Ethernity MEV Detector - Visão Geral e Arquitetura

1. Introdução

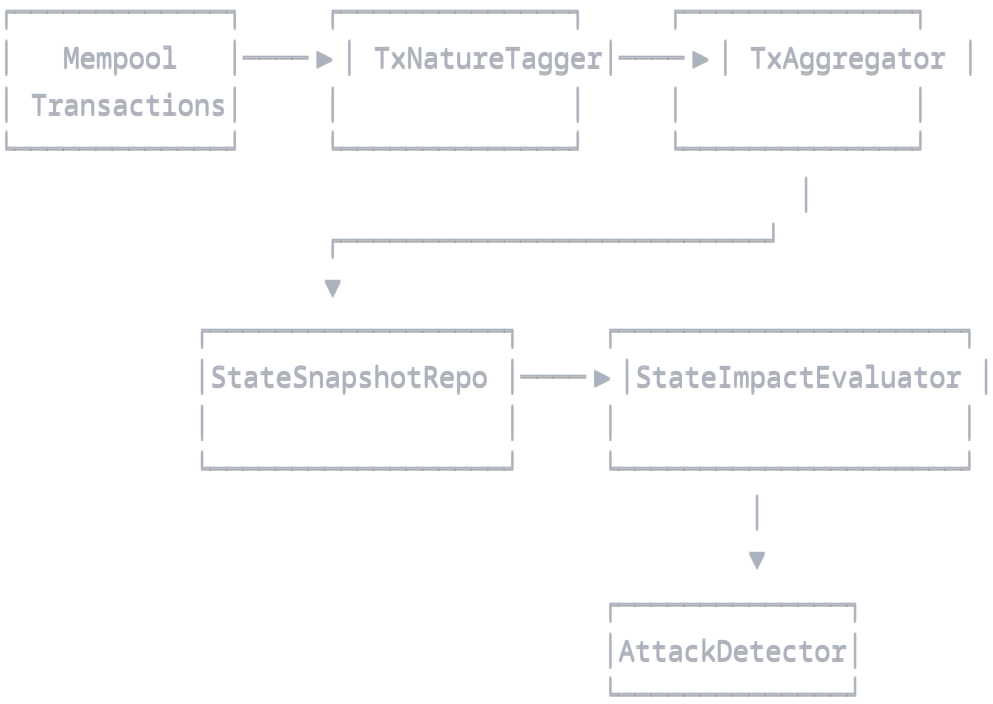
O `ethernity-detector-mev` é uma crate Rust especializada em detecção passiva de oportunidades MEV (Maximal Extractable Value) no mempool Ethereum. Diferentemente de soluções que simulam ou executam transações, esta crate opera exclusivamente através de análise estática e inferência local, tornando-a ideal para ambientes com recursos limitados ou requisitos de baixa latência.

1.1 Características Principais

- **Detecção Passiva**: Não simula nem envia transações
- **Multi-Vítima**: Identifica agrupamentos de múltiplas vítimas potenciais
- **Multi-Token**: Suporta análise de swaps envolvendo múltiplos tokens
- **Tempo Real**: Processamento otimizado para análise em tempo real do mempool
- **Persistência**: Armazenamento eficiente de snapshots usando `redb`
- **Extensível**: Arquitetura modular permite adição de novos detectores

2. Arquitetura do Sistema

2.1 Pipeline de Processamento



2.2 Componentes Principais

2.2.1 TxNatureTagger

Responsável pela classificação inicial de transações baseada em:

- Análise de seletores de função (4 bytes)
- Detecção de bytecode do contrato destino
- Extração de caminhos de tokens do calldata
- Inferência de tags como "swap-v2", "swap-v3", "proxy-call"

2.2.2 TxAggregator

Agrupa transações relacionadas usando:

- Hash determinístico baseado em tokens e alvos
- Janelas temporais adaptativas
- Cache LRU para limitar uso de memória
- Detecção de contaminação e reordenabilidade

2.2.3 StateSnapshotRepository

Gerencia snapshots de estado on-chain:

- Persistência em banco de dados `redb`
- Versionamento de schema para compatibilidade
- Detecção de forks e invalidação de cache
- Suporte a múltiplos perfis de snapshot (Basic, Extended, Deep)

2.2.4 StateImpactEvaluator

Calcula o impacto econômico esperado:

- Modelos de curva para AMMs (Constant Product, Uniswap V3)
- Cálculo de slippage tolerada vs esperada
- Histórico adaptativo de slippage
- Simulação leve de sequências de trades

2.2.5 AttackDetector

Identifica padrões de ataque MEV:

- Frontrun: Detecção por prioridade de gas dominante
- Sandwich: Análise de sequências A-B-A com gas patterns
- Backrun: Identificação de transações oportunistas
- Ataques complexos: Flash loans, cross-chain, multi-token

2.2.6 MempoolSupervisor

Orquestra todo o pipeline:

- Gerenciamento de modos operacionais (Normal, Burst, Recovery)
- TTL adaptativo baseado em condições de rede
- Sincronização com altura de bloco
- Emissão de eventos para processamento assíncrono

3. Fluxo de Dados

3.1 Entrada de Transação

rust

```rust
struct RawTx {
    pub tx_hash: TransactionHash,
    pub to: Address,
    pub input: Vec<u8>,
    pub first_seen: u64,
    pub gas_price: f64,
    pub max_priority_fee_per_gas: Option<f64>,
}
```

3.2 Transação Anotada

rust

```rust
struct AnnotatedTx {
    pub tx_hash: TransactionHash,
    pub token_paths: Vec<Address>,
    pub targets: Vec<Address>,
    pub tags: Vec<String>,
    pub first_seen: u64,
    pub gas_price: f64,
    pub max_priority_fee_per_gas: Option<f64>,
    pub confidence: f64,
}
```

3.3 Grupo de Transações

rust

```rust
struct TxGroup {
    pub group_key: H256,
    pub token_paths: Vec<Address>,
    pub targets: Vec<Address>,
    pub txs: Vec<AnnotatedTx>,
    pub block_number: Option<u64>,
    pub direction_signature: String,
    pub ordering_certainty_score: f64,
    pub reorderable: bool,
    pub contaminated: bool,
    pub window_start: u64,
}
```

3.4 Resultado Final

rust

```rust
struct GroupImpact {
    pub group_id: H256,
    pub tokens: Vec<Address>,
    pub victims: Vec<VictimImpact>,
    pub opportunity_score: f64,
    pub expected_profit_backrun: f64,
    pub state_confidence: f64,
    pub impact_certainty: f64,
    pub execution_assumption: String,
    pub reorg_risk_level: String,
}
```

4. Modelo de Concorrência

4.1 Paralelização Natural

- Grupos independentes podem ser processados em paralelo
- Cada grupo tem sua própria chave determinística
- Não há dependências entre grupos diferentes

4.2 Proteções de Concorrência

- `Mutex` para cache de código de contratos
- `DashMap` para buffer de transações no supervisor
- `LruCache` thread-safe para estado RPC

4.3 Pipeline Assíncrono

rust

```rust
// Exemplo de pipeline com canais
let (tx_raw, rx_raw) = mpsc::channel(1024);
let (tx_annotated, rx_annotated) = mpsc::channel(1024);
let (tx_grouped, rx_grouped) = mpsc::channel(256);

tokio::spawn(tagger.process_stream(rx_raw, tx_annotated));
tokio::spawn(aggregator.process_stream(rx_annotated, tx_grouped));
```

5. Persistência e Estado

5.1 Banco de Dados redb

- Armazenamento key-value eficiente
- Transações ACID para consistência
- Compactação automática
- Suporte a backup/restore

5.2 Schema de Dados

```
Tabela: snapshots
Chave: "0x{address}:{block_number}:{profile}"
Valor: PersistedSnapshot (JSON serializado)

Tabela: metadata
Chave: "schema_version"
Valor: u32 (versão atual: 1)
```

5.3 Gerenciamento de Forks

- Validação de `block_hash` em cada snapshot
- Invalidação automática em caso de fork
- Re-fetch transparente de dados atualizados

6. Métricas de Desempenho

6.1 Latência Típica

- Classificação de transação: <1ms
- Agregação: <100µs por transação
- Snapshot de estado: 10-50ms (dependente de RPC)

- Detecção de ataque: <5ms por grupo

6.2 Capacidade

- Grupos simultâneos: 1000 (configurável)
- Cache de bytecode: 1024 contratos
- Cache de estado RPC: 128 endereços
- Histórico de snapshots: 3 por endereço

6.3 Uso de Memória

- Base: ~10MB
- Por grupo ativo: ~1KB
- Cache total: ~50MB (limite configurável)
- Banco de dados: Crescimento linear com snapshots

7. Decisões de Design

7.1 Por que Detecção Passiva?

- Menor latência comparada a simulação
- Não requer estado completo da blockchain
- Funciona com RPCs limitados (incluindo BSC)
- Reduz custos de infraestrutura

7.2 Por que redb?

- Performance superior para workloads key-value
- Embarcado (não requer processo separado)
- ACID completo com overhead mínimo
- API ergonômica para Rust

7.3 Trade-offs

- Precisão vs Velocidade: Priorizamos velocidade com heurísticas
- Memória vs Disco: Cache agressivo com eviction policies
- Generalização vs Especialização: Foco em AMMs principais