /* ... */ }
struct ArbitrumProvider { /* ... */ }
struct PolygonProvider { /* ... */ }
```

4.5 Dashboard e Monitoring

```rust
// Métricas Prometheus
use prometheus::{Counter, Histogram, Registry};

struct MevMetrics {
    transactions_processed: Counter,
    groups_created: Counter,
    attacks_detected: Counter,
    detection_latency: Histogram,
    snapshot_cache_hits: Counter,
    rpc_errors: Counter,
}

// API REST para dashboard
use axum::{Router, Json};

async fn metrics_handler() -> Json<DashboardData> {
    Json(DashboardData {
        total_attacks: METRICS.attacks_detected.get(),
        detection_rate: calculate_detection_rate(),
        top_pairs: get_top_targeted_pairs(),
        profit_distribution: get_profit_histogram(),
    })
}

let app = Router::new()
    .route("/metrics", get(metrics_handler))
    .route("/groups", get(list_active_groups))
    .route("/attacks", get(recent_attacks));
```

5. Roadmap de Desenvolvimento

5.1 Curto Prazo (1-3 meses)

- [ ] Suporte para Uniswap V4
- [ ] Otimização de cache distribuído
- [ ] API GraphQL para queries
- [ ] Melhor suporte para flashloans

5.2 Médio Prazo (3-6 meses)

- [ ] ML model para detecção
- [ ] Simulação lightweight
- [ ] Multi-chain support (Polygon, BSC)
- [ ] Plugin system para protocolos

5.3 Longo Prazo (6-12 meses)

- [ ] ZK proofs para detecção privada
- [ ] Integração com sequencers L2
- [ ] Análise de bytecode avançada
- [ ] MEV-Share compatibility

6. Considerações de Segurança

6.1 Segurança Operacional

```rust
// Sanitização de inputs
fn sanitize_address(input: &str) -> Result<Address> {
    let cleaned = input.trim().to_lowercase();
    if !cleaned.starts_with("0x") || cleaned.len() != 42 {
        return Err(Error::InvalidInput("Invalid address format".into()));
    }

    Address::from_str(&cleaned)
        .map_err(|_| Error::InvalidInput("Invalid hex in address".into()))
}

// Rate limiting interno
struct RateLimiter {
    requests: DashMap<IpAddr, Vec<Instant>>,
    max_per_minute: usize,
}
```

6.2 Proteção de Dados

rust

```rust
// Criptografia de snapshots sensíveis
use aes_gcm::{Aes256Gcm, Key, Nonce};

fn encrypt_snapshot(snapshot: &StateSnapshot, key: &Key) -> Vec<u8> {
    let cipher = Aes256Gcm::new(key);
    let nonce = generate_nonce();
    let plaintext = serde_json::to_vec(snapshot).unwrap();

    cipher.encrypt(&nonce, plaintext.as_ref())
        .expect("encryption failure")
}

// Logs sem PII
fn log_transaction(tx: &AnnotatedTx) {
    info!(
        "TX processed: hash={:x} tags={:?} confidence={}",
        tx.tx_hash,
        tx.tags,
        tx.confidence
    );
    // Nunca logar addresses completos ou valores
}
```

7. Contribuindo

7.1 Guidelines

bash

```bash
# Setup desenvolvimento
git clone https://github.com/ethernity/detector-mev
cd detector-mev
cargo build --all-features
cargo test --all

# Estilo de código
cargo fmt -- --check
cargo clippy -- -D warnings

# Benchmarks
cargo bench --bench detection_performance
```

7.2 Áreas Prioritárias

1. **Novos Detectores**: Padrões de MEV emergentes

2. **Performance**: Otimizações de hot paths
3. **Testes**: Casos edge e fuzzing
4. **Documentação**: Exemplos e tutoriais
5. **Integrações**: Novos protocolos DeFi