Ethernity MEV Detector - Exemplos Práticos e Integração

1. Exemplo Completo: Análise de Bloco

rust

```rust
use std::env;
use std::sync::Arc;
use ethernity_detector_mev::*;
use ethernity_rpc::{EthernityRpcClient, RpcConfig};
use web3::types::{Block, Transaction};

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    // Configuração
    let endpoint = env::var("ETH_RPC_URL")?;
    let block_number: Option<u64> = env::args()
        .nth(1)
        .and_then(|s| s.parse().ok());

    // Cliente RPC
    let rpc_config = RpcConfig {
        endpoint,
        timeout: Duration::from_secs(30),
        max_retries: 3,
    };
    let rpc = Arc::new(EthernityRpcClient::new(rpc_config).await?);

    // Componentes
    let tagger = TxNatureTagger::new(rpc.clone());
    let mut aggregator = TxAggregator::new();
    let detector = AttackDetector::new(1.0, 10);

    // Determina bloco alvo
    let target_block = block_number
        .unwrap_or(rpc.get_block_number().await?);

    println!("Analisando bloco {}...", target_block);

    // Recupera bloco
    let block_bytes = rpc.get_block(target_block).await?;
    let block: Block<Transaction> = serde_json::from_slice(&block_bytes)?;

    // Processa transações
    for tx in block.transactions.iter() {
        if let Some(to_addr) = tx.to {
            let nature = tagger.analyze(
                to_addr,
                &tx.input.0,
                tx.hash
            ).await?;
```

```rust
            let annotated = AnnotatedTx {
                tx_hash: tx.hash,
                token_paths: nature.token_paths,
                targets: nature.targets,
                tags: nature.tags,
                first_seen: block.timestamp.as_u64(),
                gas_price: u256_to_f64(tx.gas_price.unwrap_or_default()),
                max_priority_fee_per_gas: tx.max_priority_fee_per_gas
                    .map(u256_to_f64),
                confidence: nature.confidence,
            };

            aggregator.add_tx(annotated);
        }
    }

    // Analisa grupos
    analyze_groups(&mut aggregator, &detector, rpc, target_block).await?;

    Ok(())
}

async fn analyze_groups<P: RpcProvider>(
    aggregator: &TxAggregator,
    detector: &AttackDetector,
    rpc: Arc<P>,
    block_number: u64,
) -> anyhow::Result<()> {
    // Repositório de snapshots
    let snapshot_dir = std::env::temp_dir().join("mev_detector_db");
    let repo = StateSnapshotRepository::open(rpc, &snapshot_dir)?;

    // Captura snapshots
    repo.snapshot_groups(
        aggregator.groups(),
        block_number,
        SnapshotProfile::Basic
    ).await?;

    // Avalia cada grupo
    let mut evaluator = StateImpactEvaluator::default();

    for group in aggregator.groups().values() {
        println!("\n========================================");
        println!("Grupo: {:x}", group.group_key);
        println!("Transações: {}", group.txs.len());
        println!("Tokens: {:?}", format_addresses(&group.token_paths));
```

```rust
        println!("Alvos: {:?}", format_addresses(&group.targets));

        if let Some(target) = group.targets.first() {
            if let Some(snapshot) = repo.get_state(
                *target,
                block_number,
                SnapshotProfile::Basic
            ) {
                // Cria vítimas fictícias para demonstração
                let victims = create_victim_inputs(group);

                // Avalia impacto
                let impact = evaluator.evaluate_group(
                    group,
                    &victims,
                    &snapshot
                );

                print_impact_report(&impact);

                // Detecta ataques
                if let Some(verdict) = detector.analyze_group(group) {
                    print_attack_report(&verdict);
                }
            }
        }
    }

    Ok(())
}

fn format_addresses(addrs: &[Address]) -> Vec<String> {
    addrs.iter()
        .map(|a| format!("0x{:x}", a))
        .collect()
}

fn create_victim_inputs(group: &TxGroup) -> Vec<VictimInput> {
    group.txs.iter()
        .map(|tx| VictimInput {
            tx_hash: tx.tx_hash,
            amount_in: 100.0, // Valor exemplo
            amount_out_min: 95.0, // 5% slippage
            token_behavior_unknown: false,
            flash_loan_amount: None,
        })
        .collect()
```

```rust
}

fn print_impact_report(impact: &GroupImpact) {
    println!("\n📊 Análise de Impacto:");
    println!("├─ Score de Oportunidade: {:.2}%", impact.opportunity_score * 100.0);
    println!("├─ Lucro Esperado (Backrun): {:.2}", impact.expected_profit_backrun);
    println!("├─ Confiança do Estado: {:.2}%", impact.state_confidence * 100.0);
    println!("└─ Certeza do Impacto: {:.2}%", impact.impact_certainty * 100.0);

    if !impact.victims.is_empty() {
        println!("\n📉 Vítimas Potenciais:");
        for (i, victim) in impact.victims.iter().enumerate() {
            println!("  Vítima #{}:", i + 1);
            println!("  ├─ TX: {:x}", victim.tx_hash);
            println!("  ├─ Slippage Tolerada: {:.2}%", victim.slippage_tolerated);
            println!("  └─ Slippage Ajustada: {:.2}%", victim.slippage_adjusted);
        }
    }
}

fn print_attack_report(verdict: &AttackVerdict) {
    println!("\n🚨 Ataques Detectados:");
    println!("├─ Confiança: {:.2}%", verdict.confidence * 100.0);
    println!("├─ Reconsiderável: {}", if verdict.reconsiderable { "Sim" } else { "Não"
    println!("└─ Tipos:");

    for attack in &verdict.attack_types {
        match attack {
            AttackType::Frontrun { justification } => {
                println!("   ├─ FRONTRUN: {}", justification);
            }
            AttackType::Sandwich { justification } => {
                println!("   ├─ SANDWICH: {}", justification);
            }
            AttackType::Backrun { justification } => {
                println!("   ├─ BACKRUN: {}", justification);
            }
            _ => {
                println!("   ├─ {:?}", attack);
            }
        }
    }
}

fn u256_to_f64(value: web3::types::U256) -> f64 {
    let val: u128 = value.into();
```

```
    val as f64
}
```

2. Exemplo: Monitor de Mempool em Tempo Real

rust

```rust
use tokio::sync::mpsc;
use tokio::time::{interval, Duration};
use futures::StreamExt;

struct MempoolMonitor<P> {
    supervisor: MempoolSupervisor<P>,
    ws_client: Option<WebSocketClient>,
}

impl<P: RpcProvider + Clone + 'static> MempoolMonitor<P> {
    pub async fn new(provider: P) -> Result<Self> {
        let supervisor = MempoolSupervisor::new(
            provider.clone(),
            2,         // min_tx_count
            Duration::from_secs(5), // dt_max
            1000,      // max_active_groups
        );

        Ok(Self {
            supervisor,
            ws_client: None,
        })
    }

    pub async fn run_with_websocket(
        mut self,
        ws_url: &str,
    ) -> Result<()> {
        // Conecta WebSocket
        let ws = WebSocketClient::connect(ws_url).await?;
        self.ws_client = Some(ws);

        // Canais
        let (tx_events, rx_events) = mpsc::channel(1024);
        let (tx_groups, mut rx_groups) = mpsc::channel(256);

        // Spawn supervisor
        tokio::spawn(async move {
            self.supervisor.process_stream(rx_events, tx_groups).await;
        });

        // Spawn monitor de blocos
        let tx_events_block = tx_events.clone();
        tokio::spawn(async move {
            let mut ticker = interval(Duration::from_secs(12));
            loop {
```

```
            ticker.tick().await;
            let _ = tx_events_block.send(
                SupervisorEvent::BlockAdvanced(BlockMetadata {
                    number: get_current_block().await,
                })
            ).await;
        }
    });

    // Processa stream de pendentes
    if let Some(ws) = &mut self.ws_client {
        let pending_stream = ws.subscribe_pending_txs().await?;

        tokio::pin!(pending_stream);

        while let Some(tx) = pending_stream.next().await {
            if let Ok(annotated) = self.process_pending_tx(tx).await {
                let _ = tx_events.send(
                    SupervisorEvent::NewTxObserved(annotated)
                ).await;
            }
        }
    }

    // Processa grupos prontos
    while let Some(group_ready) = rx_groups.recv().await {
        self.handle_group_ready(group_ready).await?;
    }

    Ok(())
}

async fn process_pending_tx(
    &self,
    raw: RawTransaction,
) -> Result<AnnotatedTx> {
    let tagger = TxNatureTagger::new(self.provider.clone());

    let nature = tagger.analyze(
        raw.to,
        &raw.input,
        raw.hash
    ).await?;

    Ok(AnnotatedTx {
        tx_hash: raw.hash,
        token_paths: nature.token_paths,
```

```
            targets: nature.targets,
            tags: nature.tags,
            first_seen: timestamp_now(),
            gas_price: raw.gas_price,
            max_priority_fee_per_gas: raw.max_priority_fee,
            confidence: nature.confidence,
        })
    }

    async fn handle_group_ready(
        &self,
        ready: GroupReady,
    ) -> Result<()> {
        println!("\n🔔 Grupo Pronto para Análise");
        println!("Window ID: {}", ready.metadata.window_id);
        println!("Alinhamento: {:.2}%", ready.metadata.state_alignment_score * 100.0);

        // Aqui você pode:
        // 1. Enviar para sistema de execução
        // 2. Armazenar em banco de dados
        // 3. Emitir alertas
        // 4. Calcular estratégia de MEV

        if ready.metadata.state_alignment_score > 0.8 {
            execute_mev_strategy(&ready.group).await?;
        }

        Ok(())
    }
}

async fn execute_mev_strategy(group: &TxGroup) -> Result<()> {
    // Implementação específica da estratégia
    println!("Executando estratégia para grupo {:x}", group.group_key);
    Ok(())
}
```

3. Exemplo: Bot MEV Defensivo

rust

```rust
/// Bot que monitora transações do usuário e alerta sobre possíveis ataques
struct DefensiveMevBot {
    detector: AttackDetector,
    user_addresses: HashSet<Address>,
    alert_channel: mpsc::Sender<MevAlert>,
}

#[derive(Debug)]
struct MevAlert {
    user_tx: TransactionHash,
    attack_type: AttackType,
    confidence: f64,
    recommendation: String,
}

impl DefensiveMevBot {
    pub fn new(
        user_addresses: Vec<Address>,
        alert_channel: mpsc::Sender<MevAlert>,
    ) -> Self {
        Self {
            detector: AttackDetector::new(1.0, 10),
            user_addresses: user_addresses.into_iter().collect(),
            alert_channel,
        }
    }

    pub async fn monitor_group(&self, group: &TxGroup) -> Result<()> {
        // Verifica se alguma transação é do usuário
        let user_txs: Vec<_> = group.txs.iter()
            .filter(|tx| self.is_user_transaction(tx))
            .collect();

        if user_txs.is_empty() {
            return Ok(());
        }

        // Analisa ataques
        if let Some(verdict) = self.detector.analyze_group(group) {
            for user_tx in user_txs {
                self.analyze_user_risk(user_tx, &verdict).await?;
            }
        }

        Ok(())
    }
```

```rust
    fn is_user_transaction(&self, tx: &AnnotatedTx) -> bool {
        tx.targets.iter()
            .any(|target| self.user_addresses.contains(target))
    }

    async fn analyze_user_risk(
        &self,
        user_tx: &AnnotatedTx,
        verdict: &AttackVerdict,
    ) -> Result<()> {
        for attack in &verdict.attack_types {
            let recommendation = match attack {
                AttackType::Frontrun { .. } => {
                    "Considere aumentar o gas price ou usar flashbots"
                }
                AttackType::Sandwich { .. } => {
                    "Reduza o slippage máximo ou divida a transação"
                }
                AttackType::Backrun { .. } => {
                    "Transação pode ser seguida, considere MEV protection"
                }
                _ => "Verifique configurações de segurança"
            };

            let alert = MevAlert {
                user_tx: user_tx.tx_hash,
                attack_type: attack.clone(),
                confidence: verdict.confidence,
                recommendation: recommendation.to_string(),
            };

            self.alert_channel.send(alert).await?;
        }

        Ok(())
    }
}
```

4. Exemplo: Análise Forense

rust

```rust
/// Analisa histórico de MEV em range de blocos
struct MevForensics<P> {
    provider: Arc<P>,
    repo: StateSnapshotRepository<P>,
}

impl<P: RpcProvider + Clone> MevForensics<P> {
    pub async fn analyze_range(
        &self,
        start_block: u64,
        end_block: u64,
    ) -> ForensicsReport {
        let mut report = ForensicsReport::default();

        for block_num in start_block..=end_block {
            if let Ok(stats) = self.analyze_block(block_num).await {
                report.add_block_stats(block_num, stats);
            }
        }

        report
    }

    async fn analyze_block(&self, block_num: u64) -> Result<BlockMevStats> {
        let mut stats = BlockMevStats::default();

        // Recupera e processa bloco
        let block = fetch_block(self.provider.clone(), block_num).await?;
        let groups = extract_groups(&block).await?;

        // Snapshot de estado
        self.repo.snapshot_groups(
            &groups,
            block_num,
            SnapshotProfile::Extended
        ).await?;

        // Analisa cada grupo
        let detector = AttackDetector::new(1.0, 10);
        let mut evaluator = StateImpactEvaluator::default();

        for group in groups.values() {
            // Detecta ataques
            if let Some(verdict) = detector.analyze_group(group) {
                stats.attacks_detected += 1;
                stats.total_confidence += verdict.confidence;
```

```
                for attack_type in verdict.attack_types {
                    *stats.attack_distribution
                        .entry(format!("{:?}", attack_type))
                        .or_insert(0) += 1;
                }
            }

            // Calcula impacto
            if let Some(target) = group.targets.first() {
                if let Some(snapshot) = self.repo.get_state(
                    *target,
                    block_num,
                    SnapshotProfile::Extended
                ) {
                    let impact = evaluator.evaluate_group(
                        group,
                        &[],
                        &snapshot
                    );

                    stats.total_opportunity_score += impact.opportunity_score;
                    stats.total_expected_profit += impact.expected_profit_backrun;
                }
            }
        }

        stats.groups_analyzed = groups.len();
        Ok(stats)
    }
}

#[derive(Default)]
struct ForensicsReport {
    blocks_analyzed: usize,
    total_attacks: usize,
    attack_types: HashMap<String, usize>,
    profit_distribution: Vec<f64>,
    high_risk_addresses: HashSet<Address>,
}

impl ForensicsReport {
    fn add_block_stats(&mut self, block: u64, stats: BlockMevStats) {
        self.blocks_analyzed += 1;
        self.total_attacks += stats.attacks_detected;

        for (attack_type, count) in stats.attack_distribution {
```

```
            *self.attack_types.entry(attack_type).or_insert(0) += count;
        }

        if stats.total_expected_profit > 0.0 {
            self.profit_distribution.push(stats.total_expected_profit);
        }
    }

    fn generate_summary(&self) -> String {
        format!(
            r#"
MEV Forensics Report
====================
Blocks Analyzed: {}
Total Attacks Detected: {}
Average Attacks per Block: {:.2}

Attack Distribution:
{:?}

Profit Statistics:
- Total Opportunities: {}
- Average Profit: {:.2}
- Max Profit: {:.2}
            "#,
            self.blocks_analyzed,
            self.total_attacks,
            self.total_attacks as f64 / self.blocks_analyzed as f64,
            self.attack_types,
            self.profit_distribution.len(),
            self.profit_distribution.iter().sum::<f64>() / self.profit_distribution.ler
            self.profit_distribution.iter().max_by(|a, b| a.partial_cmp(b).unwrap()).ur
        )
    }
}
```

5. Integração com Sistemas Externos

5.1 Integração com Flashbots

rust

```rust
use flashbots::{BundleRequest, FlashbotsMiddleware};

async fn submit_mev_bundle(
    group: &TxGroup,
    impact: &GroupImpact,
    flashbots: &FlashbotsMiddleware,
) -> Result<()> {
    // Constrói bundle baseado na análise
    let mut bundle = BundleRequest::new();

    // Adiciona transação de captura
    let capture_tx = build_capture_transaction(impact)?;
    bundle.add_transaction(capture_tx);

    // Adiciona transações originais se necessário
    for victim in &impact.victims {
        if should_include_victim(victim) {
            bundle.add_transaction_hash(victim.tx_hash);
        }
    }

    // Envia bundle
    let response = flashbots
        .send_bundle(&bundle)
        .await?;

    println!("Bundle enviado: {:?}", response);
    Ok(())
}
```

5.2 Webhook para Alertas

rust

```rust
use reqwest::Client;
use serde_json::json;

struct WebhookNotifier {
    client: Client,
    webhook_url: String,
}

impl WebhookNotifier {
    async fn notify_attack(
        &self,
        verdict: &AttackVerdict,
        group: &TxGroup,
    ) -> Result<()> {
        let payload = json!({
            "type": "mev_attack_detected",
            "timestamp": chrono::Utc::now().to_rfc3339(),
            "group_key": format!("{:x}", verdict.group_key),
            "confidence": verdict.confidence,
            "attacks": verdict.attack_types.iter()
                .map(|a| format!("{:?}", a))
                .collect::<Vec<_>>(),
            "transaction_count": group.txs.len(),
            "tokens": group.token_paths.iter()
                .map(|a| format!("{:x}", a))
                .collect::<Vec<_>>(),
        });

        self.client
            .post(&self.webhook_url)
            .json(&payload)
            .send()
            .await?;

        Ok(())
    }
}
```

5.3 Exportação para Analytics

rust

```rust
use csv::Writer;
use std::fs::File;

fn export_mev_data(
    groups: &HashMap<H256, TxGroup>,
    impacts: &HashMap<H256, GroupImpact>,
    output_path: &str,
) -> Result<()> {
    let file = File::create(output_path)?;
    let mut writer = Writer::from_writer(file);

    // Headers
    writer.write_record(&[
        "group_id",
        "timestamp",
        "tx_count",
        "tokens",
        "opportunity_score",
        "expected_profit",
        "attack_detected",
        "attack_types",
    ])?;

    // Data
    for (group_id, group) in groups {
        let impact = impacts.get(group_id);
        let detector = AttackDetector::new(1.0, 10);
        let verdict = detector.analyze_group(group);

        writer.write_record(&[
            format!("{:x}", group_id),
            group.txs.first()
                .map(|t| t.first_seen.to_string())
                .unwrap_or_default(),
            group.txs.len().to_string(),
            group.token_paths.len().to_string(),
            impact.map(|i| i.opportunity_score.to_string())
                .unwrap_or_default(),
            impact.map(|i| i.expected_profit_backrun.to_string())
                .unwrap_or_default(),
            verdict.is_some().to_string(),
            verdict.map(|v| format!("{:?}", v.attack_types))
                .unwrap_or_default(),
        ])?;
    }
```

```
    writer.flush()?;
    Ok(())
}
```