Patent Applications for AI Agent Security Platform Innovations

Patent Application 1: Graph-Based Agent Trust Networks

TITLE

System and Method for Graph-Based Trust Network Analysis in Multi-Agent Artificial Intelligence Systems

TECHNICAL FIELD

This invention relates to computer security systems, and more particularly to methods and systems for establishing, maintaining, and analyzing trust relationships in networks of autonomous artificial intelligence agents using graph-based computational models.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention addresses the critical security challenge of managing trust relationships in multi-agent artificial intelligence systems. As AI agents become increasingly autonomous and interact in complex networks, traditional security models prove inadequate for managing inter-agent trust and preventing coordinated malicious activities.

Description of Related Art

Current agent security systems primarily focus on individual agent monitoring without considering the complex trust relationships that develop between agents in multi-agent environments. Existing solutions include:

1. **Individual Agent Monitoring**: Traditional systems monitor single agents in isolation, failing to account for trust propagation and collective behaviors.
2. **Static Trust Scores**: Current trust systems assign fixed scores without considering dynamic relationship changes and network effects.
3. **Rule-Based Security**: Existing systems rely on predefined rules that cannot adapt to emergent trust patterns in agent networks.

Problems with Prior Art:

- Inability to detect coordinated attacks across multiple agents
- Lack of dynamic trust adjustment based on network behavior
- No consideration of trust transitivity and propagation effects

- Limited scalability for large agent networks
- Absence of graph-based analysis for trust relationship visualization

SUMMARY OF THE INVENTION

The present invention provides a novel system and method for establishing, analyzing, and maintaining trust networks in multi-agent AI systems using graph-based computational models. The system dynamically calculates trust scores based on network topology, interaction history, and behavioral patterns while detecting anomalous trust relationships that may indicate security threats.

Key Technical Contributions:

1. **Dynamic Trust Propagation Algorithm**: A novel algorithm that calculates trust scores considering both direct interactions and indirect trust propagation through network paths.
2. **Graph-Based Anomaly Detection**: Advanced detection methods that identify suspicious trust patterns, coordinated malicious behavior, and trust network manipulation.
3. **Real-Time Trust Network Analysis**: Efficient algorithms for analyzing large-scale agent networks in real-time with minimal computational overhead.
4. **Adaptive Trust Thresholds**: Dynamic threshold adjustment based on network behavior patterns and threat landscape evolution.

DETAILED DESCRIPTION OF THE INVENTION

System Architecture

The graph-based trust network system comprises:

1. **Trust Graph Database**: A specialized graph database storing agent nodes and trust relationships as weighted edges.
2. **Trust Propagation Engine**: Computational engine implementing novel trust propagation algorithms.
3. **Anomaly Detection Module**: Real-time analysis system for identifying suspicious trust patterns.
4. **Trust Score Calculator**: Dynamic scoring system considering multiple trust factors.

Core Algorithms

Algorithm 1: Dynamic Trust Propagation

python

```python
def calculate_propagated_trust(source_agent, target_agent, max_hops=3, decay_factor=0.8):
    """
    Calculate trust score considering indirect trust paths

    Args:
        source_agent: Starting agent in trust calculation
        target_agent: Target agent for trust evaluation
        max_hops: Maximum path length for trust propagation
        decay_factor: Trust decay rate per hop

    Returns:
        Float: Propagated trust score (0.0 to 1.0)
    """
    direct_trust = get_direct_trust(source_agent, target_agent)

    if direct_trust is not None:
        return direct_trust

    trust_paths = find_trust_paths(source_agent, target_agent, max_hops)
    propagated_scores = []

    for path in trust_paths:
        path_trust = 1.0
        for hop_index, (agent_a, agent_b) in enumerate(path):
            hop_trust = get_direct_trust(agent_a, agent_b)
            path_trust *= hop_trust * (decay_factor ** hop_index)
        propagated_scores.append(path_trust)

    # Combine multiple paths using weighted average
    return calculate_weighted_average(propagated_scores)
```

Algorithm 2: Trust Network Anomaly Detection

```
python
```

```python
def detect_trust_anomalies(agent_network, time_window=24):
    """
    Detect anomalous patterns in agent trust networks

    Args:
        agent_network: Graph representation of agent trust relationships
        time_window: Analysis time window in hours

    Returns:
        List[TrustAnomaly]: Detected anomalies with confidence scores
    """
    anomalies = []

    # Detect sudden trust changes
    trust_changes = analyze_trust_velocity(agent_network, time_window)
    for change in trust_changes:
        if change.velocity > VELOCITY_THRESHOLD:
            anomalies.append(TrustAnomaly(
                type="rapid_trust_change",
                agents=change.involved_agents,
                confidence=calculate_confidence(change)
            ))

    # Detect coordinated trust manipulation
    coordination_patterns = detect_coordination_patterns(agent_network)
    for pattern in coordination_patterns:
        if pattern.coordination_score > COORDINATION_THRESHOLD:
            anomalies.append(TrustAnomaly(
                type="coordinated_manipulation",
                agents=pattern.participating_agents,
                confidence=pattern.coordination_score
            ))

    # Detect trust isolation attacks
    isolation_attempts = detect_isolation_patterns(agent_network)
    for isolation in isolation_attempts:
        anomalies.append(TrustAnomaly(
            type="trust_isolation",
            target_agent=isolation.target,
            attacking_agents=isolation.attackers,
            confidence=isolation.isolation_score
        ))
    return sorted(anomalies, key=lambda x: x.confidence, reverse=True)
```

Novel Technical Features

1. Trust Transitivity Calculation The system implements a novel approach to calculating trust transitivity that considers:

- Path length and trust decay
- Multiple path aggregation
- Temporal trust evolution
- Bidirectional trust relationships

2. Graph-Based Clustering for Trust Communities Advanced clustering algorithms identify trust communities within agent networks:

- Community detection using modularity optimization
- Trust-weighted edge consideration
- Dynamic community evolution tracking
- Cross-community trust bridge identification

3. Real-Time Trust Graph Updates Efficient algorithms for maintaining trust graphs in real-time:

- Incremental trust score updates
- Lazy propagation for computational efficiency
- Event-driven trust recalculation
- Distributed trust computation for scalability

Implementation Details

Graph Database Schema:

```
cypher
```

```
// Agent nodes with trust-related properties
CREATE (agent:Agent {
  id: "uuid",
  trust_score: 0.0-1.0,
  reputation: 0.0-1.0,
  interaction_count: integer,
  last_active: timestamp,
  behavioral_fingerprint: "hash"
})

// Trust relationships with weighted edges
CREATE (agent1:Agent)-[:TRUSTS {
  score: 0.0-1.0,
  confidence: 0.0-1.0,
  interaction_count: integer,
  last_interaction: timestamp,
  trust_history: [scores],
  relationship_type: "direct|inferred|delegated"
}]->(agent2:Agent)
```

Performance Optimizations:

- Graph traversal optimization using bidirectional search
- Trust score caching with TTL-based invalidation
- Parallel processing for large network analysis
- Memory-efficient graph representation

CLAIMS

Claim 1: A computer-implemented method for analyzing trust relationships in multi-agent artificial intelligence systems, comprising: a) maintaining a graph database representing agents as nodes and trust relationships as weighted edges; b) implementing a dynamic trust propagation algorithm that calculates trust scores considering both direct and indirect trust paths; c) detecting anomalous trust patterns using graph-based analysis techniques; d) updating trust scores in real-time based on agent interactions and behavioral changes.

Claim 2: The method of claim 1, wherein the dynamic trust propagation algorithm includes: a) identifying trust paths between agents with configurable maximum hop limits; b) applying decay factors to trust scores based on path length; c) aggregating multiple trust paths using weighted averaging; d) considering temporal factors in trust calculation.

Claim 3: The method of claim 1, wherein anomaly detection comprises: a) detecting rapid trust score changes exceeding velocity thresholds; b) identifying coordinated trust manipulation patterns; c) recognizing trust isolation attack attempts; d) calculating confidence scores for detected anomalies.

Claim 4: A system for graph-based trust network analysis comprising: a) a graph database storing agent trust relationships; b) a trust propagation engine implementing novel trust calculation algorithms; c) an anomaly detection module for identifying suspicious trust patterns; d) a real-time update mechanism for maintaining current trust states.

Claim 5: The system of claim 4, further comprising: a) trust community detection algorithms for identifying agent clusters; b) performance optimization mechanisms for large-scale network analysis; c) distributed computation capabilities for horizontal scaling; d) temporal trust evolution tracking and analysis.

Patent Application 2: Multi-Agent Orchestration Security Algorithms

TITLE

System and Method for Secure Orchestration and Coordination Validation in Multi-Agent Artificial Intelligence Systems

TECHNICAL FIELD

This invention relates to cybersecurity systems for artificial intelligence applications, and more specifically to methods and systems for securing, validating, and monitoring orchestration patterns in multi-agent AI systems to prevent malicious coordination and ensure secure collaborative behaviors.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention addresses the emerging security challenges in multi-agent AI systems where autonomous agents coordinate to accomplish complex tasks. As AI agents become more sophisticated and autonomous, the need for secure orchestration mechanisms becomes critical to prevent malicious coordination patterns and ensure trustworthy multi-agent operations.

Description of Related Art

Current multi-agent coordination systems focus primarily on efficiency and task completion without adequate security considerations:

1. **Traditional Orchestration Systems**: Existing systems prioritize task efficiency without considering security implications of agent coordination patterns.

2. **Basic Agent Monitoring**: Current solutions monitor individual agents but lack comprehensive orchestration pattern analysis.
3. **Static Permission Models**: Existing systems use fixed permission structures that cannot adapt to dynamic orchestration requirements.

Problems with Prior Art:

- Inability to validate security of complex orchestration patterns
- Lack of real-time monitoring for malicious coordination detection
- Absence of dynamic permission models for orchestration contexts
- No consideration of collective agent behavior security implications
- Limited capability to prevent coordinated attacks across agent networks

SUMMARY OF THE INVENTION

The present invention provides a comprehensive system and method for securing multi-agent orchestration through pattern validation, real-time monitoring, and dynamic security policy enforcement. The system analyzes orchestration plans before execution, monitors coordination patterns during execution, and adapts security policies based on observed behaviors.

Key Technical Contributions:

1. **Orchestration Pattern Security Validation**: Novel algorithms for analyzing proposed orchestration patterns and identifying potential security risks.
2. **Real-Time Coordination Monitoring**: Advanced monitoring system that tracks agent coordination behaviors and detects malicious patterns.
3. **Dynamic Security Policy Enforcement**: Adaptive policy system that adjusts security controls based on orchestration context and risk assessment.
4. **Coordinated Attack Prevention**: Specialized detection and prevention mechanisms for multi-agent attacks and malicious coordination.

DETAILED DESCRIPTION OF THE INVENTION

System Architecture

The multi-agent orchestration security system comprises:

1. **Orchestration Plan Analyzer**: Pre-execution validation system for orchestration patterns.
2. **Coordination Monitor**: Real-time tracking system for agent coordination behaviors.
3. **Dynamic Policy Engine**: Adaptive security policy system for orchestration contexts.

4. **Attack Prevention Module**: Specialized system for detecting and preventing coordinated attacks.

Core Algorithms

Algorithm 1: Orchestration Pattern Security Validation

```
python
```

```python
def validate_orchestration_security(orchestration_plan, security_context):
    """
    Validate security implications of proposed orchestration pattern

    Args:
        orchestration_plan: Detailed plan for agent coordination
        security_context: Current security state and constraints

    Returns:
        OrchestrationValidation: Security assessment with risk scores
    """
    validation_result = OrchestrationValidation()

    # Analyze coordination complexity
    complexity_score = calculate_coordination_complexity(orchestration_plan)
    validation_result.complexity_risk = assess_complexity_risk(complexity_score)

    # Validate participant trust levels
    participant_trust = evaluate_participant_trust(
        orchestration_plan.participants,
        security_context.trust_requirements
    )
    validation_result.trust_assessment = participant_trust

    # Check for suspicious coordination patterns
    pattern_analysis = analyze_coordination_patterns(orchestration_plan)
    validation_result.pattern_risks = identify_pattern_risks(pattern_analysis)

    # Evaluate resource access implications
    resource_analysis = analyze_resource_access(
        orchestration_plan.resource_requirements,
        orchestration_plan.participants
    )
    validation_result.resource_risks = assess_resource_risks(resource_analysis)

    # Calculate overall security score
    validation_result.overall_score = calculate_security_score([
        validation_result.complexity_risk,
        validation_result.trust_assessment,
        validation_result.pattern_risks,
        validation_result.resource_risks
    ])
    return validation_result
```

Algorithm 2: Real-Time Coordination Monitoring

```
python
```

```
def monitor_coordination_execution(orchestration_id, monitoring_config):
    """
    Monitor orchestration execution for security anomalies

    Args:
        orchestration_id: Unique identifier for orchestration instance
        monitoring_config: Configuration for monitoring parameters

    Returns:
        CoordinationMonitoringResult: Real-time security assessment
    """
    monitoring_result = CoordinationMonitoringResult()

    # Track agent interaction patterns
    interaction_patterns = track_agent_interactions(orchestration_id)

    # Detect deviation from expected patterns
    pattern_deviations = detect_pattern_deviations(
        interaction_patterns,
        get_expected_patterns(orchestration_id)
    )
    monitoring_result.pattern_deviations = pattern_deviations

    # Monitor resource usage anomalies
    resource_usage = monitor_resource_consumption(orchestration_id)
    usage_anomalies = detect_usage_anomalies(resource_usage)
    monitoring_result.resource_anomalies = usage_anomalies

    # Track communication security
    communication_analysis = analyze_inter_agent_communication(orchestration_id)
    monitoring_result.communication_security = communication_analysis

    # Detect coordinated malicious behavior
    malicious_indicators = detect_coordinated_threats(
        interaction_patterns,
        resource_usage,
        communication_analysis
    )
    monitoring_result.threat_indicators = malicious_indicators

    # Calculate real-time risk score
    monitoring_result.current_risk_score = calculate_real_time_risk([
        pattern_deviations,
        usage_anomalies,
        communication_analysis.security_score,
        malicious_indicators
    ])

    return monitoring_result
```

Algorithm 3: Dynamic Security Policy Enforcement

```
python
```

```python
def enforce_dynamic_orchestration_policies(orchestration_context, policy_framework):
    """
    Dynamically enforce security policies based on orchestration context

    Args:
        orchestration_context: Current state of orchestration execution
        policy_framework: Available security policies and rules

    Returns:
        PolicyEnforcementResult: Applied policies and enforcement actions
    """
    enforcement_result = PolicyEnforcementResult()

    # Evaluate current orchestration risk level
    current_risk = assess_current_orchestration_risk(orchestration_context)

    # Select applicable policies based on context
    applicable_policies = select_context_policies(
        orchestration_context,
        policy_framework,
        current_risk
    )

    # Apply dynamic permission adjustments
    permission_adjustments = calculate_permission_adjustments(
        orchestration_context.participants,
        current_risk,
        applicable_policies
    )
    enforcement_result.permission_changes = apply_permission_changes(
        permission_adjustments
    )

    # Implement communication restrictions
    communication_policies = derive_communication_policies(
        orchestration_context,
        current_risk
    )
    enforcement_result.communication_restrictions = apply_communication_policies(
        communication_policies
    )

    # Establish monitoring intensity
    monitoring_requirements = calculate_monitoring_requirements(
        current_risk,
        orchestration_context.complexity
    )
    enforcement_result.monitoring_adjustments = apply_monitoring_changes(
        monitoring_requirements
    )

    # Set up automated response triggers
    response_triggers = configure_response_triggers(
        current_risk,
        orchestration_context,
        applicable_policies
    )
    enforcement_result.response_configuration = response_triggers

    return enforcement_result
```

Novel Technical Features

1. Orchestration Pattern Fingerprinting The system creates unique fingerprints for orchestration patterns:

- Pattern topology analysis
- Behavioral signature extraction
- Temporal coordination mapping
- Resource access pattern identification

2. Multi-Dimensional Risk Assessment Comprehensive risk evaluation considering:

- Agent trust levels and histories
- Coordination complexity metrics
- Resource access implications
- Communication pattern analysis
- Historical orchestration outcomes

3. Adaptive Security Boundaries Dynamic security perimeter adjustment:

- Context-aware permission modification
- Real-time policy adaptation
- Risk-based access control

- Automated security escalation

Implementation Details

Orchestration Security Database Schema:

```sql
CREATE TABLE orchestration_patterns (
  id UUID PRIMARY KEY,
  pattern_name VARCHAR(255),
  pattern_fingerprint VARCHAR(128),
  complexity_score DECIMAL(3,2),
  risk_assessment JSONB,
  security_requirements JSONB,
  created_at TIMESTAMP
);

CREATE TABLE coordination_monitoring (
  id UUID PRIMARY KEY,
  orchestration_id UUID,
  monitoring_timestamp TIMESTAMP,
  interaction_patterns JSONB,
  anomaly_indicators JSONB,
  risk_score DECIMAL(3,2),
  automated_actions JSONB
);
```

Performance Characteristics:

- Pattern validation: <100ms for complex orchestrations
- Real-time monitoring: <50ms update intervals
- Policy enforcement: <25ms for policy application
- Scalability: 1000+ concurrent orchestrations

CLAIMS

Claim 1: A computer-implemented method for securing multi-agent orchestration in artificial intelligence systems, comprising: a) analyzing proposed orchestration patterns for security risks before execution; b) monitoring agent coordination behaviors in real-time during orchestration execution; c) dynamically enforcing security policies based on orchestration context and risk assessment; d) detecting and preventing coordinated malicious activities across agent networks.

Claim 2: The method of claim 1, wherein orchestration pattern analysis includes: a) calculating coordination complexity scores; b) evaluating participant trust levels and capabilities; c) identifying suspicious coordination patterns; d) assessing resource access security implications.

Claim 3: The method of claim 1, wherein real-time monitoring comprises: a) tracking agent interaction patterns and deviations; b) monitoring resource usage anomalies; c) analyzing inter-agent communication security; d) detecting coordinated threat indicators.

Claim 4: A system for multi-agent orchestration security comprising: a) an orchestration plan analyzer for pre-execution security validation; b) a coordination monitor for real-time behavior tracking; c) a dynamic policy engine for context-aware security enforcement; d) an attack prevention module for coordinated threat detection.

Claim 5: The system of claim 4, further comprising: a) pattern fingerprinting algorithms for orchestration identification; b) multi-dimensional risk assessment capabilities; c) adaptive security boundary management; d) automated response and mitigation mechanisms.

Patent Application 3: Framework-Native Security Integration Methods

TITLE

System and Method for Framework-Native Security Integration in Artificial Intelligence Agent Development Platforms

TECHNICAL FIELD

This invention relates to cybersecurity integration for artificial intelligence development frameworks, and specifically to methods and systems for embedding security controls directly into AI agent development frameworks such as LangChain, AutoGPT, and CrewAI to provide seamless, transparent security without disrupting agent functionality.

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention addresses the critical need for seamless security integration in AI agent development frameworks. As AI agents become more prevalent and sophisticated, there is an urgent need for security systems that integrate natively with popular development frameworks rather than operating as external security layers that can be bypassed or circumvented.

Description of Related Art

Current AI agent security approaches primarily rely on external monitoring and control systems:

1. **External Security Wrappers**: Existing solutions implement security as external layers around AI agents, creating potential bypass vulnerabilities and integration friction.
2. **Framework-Agnostic Security**: Current security systems treat all AI agents generically without considering framework-specific behaviors and vulnerabilities.
3. **Bolt-On Security Solutions**: Traditional approaches add security as an afterthought, leading to performance penalties and development workflow disruption.

Problems with Prior Art:

- Security controls can be bypassed by modifying agent implementation
- Performance overhead from external security monitoring
- Developer workflow disruption and reduced productivity
- Lack of framework-specific security understanding
- Inability to provide proactive security guidance during development
- Limited visibility into framework-internal operations

SUMMARY OF THE INVENTION

The present invention provides novel methods and systems for embedding security controls directly into AI agent development frameworks at the source code level, creating framework-native security that is transparent to developers while providing comprehensive protection that cannot be bypassed or disabled.

Key Technical Contributions:

1. **Source-Level Security Integration**: Novel techniques for embedding security controls directly into framework source code without modifying public APIs.
2. **Framework-Specific Security Adaptation**: Specialized security implementations tailored to each framework's architecture and execution patterns.
3. **Transparent Security Middleware**: Security layers that operate invisibly within framework execution flows without impacting developer experience.
4. **Proactive Security Guidance**: Development-time security analysis and guidance integrated into framework tooling.

DETAILED DESCRIPTION OF THE INVENTION

System Architecture

The framework-native security integration system comprises:

1. **Framework Integration Engine**: Core system for embedding security into framework source code.
2. **Security Middleware Layer**: Transparent security operations within framework execution flows.
3. **Framework-Specific Adapters**: Specialized security implementations for different frameworks.
4. **Development-Time Security Analysis**: Proactive security guidance during agent development.

Core Algorithms

Algorithm 1: Dynamic Security Injection

```python
```

```python
def inject_security_middleware(framework_component, security_config):
    """
    Dynamically inject security controls into framework components

    Args:
        framework_component: Framework class or method to secure
        security_config: Security configuration and policies

    Returns:
        SecureComponent: Framework component with embedded security
    """
    # Create security wrapper that preserves original interface
    security_wrapper = create_transparent_wrapper(framework_component)

    # Inject pre-execution security checks
    security_wrapper.add_pre_execution_hook(
        lambda *args, **kwargs: validate_execution_security(
            framework_component,
            args,
            kwargs,
            security_config
        )
    )

    # Inject post-execution security analysis
    security_wrapper.add_post_execution_hook(
        lambda result, *args, **kwargs: analyze_execution_result(
            framework_component,
            result,
            args,
            kwargs,
            security_config
        )
    )

    # Add runtime monitoring
    security_wrapper.add_runtime_monitor(
        create_runtime_monitor(framework_component, security_config)
    )

    # Preserve all original methods and properties
    security_wrapper.preserve_original_interface(framework_component)
    return security_wrapper
```

Algorithm 2: Framework-Specific Security Adaptation

```
python
```

```python
def adapt_security_for_framework(framework_type, agent_config, security_requirements):
    """
    Adapt security implementation for specific framework characteristics

    Args:
        framework_type: Type of AI agent framework (LangChain, AutoGPT, etc.)
        agent_config: Agent configuration and capabilities
        security_requirements: Required security controls

    Returns:
        FrameworkSecurityAdapter: Framework-specific security implementation
    """
    adapter = create_base_adapter(framework_type)

    if framework_type == "langchain":
        # LangChain-specific security adaptations
        adapter.add_chain_security_validation()
        adapter.add_tool_usage_monitoring()
        adapter.add_memory_protection()
        adapter.add_llm_call_interception()

        # Secure LangChain-specific components
        adapter.secure_component("AgentExecutor", langchain_executor_security)
        adapter.secure_component("Chain", langchain_chain_security)
        adapter.secure_component("Tool", langchain_tool_security)

    elif framework_type == "autogpt":
        # AutoGPT-specific security adaptations
        adapter.add_goal_validation()
        adapter.add_action_interception()
        adapter.add_resource_monitoring()
        adapter.add_planning_security()

        # Secure AutoGPT-specific components
        adapter.secure_component("Agent", autogpt_agent_security)
        adapter.secure_component("ActionRegistry", autogpt_action_security)
        adapter.secure_component("Memory", autogpt_memory_security)

    elif framework_type == "crewai":
        # CrewAI-specific security adaptations
        adapter.add_crew_coordination_security()
        adapter.add_agent_role_validation()
        adapter.add_task_delegation_monitoring()
        adapter.add_collaboration_analysis()

        # Secure CrewAI-specific components
        adapter.secure_component("Crew", crewai_crew_security)
        adapter.secure_component("Agent", crewai_agent_security)
        adapter.secure_component("Task", crewai_task_security)

    # Apply security requirements to adapter
    adapter.configure_security_requirements(security_requirements)

    # Optimize adapter for framework-specific performance characteristics
    adapter.optimize_for_framework(framework_type, agent_config)

    return adapter
```

Algorithm 3: Transparent Security Middleware

```
python
```

```
def create_transparent_security_middleware(framework_method, security_context):
    """
    Create security middleware that operates transparently within framework

    Args:
        framework_method: Original framework method to secure
        security_context: Security context and configuration

    Returns:
        SecureMethodWrapper: Transparent security-enabled method
    """
    @functools.wraps(framework_method)
    async def secure_method_wrapper(*args, **kwargs):
        # Pre-execution security validation
        security_check = await validate_method_security(
            framework_method,
            args,
            kwargs,
            security_context
        )

        if not security_check.approved:
            if security_check.requires_approval:
                # Request human approval for risky operations
                approval = await request_security_approval(
                    framework_method,
                    args,
                    kwargs,
                    security_check.risk_factors
                )
                if not approval.granted:
                    raise SecurityException(f"Operation denied: {approval.reason}")
            else:
                raise SecurityException(f"Security violation: {security_check.reason}")

        # Execute with security monitoring
        start_time = time.time()
        execution_monitor = start_execution_monitoring(
            framework_method,
            args,
            kwargs,
            security_context
        )
        try:
            # Execute original method
            if asyncio.iscoroutinefunction(framework_method):
                result = await framework_method(*args, **kwargs)
            else:
                result = framework_method(*args, **kwargs)

            # Post-execution security analysis
            await analyze_execution_outcome(
                framework_method,
                result,
                args,
                kwargs,
                time.time() - start_time,
                security_context
            )

            return result

        except Exception as e:
            # Security analysis of exceptions
            await analyze_execution_exception(
                framework_method,
                e,
                args,
                kwargs,
                security_context
            )
            raise

        finally:
            # Clean up monitoring
            stop_execution_monitoring(execution_monitor)

    # Preserve method metadata and documentation
    secure_method_wrapper.__name__ = framework_method.__name__
    secure_method_wrapper.__doc__ = framework_method.__doc__
    secure_method_wrapper.__annotations__ = getattr(framework_method, '__annotations__', {})

    return secure_method_wrapper
```

Novel Technical Features

1. Non-Intrusive Security Integration The system embeds security without modifying framework public APIs:

- Bytecode-level security injection
- Runtime method replacement with identical signatures
- Transparent execution flow modification
- Zero-impact developer experience

2. Framework-Aware Security Policies Security implementations tailored to framework-specific behaviors:

- LangChain chain execution security
- AutoGPT goal and action validation
- CrewAI multi-agent coordination security
- Framework-specific threat models

3. Development-Time Security Analysis Proactive security guidance during development:

- Static analysis of agent configurations
- Security recommendation engine
- Vulnerability detection in agent designs
- Best practice enforcement

Implementation Details

Security Integration Database Schema:

```sql
```

```
CREATE TABLE framework_integrations (
    id UUID PRIMARY KEY,
    framework_name VARCHAR(50) NOT NULL,
    framework_version VARCHAR(20),
    integration_version VARCHAR(20),
    security_components JSONB,
    performance_impact JSONB,
    integration_status VARCHAR(20),
    created_at TIMESTAMP
);

CREATE TABLE security_adaptations (
    id UUID PRIMARY KEY,
    framework_integration_id UUID REFERENCES framework_integrations(id),
    component_name VARCHAR(255),
    security_method VARCHAR(100),
    adaptation_config JSONB,
    effectiveness_metrics JSONB,
    created_at TIMESTAMP
);
```

Framework Integration Examples:

LangChain Integration:

```python
```

```
# Automatic security injection for LangChain AgentExecutor
from langchain.agents import AgentExecutor
from our_security_framework import secure_component

# Security is automatically injected when AgentExecutor is imported
@secure_component(
    monitor_tool_usage=True,
    validate_chain_execution=True,
    protect_memory_access=True
)
class SecureAgentExecutor(AgentExecutor):
    # All original functionality preserved
    # Security operates transparently
    pass

# Replace original AgentExecutor with secure version
langchain.agents.AgentExecutor = SecureAgentExecutor
```

AutoGPT Integration:

```python
# Transparent security for AutoGPT agent actions
from autogpt.agent import Agent
from our_security_framework import inject_security

# Security injection happens at import time
original_execute_action = Agent.execute_action

@inject_security(
    validate_goals=True,
    monitor_resources=True,
    intercept_actions=True
)
async def secure_execute_action(self, action):
    # Security validation happens transparently
    return await original_execute_action(self, action)

# Replace method with secure version
Agent.execute_action = secure_execute_action
```

CLAIMS

Claim 1: A computer-implemented method for framework-native security integration in AI agent development platforms, comprising: a) dynamically injecting security controls into framework source code at runtime; b) adapting security implementations to framework-specific architectures and execution patterns; c) providing transparent security middleware that operates invisibly within framework execution flows; d) offering development-time security analysis and guidance integrated into framework tooling.

Claim 2: The method of claim 1, wherein dynamic security injection includes: a) creating transparent wrappers that preserve original framework interfaces; b) injecting pre-execution and post-execution security hooks; c) adding runtime monitoring without modifying public APIs; d) preserving all original framework methods and properties.

Claim 3: The method of claim 1, wherein framework-specific adaptation comprises: a) implementing specialized security controls for LangChain chain execution; b) providing AutoGPT-specific goal validation and action interception; c) securing CrewAI multi-agent coordination and task delegation; d) optimizing security implementations for framework-specific performance characteristics.

Claim 4: A system for framework-native security integration comprising: a) a framework integration engine for embedding security into framework source code; b) a security middleware layer for transparent security operations; c) framework-specific adapters for specialized security implementations; d) a development-time security analysis system for proactive guidance.

Claim 5: The system of claim 4, further comprising: a) non-intrusive security integration mechanisms; b) framework-aware security policy engines; c) development-time vulnerability detection capabilities; d) automated security optimization and performance tuning.

Patent Filing Process and Recommendations

Filing Strategy

1. **Provisional Patent Applications First**
    - File provisional applications for all three inventions
    - Cost: ~$1,600 per application ($4,800 total)
    - Provides 12-month priority period
    - Allows "Patent Pending" status
2. **PCT (Patent Cooperation Treaty) Filing**
    - File within 12 months of provisional applications
    - Provides international patent protection
    - Cost: ~$4,000-6,000 per application

3. **National Phase Entry**
    - Enter specific countries within 30 months
    - Target: US, EU, China, Japan, Canada
    - Cost varies by country (~$3,000-8,000 per country per patent)

Total Investment Estimate

- **Year 1**: $15,000-25,000 (provisionals + PCT)
- **Year 2-3**: $45,000-75,000 (national phase entries)
- **Total**: $60,000-100,000 for comprehensive patent protection

Patent Attorney Recommendations

1. **Specialized IP Firms**:
    - Kilpatrick Townsend (AI/Software focus)
    - Wilson Sonsini (Tech startup expertise)
    - Fenwick & West (Silicon Valley presence)
2. **Selection Criteria**:
    - AI and cybersecurity patent experience
    - Software patent prosecution track record
    - Startup-friendly fee structures
    - International filing experience

Timeline Recommendations

Immediate (Next 30 days):

- Conduct prior art search for each invention
- Prepare detailed technical descriptions
- File provisional patent applications

3-6 Months:

- Refine patent applications based on initial feedback
- Conduct additional prior art analysis
- Prepare for PCT filing

12 Months:

- File PCT applications
- Begin national phase planning
- Evaluate patent landscape in target markets