Module 5: Basic Sandwich Attack Implementation

Duration: 95 minutes | **Level:** Intermediate-Advanced | **Author:** Obelisk Core

Learning Objectives

By the end of this module, you will be able to:
- Understand sandwich attack mechanics and profitability factors
- Build systems to detect sandwichable transactions in the mempool
- Implement profitable sandwich attack strategies
- Manage risks and navigate ethical considerations
- Handle complex scenarios and competitive environments

Understanding Sandwich Attacks

Attack Mechanics

A sandwich attack is a three-step MEV strategy that exploits predictable price movements:

1. **Front-Run**: Detect a large trade about to be executed
2. **Trade**: Execute your own trade to move the price in your favor
3. **Back-Run**: Execute the victim's trade (which moves price further)
4. **Exit**: Trade back to original position, capturing the spread

Mathematical Foundation:

```
Price Impact = f(trade_size, liquidity)
Sandwich Profit = (Price_Impact × Trade_Size) - Gas_Costs -
Competition_Costs
```

Attack Prerequisites

Technical Requirements:
- Real-time mempool monitoring
- Fast transaction execution
- Sufficient capital for intermediate trades
- Prediction accuracy for price movements

Market Conditions:
- Sufficient liquidity to execute trades
- Predictable price impact from victim trades
- Network conditions favorable for rapid execution
- Limited competition from other MEV bots

Profitability Analysis

Example Scenario:

```
Victim Trade: Buy 100 ETH on Uniswap
Current Pool: 1000 ETH / 200,000 USDC
Expected Price Impact: 4.76%

Attack Sequence:
1. Buy 50 ETH (moves price to 1.0547 ETH/USDC)
2. Victim buys 100 ETH (moves price to 1.1105 ETH/USDC)
3. Sell 50 ETH (price returns partially to 1.0813 ETH/USDC)

Expected Profit: ~$160 minus gas costs
```

Building Detection Systems

Mempool Monitoring

```
import asyncio
import websockets
import json
from typing import Dict, List, Optional, Callable
from dataclasses import dataclass
from datetime import datetime, timedelta

@dataclass
class Transaction:
    hash: str
    from_address: str
    to_address: str
    value: int
    gas_price: int
    gas_limit: int
    data: str
    timestamp: datetime
    estimated_gas_used: int = 0

@dataclass
class SandwichOpportunity:
    victim_tx: Transaction
    predicted_impact: float
    estimated_profit: float
    execution_complexity: str
    risk_level: str
    detection_confidence: float

class MempoolMonitor:
    def __init__(self, config: Dict):
        self.config = config
        self.etherscan_api_key = config.get('etherscan_api_key')
        self.alchemy_api_key = config.get('alchemy_api_key')
        self.infura_project_id = config.get('infura_project_id')

        # WebSocket connections for real-time monitoring
        self.ws_connections = []

        # Transaction tracking
        self.pending_transactions = {}
        self.processed_hashes = set()
        self.sandwich_opportunities = []
```

```
        # DEX router addresses for filtering
        self.dex_routers = {
            'uniswap_v2': '0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',
            'uniswap_v3': '0xE592427A0AEce92De3Edee1F18E0157C05861564',
            'sushiswap': '0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F',
            '1inch': '0x1111111254EEB25477B68fb85Ed929f73A960582'
        }

        # Filter criteria
        self.min_profit_threshold = config.get('min_profit_threshold',
50)  # $50 minimum
        self.max_gas_price = config.get('max_gas_price', 100e9)  # 100
gwei max
        self.min_transaction_value =
config.get('min_transaction_value', 100000)  # $100k min

    async def start_monitoring(self):
        """Start real-time mempool monitoring"""

        tasks = [
            asyncio.create_task(self.monitor_websockets()),
            asyncio.create_task(self.monitor_pending_transactions()),
            asyncio.create_task(self.analyze_sandwich_opportunities()),
            asyncio.create_task(self.cleanup_old_transactions())
        ]

        await asyncio.gather(*tasks)

    async def monitor_websockets(self):
        """Monitor transactions via WebSocket connections"""

        # Alchemy WebSocket
        alchemy_ws = f"wss://eth-mainnet.alchemyapi.io/v2/
{self.alchemy_api_key}"

        # Infura WebSocket
        infura_ws = f"wss://mainnet.infura.io/ws/v3/
{self.infura_project_id}"

        websocket_urls = [alchemy_ws, infura_ws]
```

```
        for ws_url in websocket_urls:
            try:
                async with websockets.connect(ws_url) as websocket:
                    # Subscribe to new pending transactions
                    subscribe_msg = {
                        "id": 1,
                        "method": "eth_subscribe",
                        "params": ["newPendingTransactions",
{"threshold": 1}]
                    }

                    await websocket.send(json.dumps(subscribe_msg))

                    async for message in websocket:
                        try:
                            data = json.loads(message)

                            if 'params' in data and 'result' in
data['params']:
                                tx_hash = data['params']['result']
                                await
self.process_pending_transaction(tx_hash)

                        except json.JSONDecodeError:
                            continue
                        except Exception as e:

logging.error(f"WebSocket processing error: {e}")

            except Exception as e:
                logging.error(f"WebSocket connection failed for
{ws_url}: {e}")
                await asyncio.sleep(5)  # Reconnect delay

    async def process_pending_transaction(self, tx_hash: str):
        """Process a newly detected pending transaction"""

        if tx_hash in self.processed_hashes:
            return

        try:
            # Get transaction details from JSON-RPC
```

```
            tx_details = await self.get_transaction_details(tx_hash)

            if not tx_details:
                return

            # Create Transaction object
            tx = Transaction(
                hash=tx_hash,
                from_address=tx_details['from'],
                to_address=tx_details.get('to', ''),
                value=int(tx_details.get('value', '0'), 16),
                gas_price=int(tx_details.get('gasPrice', '0'), 16),
                gas_limit=int(tx_details.get('gas', '0'), 16),
                data=tx_details.get('input', '0x'),
                timestamp=datetime.now()
            )

            # Filter for potential sandwich opportunities
            if await self.is_sandwich_candidate(tx):
                self.pending_transactions[tx_hash] = tx
                await self.analyze_sandwich_potential(tx)

            self.processed_hashes.add(tx_hash)

        except Exception as e:
            logging.error(f"Error processing transaction {tx_hash}:
{e}")

    async def get_transaction_details(self, tx_hash: str) ->
Optional[Dict]:
        """Get transaction details using JSON-RPC"""

        # Try multiple providers
        providers = [
            f"https://eth-mainnet.alchemyapi.io/v2/
{self.alchemy_api_key}",
            f"https://mainnet.infura.io/ws/v3/{self.infura_project_id}"
        ]

        for provider in providers:
            try:
                payload = {
```

```
                    "jsonrpc": "2.0",
                    "method": "eth_getTransactionByHash",
                    "params": [tx_hash],
                    "id": 1
                }

                async with aiohttp.ClientSession() as session:
                    async with session.post(provider, json=payload) as
response:
                        if response.status == 200:
                            data = await response.json()
                            return data.get('result')

            except Exception as e:
                logging.warning(f"Provider {provider} failed: {e}")
                continue

        return None

    async def is_sandwich_candidate(self, tx: Transaction) -> bool:
        """Determine if transaction is a good sandwich candidate"""

        # Basic filters
        if tx.value < self.min_transaction_value:
            return False

        if tx.gas_price > self.max_gas_price:
            return False

        # Check if transaction targets DEX router
        if tx.to_address not in self.dex_routers.values():
            return False

        # Decode transaction data to check if it's a swap
        if not self.is_swap_transaction(tx):
            return False

        # Check transaction complexity (simulate to estimate gas)
        estimated_gas = await self.estimate_transaction_gas(tx)
        if estimated_gas > 500000:  # Too complex for sandwich
            return False
```

```
        return True

    def is_swap_transaction(self, tx: Transaction) -> bool:
        """Check if transaction is a token swap"""

        # Simple check - transaction data starts with function selector
        # For Uniswap V2: transfer or swapExactETHForTokens,
swapExactTokensForETH, etc.
        if not tx.data or len(tx.data) < 10:
            return False

        # Get function selector (first 4 bytes)
        function_selector = tx.data[:10]

        # Common DEX function selectors
        swap_selectors = [
            '0x38ed1739',  # swapExactTokensForTokens
            '0x7ff36ab5',  # swapExactETHForTokens
            '0x18cbafe5',  # swapExactTokensForETH
            '0xfb3bdb41',  # swapExactETHForTokens supporting fee on
transfer
            '0x5c11d795',  #
swapExactTokensForTokensSupportingFeeOnTransferTokens
            '0x8803dbee',  #
swapExactTokensForETHSupportingFeeOnTransferTokens
        ]

        return function_selector in swap_selectors

    async def estimate_transaction_gas(self, tx: Transaction) -> int:
        """Estimate gas usage for transaction"""

        # This is a simplified estimation
        # In practice, you'd use more sophisticated methods

        base_gas = 21000
        data_gas = len(tx.data) * 68  # 68 gas per byte of data

        # Estimate based on transaction type
        if 'swap' in tx.data.lower():
            contract_gas = 200000  # Typical for swap
        else:
```

```
            contract_gas = 50000   # Other operations

        return base_gas + data_gas + contract_gas

    async def analyze_sandwich_potential(self, tx: Transaction):
        """Analyze sandwich attack potential for transaction"""

        try:
            # Simulate transaction to understand its effect
            simulation_result = await self.simulate_transaction(tx)

            if simulation_result['success']:
                # Calculate sandwich profit potential
                sandwich_analysis = await
self.calculate_sandwich_profit(tx, simulation_result)

                if sandwich_analysis['estimated_profit'] >=
self.min_profit_threshold:
                    opportunity = SandwichOpportunity(
                        victim_tx=tx,

predicted_impact=sandwich_analysis['predicted_impact'],

estimated_profit=sandwich_analysis['estimated_profit'],

execution_complexity=sandwich_analysis['complexity'],
                        risk_level=sandwich_analysis['risk_level'],

detection_confidence=sandwich_analysis['confidence']
                    )

                    self.sandwich_opportunities.append(opportunity)
                    logging.info(f"Sandwich opportunity detected: $
{sandwich_analysis['estimated_profit']:.2f} profit")

        except Exception as e:
            logging.error(f"Error analyzing sandwich potential for
{tx.hash}: {e}")

    async def simulate_transaction(self, tx: Transaction) -> Dict:
        """Simulate transaction to understand its effects"""
```

```
        # This would use tools like Tenderly or custom simulation
        # For now, return a simplified simulation result

        # Parse transaction data
        swap_details = self.parse_swap_transaction(tx)

        if not swap_details:
            return {'success': False, 'error': 'Not a valid swap
transaction'}

        # Simulate the swap's effect on AMM pools
        pool_effects = await self.simulate_pool_impact(swap_details)

        return {
            'success': True,
            'swap_details': swap_details,
            'pool_effects': pool_effects,
            'estimated_price_impact': pool_effects.get('total_impact',
0),
            'estimated_gas_used': tx.estimated_gas_used
        }

    def parse_swap_transaction(self, tx: Transaction) ->
Optional[Dict]:
        """Parse swap transaction to extract details"""

        if not tx.data or len(tx.data) < 10:
            return None

        function_selector = tx.data[:10]

        # For simplicity, focus on Uniswap V2 style swaps
        if function_selector == '0x38ed1739':  #
swapExactTokensForTokens
            return self.parse_tokens_for_tokens_swap(tx.data)
        elif function_selector == '0x7ff36ab5':  #
swapExactETHForTokens
            return self.parse_eth_for_tokens_swap(tx.data)
        elif function_selector == '0x18cbafe5':  #
swapExactTokensForETH
            return self.parse_tokens_for_eth_swap(tx.data)
```

```
        return None

    def parse_tokens_for_tokens_swap(self, data: str) -> Dict:
        """Parse swapExactTokensForTokens transaction"""

        try:

# Simplified parsing - real implementation would use proper ABI
decoding
            # Expected format: function_selector + encoded parameters

            # Extract amounts (simplified)
            # In practice, use eth_abi library for proper decoding
            amount_in = int(data[10:74], 16)  # Simplified extraction
            amount_out_min = int(data[74:138], 16)  # Simplified
extraction

            # Extract path (simplified)
            # Real implementation would decode the path array
            path_length = (len(data) - 138) // 64  # Simplified
            tokens = []
            for i in range(path_length):
                token_start = 138 + (i * 64)
                token = data[token_start:token_start + 64]
                if token != '0' * 64:  # Non-zero address
                    tokens.append('0x' + token[24:])  # Remove padding

            return {
                'type': 'tokens_for_tokens',
                'amount_in': amount_in,
                'amount_out_min': amount_out_min,
                'path': tokens,
                'deadline': int(data[-8:], 16) if len(data) > 138 else
0
            }

        except Exception as e:
            logging.error(f"Error parsing tokens swap: {e}")
            return None

    async def calculate_sandwich_profit(self, tx: Transaction,
simulation: Dict) -> Dict:
```

```
        """Calculate potential sandwich attack profit"""

        swap_details = simulation['swap_details']
        predicted_impact = simulation['estimated_price_impact']

        # Calculate optimal sandwich trade size
        # Typically 50-80% of victim trade size
        victim_size = swap_details['amount_in']
        optimal_sandwich_size = victim_size * 0.6  # Conservative 60%

        # Calculate price movements
        price_movement_1 =
self.calculate_price_impact(optimal_sandwich_size,
swap_details['path'])
        price_movement_2 = self.calculate_price_impact(victim_size +
optimal_sandwich_size, swap_details['path'])
        price_movement_3 =
self.calculate_price_impact(optimal_sandwich_size,
swap_details['path'])

        # Calculate profit from each step
        step_1_profit = optimal_sandwich_size * price_movement_1  #
Front-run profit
        step_2_profit = optimal_sandwich_size * (price_movement_2 -
price_movement_1)  # Main profit
        step_3_profit = optimal_sandwich_size * price_movement_3  #
Back-run profit

        total_gross_profit = step_1_profit + step_2_profit +
step_3_profit

        # Subtract costs
        gas_cost_estimate = self.estimate_sandwich_gas_cost()
        slippage_cost = total_gross_profit * 0.1  # 10% slippage
estimate
        competition_cost = total_gross_profit * 0.05  # 5% competition
cost

        total_costs = gas_cost_estimate + slippage_cost +
competition_cost
        net_profit = total_gross_profit - total_costs
```

```
        # Assess complexity and risk
        complexity = self.assess_execution_complexity(swap_details)
        risk_level = self.assess_sandwich_risk(predicted_impact,
net_profit)
        confidence = self.calculate_detection_confidence(simulation)

        return {
            'estimated_profit': net_profit,
            'gross_profit': total_gross_profit,
            'total_costs': total_costs,
            'predicted_impact': predicted_impact,
            'optimal_size': optimal_sandwich_size,
            'complexity': complexity,
            'risk_level': risk_level,
            'confidence': confidence,
            'profit_breakdown': {
                'front_run': step_1_profit,
                'main_profit': step_2_profit,
                'back_run': step_3_profit
            }
        }

    def calculate_price_impact(self, trade_size: int, token_path:
List[str]) -> float:
        """Calculate price impact for a trade"""

        # This is a simplified calculation
        # Real implementation would analyze actual AMM pools

        # Assume 0.3% fee and typical pool sizes
        pool_liquidity = 1000000  # $1M liquidity assumption
        fee_rate = 0.003

        # Price impact formula for constant product AMM
        price_impact = (trade_size / pool_liquidity) * fee_rate

        return price_impact

    def estimate_sandwich_gas_cost(self) -> float:
        """Estimate total gas cost for sandwich attack"""

        # Sandwich typically requires 3-4 transactions
```

```
        gas_per_tx = 300000
        total_gas = gas_per_tx * 3.5  # Average 3.5 transactions

        # Assume average gas price of 50 gwei
        gas_price = 50e9
        gas_cost_wei = total_gas * gas_price

        # Convert to USD (assuming ETH price of $2000)
        eth_price = 2000
        gas_cost_usd = (gas_cost_wei * eth_price) / 1e18

        return gas_cost_usd

    def assess_execution_complexity(self, swap_details: Dict) -> str:
        """Assess execution complexity of sandwich attack"""

        path_length = len(swap_details.get('path', []))

        if path_length > 3:
            return 'HIGH'
        elif path_length > 2:
            return 'MEDIUM'
        else:
            return 'LOW'

    def assess_sandwich_risk(self, predicted_impact: float, profit:
float) -> str:
        """Assess risk level of sandwich attack"""

        # High impact or low profit = higher risk
        if predicted_impact > 0.05 or profit < 100:
            return 'HIGH'
        elif predicted_impact > 0.02 or profit < 200:
            return 'MEDIUM'
        else:
            return 'LOW'

    def calculate_detection_confidence(self, simulation: Dict) ->
float:
        """Calculate confidence in sandwich opportunity detection"""

        base_confidence = 0.7
```

```
        # Adjust based on simulation success
        if not simulation['success']:
            return 0.3

        # Adjust based on estimated impact
        impact = simulation.get('estimated_price_impact', 0)
        if impact > 0.03:
            base_confidence += 0.2
        elif impact > 0.01:
            base_confidence += 0.1

        return min(1.0, base_confidence)
```

Advanced Detection Algorithms

```
class SandwichDetector:
    def __init__(self, config: Dict):
        self.config = config
        self.detection_models = {
            'pattern_matching': PatternMatchingDetector(),
            'ml_classifier': MLClassifierDetector(),
            'anomaly_detector': AnomalyDetector()
        }

    async def detect_opportunities_batch(self, transactions:
List[Transaction]) -> List[SandwichOpportunity]:
        """Detect sandwich opportunities using multiple detection
methods"""

        all_opportunities = []

        # Run each detection method in parallel
        detection_results = await asyncio.gather(*[
            detector.detect(transactions)
            for detector in self.detection_models.values()
        ])

        # Combine and rank opportunities
        combined_opportunities = []
        for results in detection_results:
            combined_opportunities.extend(results)

        # Remove duplicates and rank by profit potential
        unique_opportunities =
self.remove_duplicates(combined_opportunities)
        ranked_opportunities =
self.rank_opportunities(unique_opportunities)

        return ranked_opportunities

    def remove_duplicates(self, opportunities:
List[SandwichOpportunity]) -> List[SandwichOpportunity]:
        """Remove duplicate sandwich opportunities"""

        seen_hashes = set()
        unique_opportunities = []
```

```
        for opp in opportunities:
            tx_hash = opp.victim_tx.hash
            if tx_hash not in seen_hashes:
                seen_hashes.add(tx_hash)
                unique_opportunities.append(opp)

        return unique_opportunities

    def rank_opportunities(self, opportunities:
List[SandwichOpportunity]) -> List[SandwichOpportunity]:
        """Rank opportunities by profitability and execution
likelihood"""

        def opportunity_score(opp: SandwichOpportunity) -> float:
            # Profit score (normalized)
            profit_score = min(100, opp.estimated_profit / 10)

            # Confidence score
            confidence_score = opp.detection_confidence * 50

            # Risk penalty
            risk_penalty = {'LOW': 0, 'MEDIUM': -10, 'HIGH': -25}
[opp.risk_level]

            # Complexity penalty
            complexity_penalty = {'LOW': 0, 'MEDIUM': -5, 'HIGH': -15}
[opp.execution_complexity]

            return profit_score + confidence_score + risk_penalty +
complexity_penalty

        return sorted(opportunities, key=opportunity_score,
reverse=True)

class MLClassifierDetector:
    """Machine learning-based sandwich detection"""

    def __init__(self):
        self.model = None  # Would load trained model in production
        self.feature_extractor = SandwichFeatureExtractor()

    async def detect(self, transactions: List[Transaction]) ->
```

```
List[SandwichOpportunity]:
        """Use ML model to detect sandwich opportunities"""

        opportunities = []

        for tx in transactions:
            # Extract features for ML model
            features = await
self.feature_extractor.extract_features(tx)

            # Get model prediction
            prediction = await
self.predict_sandwich_probability(features)

            if prediction['sandwich_probability'] > 0.7:  # High
confidence threshold
                opportunity = SandwichOpportunity(
                    victim_tx=tx,
                    predicted_impact=prediction['predicted_impact'],
                    estimated_profit=prediction['estimated_profit'],
                    execution_complexity=prediction['complexity'],
                    risk_level=prediction['risk_level'],

detection_confidence=prediction['sandwich_probability']
                )
                opportunities.append(opportunity)

        return opportunities

    async def predict_sandwich_probability(self, features: Dict) ->
Dict:
        """Use ML model to predict sandwich probability"""

        # This would use a trained ML model
        # For now, implement a simplified rule-based approach

        probability = 0.5  # Base probability

        # Adjust based on features
        if features.get('gas_price', 0) > 100e9:  # High gas price
            probability += 0.2
```

```
        if features.get('transaction_value', 0) > 500000:  # Large
transaction
            probability += 0.2

        if features.get('is_dex_swap', False):
            probability += 0.3

        # Simulate other model outputs
        return {
            'sandwich_probability': min(1.0, probability),
            'predicted_impact': features.get('estimated_impact', 0.01),
            'estimated_profit': features.get('transaction_value', 0) *
0.001,  # 0.1% estimate
            'complexity': 'MEDIUM' if features.get('has_path', False)
else 'LOW',
            'risk_level': 'MEDIUM' if probability > 0.7 else 'LOW'
        }

class SandwichFeatureExtractor:
    """Extract features for sandwich detection ML models"""

    async def extract_features(self, tx: Transaction) -> Dict:
        """Extract features from transaction for ML model"""

        features = {
            'gas_price': tx.gas_price,
            'gas_limit': tx.gas_limit,
            'transaction_value': tx.value,
            'data_length': len(tx.data),
            'timestamp': tx.timestamp.timestamp(),
            'is_dex_swap': self.is_dex_swap(tx),
            'estimated_impact': await self.estimate_price_impact(tx)
        }

        # Extract path information if available
        if features['is_dex_swap']:
            path_info = self.extract_swap_path(tx.data)
            features.update({
                'has_path': len(path_info.get('path', [])) > 0,
                'path_length': len(path_info.get('path', [])),
                'token_in': path_info.get('token_in'),
                'token_out': path_info.get('token_out')
```

```
            })

        return features

    def is_dex_swap(self, tx: Transaction) -> bool:
        """Check if transaction is a DEX swap"""

        if not tx.to_address or not tx.data:
            return False

        # Check against known DEX routers
        dex_routers = [
            '0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',  # Uniswap V2
            '0xE592427A0AEce92De3Edee1F18E0157C05861564',  # Uniswap V3
            '0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F',  # SushiSwap
        ]

        return tx.to_address.lower() in [router.lower() for router in
dex_routers]

    async def estimate_price_impact(self, tx: Transaction) -> float:
        """Estimate price impact of transaction"""

        if not self.is_dex_swap(tx):
            return 0.0

        # Simple estimation based on transaction value
        # In practice, this would analyze actual pool reserves

        tx_value_usd = tx.value * 1e-18 * 2000  # Assuming ETH price

        # Estimate liquidity (simplified)
        estimated_liquidity = 10000000  # $10M estimated liquidity

        # Rough price impact estimate
        impact = (tx_value_usd / estimated_liquidity) * 0.003  # 0.3%
fee

        return min(0.1, impact)  # Cap at 10%

    def extract_swap_path(self, data: str) -> Dict:
        """Extract swap path from transaction data"""
```

```
        if len(data) < 10:
            return {}

        function_selector = data[:10]

        if function_selector == '0x38ed1739':  #
swapExactTokensForTokens
            return self.extract_tokens_path(data)
        elif function_selector == '0x7ff36ab5':  #
swapExactETHForTokens
            return self.extract_eth_to_tokens_path(data)
        elif function_selector == '0x18cbafe5':  #
swapExactTokensForETH
            return self.extract_tokens_to_eth_path(data)

        return {}

    def extract_tokens_path(self, data: str) -> Dict:
        """Extract path from swapExactTokensForTokens"""

        try:
            # Simplified parsing
            amount_in = int(data[10:74], 16)
            amount_out_min = int(data[74:138], 16)

            # Extract path (simplified)
            # Real implementation would use proper ABI decoding
            path_start = 138
            path_length = (len(data) - path_start - 8) // 64  # Remove
deadline

            path = []
            for i in range(path_length):
                token_start = path_start + (i * 64)
                token_end = token_start + 64
                if token_end <= len(data) - 8:  # Account for deadline
                    token = data[token_start + 24:token_end]  # Remove
padding
                    if token != '0' * 40:
                        path.append('0x' + token)
```

```
            return {
                'token_in': path[0] if len(path) > 0 else None,
                'token_out': path[-1] if len(path) > 1 else None,
                'path': path,
                'amount_in': amount_in,
                'amount_out_min': amount_out_min
            }

        except Exception as e:
            logging.error(f"Error extracting tokens path: {e}")
            return {}
```

Implementing Sandwich Attacks

Execution Framework

```
class SandwichExecutor:
    def __init__(self, config: Dict):
        self.config = config
        self.private_key = config['private_key']
        self.w3 = Web3(Web3.HTTPProvider(config['web3_provider']))
        self.account = self.w3.eth.account.from_key(self.private_key)

        # DEX router contracts
        self.routers = {
            'uniswap_v2': self.w3.eth.contract(
                address='0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D',
                abi=self.get_uniswap_v2_abi()
            ),
            'sushiswap': self.w3.eth.contract(
                address='0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F',
                abi=self.get_uniswap_v2_abi()
            )
        }

        # Strategy configuration
        self.max_attempts = config.get('max_attempts', 3)
        self.gas_buffer = config.get('gas_buffer', 1.2)  # 20% buffer
        self.profit_threshold = config.get('profit_threshold', 50)  #
$50 minimum

    async def execute_sandwich(self, opportunity: SandwichOpportunity)
-> Dict:
        """Execute sandwich attack for detected opportunity"""

        try:
            logging.info(f"Starting sandwich execution for
{opportunity.victim_tx.hash}")

            # Step 1: Pre-execution validation
            validation_result = await
self.validate_sandwich_opportunity(opportunity)
            if not validation_result['valid']:
                return {'success': False, 'error':
validation_result['error']}

            # Step 2: Execute the sandwich
            execution_result = await
```

```
self.execute_sandwich_sequence(opportunity)

            if execution_result['success']:
                # Step 3: Analyze results
                analysis = await
self.analyze_execution_results(execution_result)
                return {
                    'success': True,
                    'execution_result': execution_result,
                    'analysis': analysis,
                    'profit': analysis.get('net_profit', 0),
                    'execution_time':
analysis.get('total_execution_time', 0)
                }
            else:
                return {
                    'success': False,
                    'error': execution_result.get('error', 'Execution
failed'),
                    'stage': execution_result.get('stage', 'unknown')
                }

        except Exception as e:
            logging.error(f"Sandwich execution error: {e}")
            return {'success': False, 'error': str(e)}

    async def validate_sandwich_opportunity(self, opportunity:
SandwichOpportunity) -> Dict:
        """Validate if sandwich opportunity is still viable"""

        # Check if victim transaction is still pending
        tx_status = await
self.check_transaction_status(opportunity.victim_tx.hash)

        if tx_status['confirmed']:
            return {'valid': False, 'error': 'Victim transaction
already confirmed'}

        # Check current gas prices
        current_gas_price = self.w3.eth.gas_price
        max_affordable_gas = opportunity.victim_tx.gas_price * 1.1  #
10% higher max
```

```
        if current_gas_price > max_affordable_gas:
            return {'valid': False, 'error':
'Gas prices too high to be competitive'}

        # Check if we have sufficient funds
        balance = self.w3.eth.get_balance(self.account.address)
        required_balance = await
self.calculate_required_balance(opportunity)

        if balance < required_balance:
            return {'valid': False, 'error': 'Insufficient funds for
execution'}

        # Check profitability after costs
        updated_profit = await
self.calculate_current_profitability(opportunity)
        if updated_profit < self.profit_threshold:
            return {'valid': False, 'error': 'Profitability below
threshold'}

        return {
            'valid': True,
            'current_gas_price': current_gas_price,
            'available_balance': balance,
            'updated_profit': updated_profit
        }

    async def execute_sandwich_sequence(self, opportunity:
SandwichOpportunity) -> Dict:
        """Execute the complete sandwich attack sequence"""

        start_time = time.time()
        stages_completed = []

        try:
            # Stage 1: Front-run transaction
            logging.info("Executing front-run transaction")
            front_run_result = await
self.execute_front_run(opportunity)
            if not front_run_result['success']:
                return {'success': False, 'stage': 'front_run',
```

```
'error': front_run_result['error']}

            stages_completed.append('front_run')

            # Wait for victim transaction to be included
            victim_confirmed = await
self.wait_for_victim_confirmation(opportunity.victim_tx.hash)
            if not victim_confirmed:
                return {'success': False, 'stage': 'victim_wait',
'error': 'Victim transaction failed'}

            stages_completed.append('victim_confirmed')

            # Stage 2: Back-run transaction
            logging.info("Executing back-run transaction")
            back_run_result = await self.execute_back_run(opportunity,
front_run_result)
            if not back_run_result['success']:
                return {'success': False, 'stage': 'back_run', 'error':
back_run_result['error']}

            stages_completed.append('back_run')

            # Calculate final results
            execution_time = time.time() - start_time

            return {
                'success': True,
                'stages_completed': stages_completed,
                'front_run_tx': front_run_result['tx_hash'],
                'back_run_tx': back_run_result['tx_hash'],
                'execution_time': execution_time,
                'gas_used': front_run_result['gas_used'] +
back_run_result['gas_used']
            }

        except Exception as e:
            # Attempt cleanup if needed
            if 'front_run' in stages_completed:
                await self.attempt_cleanup(stages_completed)

            return {
```

```
                'success': False,
                'stage': stages_completed[-1] if stages_completed else
'initialization',
                'error': str(e),
                'stages_completed': stages_completed
            }

    async def execute_front_run(self, opportunity: SandwichOpportunity)
-> Dict:
        """Execute the front-run transaction"""

        try:
            # Parse victim transaction to understand the swap
            swap_details =
self.parse_victim_swap(opportunity.victim_tx)

            if not swap_details:
                return {'success': False, 'error': 'Failed to parse
victim transaction'}

            # Calculate optimal front-run parameters
            front_run_params =
self.calculate_front_run_params(opportunity, swap_details)

            # Build front-run transaction
            transaction = await self.build_swap_transaction(
                token_in=front_run_params['token_in'],
                token_out=front_run_params['token_out'],
                amount_in=front_run_params['amount_in'],
                amount_out_min=front_run_params['amount_out_min'],
                path=front_run_params['path']
            )

            # Set gas parameters for competitive execution
            gas_price = opportunity.victim_tx.gas_price * 1.05  # 5%
higher than victim
            transaction['gasPrice'] = int(gas_price)
            transaction['gas'] = int(front_run_params['estimated_gas']
* self.gas_buffer)

            # Sign and send transaction
            signed_tx = self.account.sign_transaction(transaction)
```

```
            tx_hash =
self.w3.eth.send_raw_transaction(signed_tx.rawTransaction)

            # Wait for confirmation
            receipt = self.w3.eth.wait_for_transaction_receipt(tx_hash)

            if receipt.status == 1:
                return {
                    'success': True,
                    'tx_hash': tx_hash.hex(),
                    'gas_used': receipt.gasUsed,
                    'transaction_details': front_run_params
                }
            else:
                return {'success': False, 'error': 'Front-run
transaction failed'}

        except Exception as e:
            return {'success': False, 'error': f'Front-run execution
error: {str(e)}'}

    def parse_victim_swap(self, tx: Transaction) -> Optional[Dict]:
        """Parse victim transaction to understand the swap"""

        if tx.to_address ==
'0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D':  # Uniswap V2
            return self.parse_uniswap_v2_swap(tx.data)
        elif tx.to_address ==
'0xd9e1cE17f2641f24aE83637ab66a2cca9C378B9F':  # SushiSwap
            return self.parse_uniswap_v2_swap(tx.data)  # Same ABI

        return None

    def parse_uniswap_v2_swap(self, data: str) -> Optional[Dict]:
        """Parse Uniswap V2 style swap transaction"""

        if len(data) < 10:
            return None

        function_selector = data[:10]

        if function_selector == '0x38ed1739':  #
```

```
swapExactTokensForTokens
            return self.parse_tokens_for_tokens(data)
        elif function_selector == '0x7ff36ab5':  #
swapExactETHForTokens
            return self.parse_eth_for_tokens(data)
        elif function_selector == '0x18cbafe5':  #
swapExactTokensForETH
            return self.parse_tokens_for_eth(data)

        return None

    def parse_tokens_for_tokens(self, data: str) -> Dict:
        """Parse swapExactTokensForTokens transaction"""

        try:
            amount_in = int(data[10:74], 16)
            amount_out_min = int(data[74:138], 16)

            # Extract path (simplified - real implementation would use
proper ABI decoding)
            path_start = 138
            path_length = (len(data) - path_start - 8) // 64  # Remove
deadline

            path = []
            for i in range(path_length):
                token_start = path_start + (i * 64)
                token_end = token_start + 64
                if token_end <= len(data) - 8:
                    token = data[token_start + 24:token_end]  # Remove
padding
                    if token != '0' * 40:
                        path.append('0x' + token)

            return {
                'type': 'tokens_for_tokens',
                'amount_in': amount_in,
                'amount_out_min': amount_out_min,
                'path': path,
                'token_in': path[0] if len(path) > 0 else None,
                'token_out': path[-1] if len(path) > 1 else None
            }
```

```
        except Exception as e:
            logging.error(f"Error parsing tokens swap: {e}")
            return None

    def calculate_front_run_params(self, opportunity:
SandwichOpportunity, swap_details: Dict) -> Dict:
        """Calculate optimal parameters for front-run transaction"""

        # Optimal size is typically 50-80% of victim trade
        victim_size = swap_details['amount_in']
        optimal_ratio = 0.6  # Conservative 60%
        front_run_size = int(victim_size * optimal_ratio)

        # Calculate expected output with slippage
        # For simplicity, assume linear relationship
        expected_output_ratio = 0.999  # 0.1% slippage for front-run
        expected_output = int(front_run_size * expected_output_ratio)

        # Set minimum output (should be profitable even if some
slippage)
        min_profitable_output = int(front_run_size * 0.995)

        return {
            'token_in': swap_details['token_in'],
            'token_out': swap_details['token_out'],
            'amount_in': front_run_size,
            'amount_out_min': min_profitable_output,
            'path': swap_details['path'],
            'estimated_gas': 200000,  # Estimated gas for swap
            'expected_output': expected_output
        }

    async def build_swap_transaction(self, token_in: str, token_out:
str, amount_in: int,
                                     amount_out_min: int, path:
List[str]) -> Dict:
        """Build swap transaction for DEX"""

        # Determine which router to use
        router = self.routers['uniswap_v2']  # Default to Uniswap V2
```

```
        # Build transaction based on swap type
        if token_in.lower() ==
'0xeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee':  # ETH
            # ETH -> Token swap
            function_call = router.functions.swapExactETHForTokens(
                amount_out_min,
                path,
                self.account.address,
                int(time.time()) + 300  # 5 minute deadline
            )
            tx_data = function_call.build_transaction({
                'value': amount_in
            })
        elif token_out.lower() ==
'0xeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee':  # ETH
            # Token -> ETH swap
            # First, approve token transfer
            token_contract = self.w3.eth.contract(address=token_in,
abi=self.get_erc20_abi())

            approve_tx = token_contract.functions.approve(
                router.address,
                amount_in
            ).build_transaction({
                'from': self.account.address,
                'gasPrice': self.w3.eth.gas_price,
                'nonce':
self.w3.eth.get_transaction_count(self.account.address),
            })

            # Build swap transaction
            function_call = router.functions.swapExactTokensForETH(
                amount_in,
                amount_out_min,
                path,
                self.account.address,
                int(time.time()) + 300
            )

            swap_tx = function_call.build_transaction({
                'from': self.account.address,
                'gasPrice': self.w3.eth.gas_price,
```

```
                'nonce':
self.w3.eth.get_transaction_count(self.account.address) + 1,
            })

            return swap_tx  # Return the swap transaction
        else:
            # Token -> Token swap
            function_call = router.functions.swapExactTokensForTokens(
                amount_in,
                amount_out_min,
                path,
                self.account.address,
                int(time.time()) + 300
            )

            tx_data = function_call.build_transaction({
                'from': self.account.address,
                'gasPrice': self.w3.eth.gas_price,
                'nonce':
self.w3.eth.get_transaction_count(self.account.address),
            })

        return tx_data

    async def execute_back_run(self, opportunity: SandwichOpportunity,
front_run_result: Dict) -> Dict:
        """Execute the back-run transaction"""

        try:
            # Get the output from front-run transaction
            front_run_output = await
self.get_transaction_output(front_run_result['tx_hash'])

            if not front_run_output:
                return {'success': False, 'error': 'Failed to get
front-run output'}

            # Calculate back-run parameters
            back_run_params =
self.calculate_back_run_params(opportunity, front_run_output)

            # Build back-run transaction
```

```
            transaction = await self.build_swap_transaction(
                token_in=back_run_params['token_in'],
                token_out=back_run_params['token_out'],
                amount_in=back_run_params['amount_in'],
                amount_out_min=back_run_params['amount_out_min'],
                path=back_run_params['path']
            )

            # Set gas parameters
            transaction['gasPrice'] = int(self.w3.eth.gas_price * 1.1)
# 10% above current
            transaction['gas'] = int(back_run_params['estimated_gas'] *
self.gas_buffer)

            # Sign and send transaction
            signed_tx = self.account.sign_transaction(transaction)
            tx_hash =
self.w3.eth.send_raw_transaction(signed_tx.rawTransaction)

            # Wait for confirmation
            receipt = self.w3.eth.wait_for_transaction_receipt(tx_hash)

            if receipt.status == 1:
                return {
                    'success': True,
                    'tx_hash': tx_hash.hex(),
                    'gas_used': receipt.gasUsed,
                    'transaction_details': back_run_params
                }
            else:
                return {'success': False, 'error': 'Back-run
transaction failed'}

        except Exception as e:
            return {'success': False, 'error': f'Back-run execution
error: {str(e)}'}

    def calculate_back_run_params(self, opportunity:
SandwichOpportunity, front_run_output: int) -> Dict:
        """Calculate parameters for back-run transaction"""

        # For back-run, we typically reverse the front-run trade
```

```
        swap_details = opportunity.victim_tx  # Original victim
transaction

        # Reverse the token flow
        if swap_details.get('type') == 'tokens_for_tokens':
            # If victim swapped Token A -> Token B
            # Front-run was Token A -> Token B
            # Back-run should be Token B -> Token A

            token_in = swap_details['token_out']  # Token B
            token_out = swap_details['token_in']  # Token A
            path = list(reversed(swap_details['path']))
            amount_in = front_run_output  # Use the output from front-
run

            # Calculate minimum output for profitability
            min_output = int(amount_in * 0.995)  # Allow 0.5% slippage

        else:
            # Handle other swap types
            # This would need to be implemented based on specific swap
types
            return None

        return {
            'token_in': token_in,
            'token_out': token_out,
            'amount_in': amount_in,
            'amount_out_min': min_output,
            'path': path,
            'estimated_gas': 200000
        }
```

Risk Management and Ethics

Risk Assessment Framework

```
class SandwichRiskManager:
    def __init__(self, config: Dict):
        self.config = config
        self.max_daily_attacks = config.get('max_daily_attacks', 10)
        self.max_position_size = config.get('max_position_size',
100000)  # $100k
        self.min_profit_threshold = config.get('min_profit_threshold',
50)

        # Track daily activity
        self.daily_attack_count = 0
        self.daily_profit = 0.0
        self.last_reset_date = datetime.now().date()

    async def assess_sandwich_risk(self, opportunity:
SandwichOpportunity) -> Dict:
        """Comprehensive risk assessment for sandwich attack"""

        risks = {}

        # Market risk
        risks['market_risk'] = await
self.assess_market_risk(opportunity)

        # Execution risk
        risks['execution_risk'] = await
self.assess_execution_risk(opportunity)

        # Competition risk
        risks['competition_risk'] = await
self.assess_competition_risk(opportunity)

        # Regulatory risk
        risks['regulatory_risk'] =
self.assess_regulatory_risk(opportunity)

        # Ethical risk
        risks['ethical_risk'] = self.assess_ethical_risk(opportunity)

        # Overall risk score
        risks['overall_risk_score'] =
self.calculate_overall_risk_score(risks)
```

```
        return risks

    async def assess_market_risk(self, opportunity:
SandwichOpportunity) -> Dict:
        """Assess market-related risks"""

        market_risks = {}

        # Volatility risk
        market_volatility = await
self.get_market_volatility(opportunity.victim_tx)
        market_risks['volatility'] = {
            'level': 'high' if market_volatility > 0.05 else 'medium'
if market_volatility > 0.02 else 'low',
            'value': market_volatility,
            'impact': 'price movement unpredictability'
        }

        # Liquidity risk
        liquidity_risk = await self.assess_liquidity_risk(opportunity)
        market_risks['liquidity'] = liquidity_risk

        # Correlation risk
        correlation_risk = await
self.assess_correlation_risk(opportunity)
        market_risks['correlation'] = correlation_risk

        return market_risks

    async def assess_execution_risk(self, opportunity:
SandwichOpportunity) -> Dict:
        """Assess execution-related risks"""

        execution_risks = {}

        # Gas price risk
        current_gas_price = self.w3.eth.gas_price if hasattr(self,
'w3') else 0
        victim_gas_price = opportunity.victim_tx.gas_price

        gas_price_risk = 'low' if current_gas_price < victim_gas_price
```

```
* 1.2 else 'high'
        execution_risks['gas_price'] = {
            'level': gas_price_risk,
            'current_price': current_gas_price,
            'victim_price': victim_gas_price,
            'competition_multiplier': current_gas_price /
victim_gas_price if victim_gas_price > 0 else 1
        }

        # Network congestion risk
        network_congestion = await self.assess_network_congestion()
        execution_risks['network_congestion'] = {
            'level': 'high' if network_congestion > 0.8 else 'medium'
if network_congestion > 0.5 else 'low',
            'congestion_level': network_congestion,
            'estimated_delay': network_congestion * 30  # Seconds
        }

        # Technical risk
        technical_risk = self.assess_technical_risk(opportunity)
        execution_risks['technical'] = technical_risk

        return execution_risks

    def assess_ethical_risk(self, opportunity: SandwichOpportunity) ->
Dict:
        """Assess ethical considerations"""

        ethical_assessment = {}

        # Victim impact assessment
        victim_impact = self.assess_victim_impact(opportunity)
        ethical_assessment['victim_impact'] = victim_impact

        # Market impact assessment
        market_impact = self.assess_market_impact(opportunity)
        ethical_assessment['market_impact'] = market_impact

        # Intent assessment
        intent_assessment =
self.assess_transaction_intent(opportunity.victim_tx)
        ethical_assessment['intent'] = intent_assessment
```

```
        # Overall ethical score
        ethical_score =
self.calculate_ethical_score(ethical_assessment)
        ethical_assessment['overall_score'] = ethical_score
        ethical_assessment['recommendation'] = 'proceed' if
ethical_score > 0.7 else 'caution' if ethical_score > 0.4 else 'avoid'

        return ethical_assessment

    def assess_victim_impact(self, opportunity: SandwichOpportunity) ->
Dict:
        """Assess potential impact on victim"""

        # Estimate victim slippage increase
        estimated_victim_slippage = opportunity.predicted_impact * 1.5
# Sandwitch amplifies impact

        # Calculate victim cost
        victim_transaction_value = opportunity.victim_tx.value * 1e-18
* 2000  # Assuming ETH
        victim_additional_cost = victim_transaction_value *
estimated_victim_slippage

        return {
            'estimated_slippage_increase': estimated_victim_slippage,
            'estimated_additional_cost': victim_additional_cost,
            'victim_transaction_value': victim_transaction_value,
            'impact_severity': 'high' if victim_additional_cost > 1000
else 'medium' if victim_additional_cost > 100 else 'low'
        }

    def assess_transaction_intent(self, tx: Transaction) -> Dict:
        """Assess the intent and characteristics of the victim
transaction"""

        intent_assessment = {}

        # Check if transaction appears to be a bot or human
        intent_assessment['likely_bot'] =
self.detect_bot_transaction(tx)
```

```
        # Check transaction characteristics
        intent_assessment['characteristics'] = {
            'gas_price': self.classify_gas_price(tx.gas_price),
            'timing': self.analyze_timing_pattern(tx),
            'complexity': self.analyze_transaction_complexity(tx)
        }

        # Determine if victim deserves protection
        deserves_protection =
self.should_protect_victim(intent_assessment)
        intent_assessment['deserves_protection'] = deserves_protection

        return intent_assessment

    def detect_bot_transaction(self, tx: Transaction) -> bool:
        """Detect if transaction is likely from a bot"""

        # Bot indicators
        bot_indicators = 0

        # High gas price with round numbers
        if tx.gas_price % 1e9 == 0 and tx.gas_price > 50e9:
            bot_indicators += 1

        # Quick succession of similar transactions
        # This would require transaction history analysis

        # Complex transaction data indicating automated strategies
        if len(tx.data) > 200 and '0x' in tx.data:
            bot_indicators += 1

        # High-value transactions from contract addresses
        if tx.value > 100 * 1e18 and tx.from_address.startswith('0x'):
# Contract addresses
            bot_indicators += 1

        return bot_indicators >= 2

    def should_protect_victim(self, intent_assessment: Dict) -> bool:
        """Determine if victim transaction should be protected from
sandwiching"""
```

```
        # Don't protect bot transactions
        if intent_assessment.get('likely_bot', False):
            return False

        # Protect small retail transactions
        # This would require more sophisticated analysis

        # Protect transactions that appear to be from humans (not bots)
        characteristics = intent_assessment.get('characteristics', {})

        human_indicators = 0

        if characteristics.get('gas_price') == 'normal':
            human_indicators += 1

        if characteristics.get('timing') == 'human_like':
            human_indicators += 1

        return human_indicators >= 2

    def calculate_ethical_score(self, ethical_assessment: Dict) ->
float:
        """Calculate overall ethical score for sandwich attack"""

        base_score = 0.8  # Start with neutral score

        # Penalize high victim impact
        victim_impact = ethical_assessment.get('victim_impact', {})
        impact_severity = victim_impact.get('impact_severity', 'low')

        if impact_severity == 'high':
            base_score -= 0.4
        elif impact_severity == 'medium':
            base_score -= 0.2

        # Penalize attacks on protected victims
        if ethical_assessment.get('intent',
{}).get('deserves_protection', False):
            base_score -= 0.3

        # Reward attacks on bot transactions
        if ethical_assessment.get('intent', {}).get('likely_bot',
```

```
False):
            base_score += 0.1

        return max(0, min(1, base_score))

    def implement_ethical_filters(self, opportunities:
List[SandwichOpportunity]) -> List[SandwichOpportunity]:
        """Apply ethical filters to sandwich opportunities"""

        filtered_opportunities = []

        for opportunity in opportunities:
            ethical_assessment = self.assess_ethical_risk(opportunity)

            # Filter based on ethical score
            ethical_score = ethical_assessment.get('overall_score', 0)
            recommendation = ethical_assessment.get('recommendation',
'avoid')

            if recommendation == 'proceed':
                filtered_opportunities.append(opportunity)
            elif recommendation == 'caution' and ethical_score > 0.4:
                # Include cautiously with additional checks
                opportunity.ethical_notes = f"Caution: Score
{ethical_score:.2f}"
                filtered_opportunities.append(opportunity)
            # Skip opportunities marked as 'avoid'

        return filtered_opportunities
```

Competition Handling

```
class CompetitionManager:
    def __init__(self, config: Dict):
        self.config = config
        self.competitor_analysis = CompetitorAnalyzer()
        self.strategy_adaptor = StrategyAdaptor()

    async def handle_competition(self, opportunity:
SandwichOpportunity) -> Dict:
        """Handle competitive MEV environment"""

        # Analyze current competition
        competition_analysis = await
self.competitor_analysis.analyze_current_competition()

        # Adapt strategy based on competition
        adapted_strategy = await self.strategy_adaptor.adapt_strategy(
            opportunity, competition_analysis
        )

        # Execute with competition-aware parameters
        execution_result = await
self.execute_with_competition_awareness(
            opportunity, adapted_strategy
        )

        return {
            'competition_analysis': competition_analysis,
            'adapted_strategy': adapted_strategy,
            'execution_result': execution_result,
            'competitive_advantage':
execution_result.get('advantage_score', 0)
        }

    async def execute_with_competition_awareness(self, opportunity:
SandwichOpportunity,
                                                 strategy: Dict) -> Dict:
        """Execute sandwich attack with awareness of competition"""

        # Adjust gas pricing for competition
        adjusted_gas_params = self.adjust_gas_for_competition(
            opportunity, strategy.get('competition_level', 'medium')
        )
```

```
        # Choose optimal timing
        timing_strategy = self.determine_optimal_timing(opportunity,
competition_analysis)

        # Execute with competition parameters
        start_time = time.time()

        try:
            # Monitor for competing transactions
            competition_monitor = CompetitionMonitor()
            await
competition_monitor.start_monitoring(opportunity.victim_tx.hash)

            # Execute front-run with competition awareness
            front_run_result = await
self.execute_competitive_front_run(
                opportunity, adjusted_gas_params, timing_strategy
            )

            if front_run_result['success']:
                # Continue with back-run
                back_run_result = await
self.execute_competitive_back_run(
                    opportunity, front_run_result, competition_monitor
                )

                total_execution_time = time.time() - start_time

                return {
                    'success': True,
                    'execution_time': total_execution_time,
                    'competition_detected':
competition_monitor.was_competition_detected(),
                    'competitive_advantage':
self.calculate_competitive_advantage(
                        front_run_result, back_run_result,
competition_monitor
                    ),
                    'strategy_effectiveness':
self.assess_strategy_effectiveness(
                        adapted_strategy, front_run_result,
```

```
back_run_result
                    )
                }
            else:
                return {
                    'success': False,
                    'error': 'Failed to execute front-run under
competition',
                    'competition_level':
strategy.get('competition_level', 'unknown')
                }

        except Exception as e:
            return {
                'success': False,
                'error': str(e),
                'execution_time': time.time() - start_time
            }

class CompetitorAnalyzer:
    """Analyze competitor behavior and strategies"""

    def __init__(self):
        self.competitor_patterns = {}
        self.strategy_effectiveness = {}

    async def analyze_current_competition(self) -> Dict:
        """Analyze current MEV competition environment"""

        # Monitor recent MEV transactions
        recent_mev_activity = await self.get_recent_mev_activity()

        # Analyze competitor patterns
        competitor_analysis =
self.analyze_competitor_patterns(recent_mev_activity)

        # Assess competition intensity
        competition_intensity =
self.calculate_competition_intensity(recent_mev_activity)

        # Identify dominant strategies
        dominant_strategies =
```

```
self.identify_dominant_strategies(recent_mev_activity)

        return {
            'competition_intensity': competition_intensity,
            'dominant_strategies': dominant_strategies,
            'competitor_patterns': competitor_analysis,
            'market_state':
self.assess_market_state(recent_mev_activity),
            'recommendations':
self.generate_competition_recommendations(
                competition_intensity, dominant_strategies
            )
        }

    async def get_recent_mev_activity(self, hours: int = 1) ->
List[Dict]:
        """Get recent MEV activity from various sources"""

        # This would typically query transaction data
        # For now, return simulated data structure

        return [
            {
                'timestamp': datetime.now() - timedelta(minutes=5 * i),
                'type': 'sandwich',
                'profit': random.uniform(50, 500),
                'gas_price': random.uniform(20e9, 100e9),
                'success': random.choice([True, True, True, False]),
# 75% success rate
                'victim_type': random.choice(['bot', 'human',
'unknown'])
            }
            for i in range(12)  # 12 transactions in the last hour
        ]

    def analyze_competitor_patterns(self, mev_activity: List[Dict]) ->
Dict:
        """Analyze patterns in competitor behavior"""

        patterns = {}

        # Gas price patterns
```

```
        gas_prices = [activity['gas_price'] for activity in
mev_activity if activity['success']]
        if gas_prices:
            patterns['gas_price'] = {
                'average': sum(gas_prices) / len(gas_prices),
                'median': sorted(gas_prices)[len(gas_prices) // 2],
                'aggressive_threshold': max(gas_prices) * 0.9  # Top
10% threshold
            }

        # Timing patterns
        timestamps = [activity['timestamp'] for activity in
mev_activity]
        if len(timestamps) > 1:
            intervals = [(timestamps[i] -
timestamps[i-1]).total_seconds()
                         for i in range(1, len(timestamps))]
            patterns['timing'] = {
                'average_interval': sum(intervals) / len(intervals),
                'typical_range': [min(intervals), max(intervals)]
            }

        # Success rate analysis
        success_rate = sum(1 for activity in mev_activity if
activity['success']) / len(mev_activity)
        patterns['success_rate'] = success_rate

        return patterns

    def calculate_competition_intensity(self, mev_activity: List[Dict])
-> str:
        """Calculate competition intensity level"""

        recent_activity = [
            activity for activity in mev_activity
            if activity['timestamp'] > datetime.now() -
timedelta(minutes=30)
        ]

        if len(recent_activity) > 10:  # High frequency
            return 'HIGH'
        elif len(recent_activity) > 5:  # Medium frequency
```

```
            return 'MEDIUM'
        else:
            return 'LOW'

    def identify_dominant_strategies(self, mev_activity: List[Dict]) ->
List[str]:
        """Identify dominant MEV strategies currently being used"""

        strategy_counts = {}

        for activity in mev_activity:
            strategy_type = activity.get('type', 'unknown')
            strategy_counts[strategy_type] =
strategy_counts.get(strategy_type, 0) + 1

        # Return strategies with >20% market share
        total = sum(strategy_counts.values())
        dominant = [
            strategy for strategy, count in strategy_counts.items()
            if count / total > 0.2
        ]

        return dominant

class StrategyAdaptor:
    """Adapt sandwich strategies based on market conditions"""

    def __init__(self):
        self.adaptation_strategies = {
            'HIGH': self.adapt_to_high_competition,
            'MEDIUM': self.adapt_to_medium_competition,
            'LOW': self.adapt_to_low_competition
        }

    async def adapt_strategy(self, opportunity: SandwichOpportunity,
                             competition_analysis: Dict) -> Dict:
        """Adapt sandwich strategy based on competition analysis"""

        competition_level =
competition_analysis.get('competition_intensity', 'MEDIUM')
        adaptation_func =
self.adaptation_strategies.get(competition_level,
```

```
self.adapt_to_medium_competition)

        return await adaptation_func(opportunity, competition_analysis)

    async def adapt_to_high_competition(self, opportunity:
SandwichOpportunity,
                                        competition_analysis: Dict) ->
Dict:
        """Adapt strategy for high competition environment"""

        return {
            'competition_level': 'HIGH',
            'gas_strategy': 'aggressive',  # Pay premium for faster
inclusion
            'size_strategy': 'conservative',  # Smaller trades to
reduce detection
            'timing_strategy': 'immediate',
# Execute immediately upon detection
            'profit_threshold': opportunity.estimated_profit * 1.5,  #
Higher threshold for competition
            'expected_success_rate': 0.6,
            'risk_level': 'high'
        }

    async def adapt_to_medium_competition(self, opportunity:
SandwichOpportunity,
                                          competition_analysis: Dict) ->
Dict:
        """Adapt strategy for medium competition environment"""

        return {
            'competition_level': 'MEDIUM',
            'gas_strategy': 'balanced',  # Balanced gas pricing
            'size_strategy': 'normal',  # Normal trade sizes
            'timing_strategy': 'wait_slight',  # Small delay to let
victim transaction progress
            'profit_threshold': opportunity.estimated_profit * 1.2,
            'expected_success_rate': 0.75,
            'risk_level': 'medium'
        }

    async def adapt_to_low_competition(self, opportunity:
```

```
SandwichOpportunity,
                                    competition_analysis: Dict) ->
Dict:
        """Adapt strategy for low competition environment"""

        return {
            'competition_level': 'LOW',
            'gas_strategy': 'conservative',  # Lower gas prices
acceptable
            'size_strategy': 'aggressive',  # Can use larger trade
sizes
            'timing_strategy': 'patient',  # Can wait for better
conditions
            'profit_threshold': opportunity.estimated_profit * 1.1,
            'expected_success_rate': 0.85,
            'risk_level': 'low'
        }
```

Quick Check Quiz

1. **What are the three main components of a sandwich attack?**
   a) Front-run, main trade, back-run
   b) Detection, execution, analysis
   c) Buy, hold, sell
   d) Monitor, predict, act
2. **What is the primary profit source in a sandwich attack?**
   a) Transaction fees
   b) Price impact from victim trade
   c) Gas rebates
   d) Token price appreciation
3. **When should you avoid sandwich attacks from an ethical perspective?**
   a) When profits are too high
   b) When targeting retail users with small trades
   c) When gas prices are high
   d) When the market is volatile
4. **What is the typical success rate for sandwich attacks in competitive environments?**
   a) 90-100%
   b) 70-80%
   c) 50-60%
   d) 30-40%

5. **How can you detect if a transaction is from a bot vs. a human?**
   a) Transaction value size
   b) Gas price patterns and transaction complexity
   c) Time of day
   d) Token types involved

Summary

Sandwich attacks represent one of the most sophisticated MEV strategies. Key takeaways:

- **Detection Systems**: Real-time mempool monitoring is crucial for opportunity identification
- **Execution Complexity**: Requires precise timing and competitive gas pricing
- **Risk Management**: Both technical and ethical considerations must be evaluated
- **Competition Handling**: Adaptation strategies are essential in competitive environments
- **Profitability**: Success depends on accurate prediction and efficient execution

In the final module, we'll explore strategy testing and validation techniques to ensure your MEV strategies perform reliably.

Next Module Preview

Module 6: Strategy Testing & Validation
- Setting up comprehensive testing frameworks
- Backtesting historical MEV opportunities
- Paper trading and simulation environments
- Performance metrics and analysis
- Deployment strategies and monitoring

Author: Obelisk Core
Module Duration: 95 minutes
Prerequisites: Completion of Modules 1-4, understanding of transaction mechanics and risk management
Resources: Testing frameworks, simulation tools, and deployment guides