PROVISIONAL PATENT APPLICATION

Title: Computational Time Dilation Security System Using CPU Priority Scheduling for Temporal Domain-Based Cybersecurity

Inventor(s): [To be filled]

Application Type: Provisional Patent Application

Filing Date: [To be filled]

Application Number: [To be assigned by USPTO]

TECHNICAL FIELD

This invention relates to cybersecurity systems that utilize computational time dilation through CPU priority scheduling to create different temporal processing domains, enabling security mechanisms that operate at different computational time scales to prevent attacks and provide temporal security isolation.

BACKGROUND OF THE INVENTION

Current Computing Resource Management

Traditional computing systems manage resources through:

1. Process Scheduling: Round-robin, priority-based, and fair-share scheduling algorithms

2. CPU Resource Allocation: Time slicing and priority queuing for process management

3. Memory Management: Virtual memory, paging, and memory isolation techniques

4. I/O Scheduling: Disk and network I/O prioritization and queuing

Limitations of Traditional Security Approaches

Temporal Attack Vulnerabilities:

- All security processes operate in the same temporal domain

- Attackers can exploit timing-based vulnerabilities

- No temporal isolation between security-critical and standard processes

- Race conditions in security-sensitive operations

Resource-Based Attack Vectors:

- CPU resource exhaustion attacks (DoS)

- Timing analysis attacks on cryptographic operations

- Side-channel attacks through timing measurement

- Performance degradation attacks on security processes

Lack of Temporal Security Domains:

- No differentiated temporal processing for security operations

- All processes compete equally for computational resources

- No temporal privilege escalation for security-critical functions

- Limited temporal isolation mechanisms

Prior Art Analysis

Priority-Based Scheduling: Standard CPU scheduling with priority levels but no temporal domain differentiation for security purposes.

Real-Time Computing: Hard and soft real-time systems with timing guarantees but no security-focused temporal domain isolation.

Quality of Service (QoS): Network and system QoS mechanisms but no computational time dilation for cybersecurity applications.

CPU Affinity and Isolation: Process isolation techniques but no temporal domain-based security mechanisms.

SUMMARY OF THE INVENTION

The present invention provides a Computational Time Dilation Security System that creates different temporal processing domains through CPU priority scheduling manipulation, enabling security processes to operate at accelerated computational speeds while constraining potential attackers to slower temporal domains, thereby creating temporal security advantages and attack prevention mechanisms.

Core Innovation Elements

1. Temporal Domain Controller: Creates and manages different computational time domains

2. CPU Priority Manipulation Engine: Dynamically adjusts process priorities for temporal effects

3. Security Process Acceleration: Accelerates security-critical processes through priority scheduling

4. Attack Process Deceleration: Slows down suspicious processes through resource constraint

5. Temporal Security Isolation: Provides temporal isolation between security domains

Technical Advantages

- Temporal Security Advantage: Security processes gain computational time advantages

- Attack Prevention Through Deceleration: Slows down attack processes to prevent success

- Resource-Based Defense: Uses computational resource control as security mechanism

- Temporal Isolation: Prevents timing-based attacks through domain isolation

- Adaptive Temporal Response: Dynamically adjusts temporal domains based on threat level

DETAILED DESCRIPTION OF THE INVENTION

System Architecture

The Computational Time Dilation Security System comprises five primary components:

1. Temporal Domain Manager - Creates and manages computational time domains

2. Process Classification Engine - Classifies processes for temporal domain assignment

3. CPU Priority Scheduler - Manipulates CPU scheduling for temporal effects

4. Security Process Accelerator - Provides temporal acceleration for security functions

5. Temporal Monitoring System - Monitors and validates temporal domain effectiveness

Component 1: Temporal Domain Manager

Purpose: Create and manage different computational temporal domains with distinct processing speeds and resource allocation characteristics.

Technical Implementation:

```python
import os
import psutil
import threading
import time
from enum import Enum
from typing import Dict, List, Optional
import subprocess
import logging
class TemporalDomain(Enum):
HYPERSPACE = "hyperspace" # 10x computational acceleration
REALTIME = "realtime" # Normal computational time
SLOWTIME = "slowtime" # 0.1x computational deceleration
QUARANTINE = "quarantine" # 0.01x extreme deceleration
class TemporalDomainManager:
def __init__(self):
self.temporal_domains = {
TemporalDomain.HYPERSPACE: {
'acceleration_factor': 10.0,
'cpu_priority': 'high',
'nice_value': -20,
'cpu_affinity': None, # All CPUs available
'memory_priority': 'high',
'io_priority': 'high'
},
TemporalDomain.REALTIME: {
```

```
'acceleration_factor': 1.0,
'cpu_priority': 'normal',
'nice_value': 0,
'cpu_affinity': None,
'memory_priority': 'normal',
'io_priority': 'normal'
},
TemporalDomain.SLOWTIME: {
'acceleration_factor': 0.1,
'cpu_priority': 'low',
'nice_value': 19,
'cpu_affinity': [0], # Limited to single CPU core
'memory_priority': 'low',
'io_priority': 'idle'
},
TemporalDomain.QUARANTINE: {
'acceleration_factor': 0.01,
'cpu_priority': 'lowest',
'nice_value': 19,
'cpu_affinity': [0], # Single CPU core
'memory_priority': 'lowest',
'io_priority': 'idle'
}
}
self.process_domain_assignments = {} # PID -> TemporalDomain
self.domain_monitoring = {}
self.temporal_effects_active = False
```

```
def initialize_temporal_domains(self):

"""Initialize temporal domain infrastructure"""
```

Verify system capabilities

```
system_capabilities = self.verify_system_capabilities()

if not system_capabilities['priority_scheduling']:

raise RuntimeError("System does not support priority scheduling")
```

Initialize domain monitoring

```
for domain in TemporalDomain:

self.domain_monitoring[domain] = {

'active_processes': 0,

'total_cpu_usage': 0.0,

'average_response_time': 0.0,

'temporal_effectiveness': 0.0

}
```

Start temporal effects monitoring

```
self.temporal_effects_active = True

monitoring_thread = threading.Thread(target=self._monitor_temporal_effects,
daemon=True)

monitoring_thread.start()

logging.info("Temporal domain system initialized")

def assign_process_to_domain(self, process_id: int, domain: TemporalDomain,

justification: str = ""):

"""Assign process to specific temporal domain"""

try:
```

Get process handle

process = psutil.Process(process_id)

Verify process exists and is accessible

if not process.is_running():

return {'success': False, 'reason': 'process_not_running'}

Apply temporal domain settings

domain_config = self.temporal_domains[domain]

Set CPU priority/nice value

if hasattr(process, 'nice'):

process.nice(domain_config['nice_value'])

Set CPU affinity if specified

if domain_config['cpu_affinity'] is not None:

if hasattr(process, 'cpu_affinity'):

process.cpu_affinity(domain_config['cpu_affinity'])

Set I/O priority (Linux-specific)

if hasattr(psutil, 'IOPRIO_CLASS_IDLE') and domain_config['io_priority'] == 'idle':

self._set_io_priority_idle(process_id)

Record assignment

self.process_domain_assignments[process_id] = {

'domain': domain,

```
'assignment_time': time.time(),
'justification': justification,
'original_priority': process.nice() if hasattr(process, 'nice') else 0
}
```

Update domain monitoring

```
self.domain_monitoring[domain]['active_processes'] += 1
logging.info(f"Process {process_id} assigned to {domain.value} domain: {justification}")
return {
'success': True,
'process_id': process_id,
'domain': domain.value,
'temporal_acceleration': domain_config['acceleration_factor'],
'assignment_time': time.time()
}
except psutil.NoSuchProcess:
return {'success': False, 'reason': 'process_not_found'}
except psutil.AccessDenied:
return {'success': False, 'reason': 'insufficient_privileges'}
except Exception as e:
return {'success': False, 'reason': str(e)}
def create_security_hyperspace(self, security_process_ids: List[int]):
"""Create hyperspace temporal domain for security processes"""
hyperspace_results = []
for process_id in security_process_ids:
```

Assign to hyperspace domain

```
assignment_result = self.assign_process_to_domain(
process_id,
TemporalDomain.HYPERSPACE,
"Critical security process requiring temporal acceleration"
)
```

Apply additional hyperspace optimizations

```
if assignment_result['success']:
hyperspace_optimizations =
self._apply_hyperspace_optimizations(process_id)
assignment_result['hyperspace_optimizations'] = hyperspace_optimizations
hyperspace_results.append(assignment_result)
return {
'hyperspace_created': True,
'security_processes': len(security_process_ids),
'successful_assignments': sum(1 for r in hyperspace_results if r['success']),
'assignment_details': hyperspace_results
}
def _apply_hyperspace_optimizations(self, process_id: int):
"""Apply additional optimizations for hyperspace processes"""
optimizations = {
'cpu_affinity_optimization': False,
'memory_priority_boost': False,
'cache_optimization': False
}
try:
process = psutil.Process(process_id)
```

CPU affinity optimization - assign to fastest cores

```
if hasattr(process, 'cpu_affinity'):
fastest_cores = self._identify_fastest_cpu_cores()
if fastest_cores:
process.cpu_affinity(fastest_cores)
optimizations['cpu_affinity_optimization'] = True
```

Memory priority boost (Windows-specific)

```
if os.name == 'nt':
try:
subprocess.run([
'wmic', 'process', 'where', f'processid={process_id}',
'CALL', 'setpriority', 'high priority'
], check=True, capture_output=True)
optimizations['memory_priority_boost'] = True
except subprocess.CalledProcessError:
pass
```

Cache optimization hints

```
if hasattr(process, 'memory_info'):
```

Attempt to optimize memory access patterns

```
optimizations['cache_optimization'] = True
except Exception as e:
logging.warning(f"Failed to apply hyperspace optimizations: {str(e)}")
return optimizations
```

```
def create_quarantine_slowtime(self, suspicious_process_ids: List[int]):
"""Create quarantine temporal domain for suspicious processes"""
quarantine_results = []
for process_id in suspicious_process_ids:
```

Assign to quarantine domain

```
assignment_result = self.assign_process_to_domain(
process_id,
TemporalDomain.QUARANTINE,
"Suspicious process requiring temporal isolation and deceleration"
)
```

Apply additional quarantine restrictions

```
if assignment_result['success']:
quarantine_restrictions = self._apply_quarantine_restrictions(process_id)
assignment_result['quarantine_restrictions'] = quarantine_restrictions
quarantine_results.append(assignment_result)
return {
'quarantine_created': True,
'quarantined_processes': len(suspicious_process_ids),
'successful_quarantines': sum(1 for r in quarantine_results if r['success']),
'quarantine_details': quarantine_results
}
def _apply_quarantine_restrictions(self, process_id: int):
"""Apply additional restrictions for quarantined processes"""
restrictions = {
'cpu_throttling': False,
'memory_limitation': False,
'io_throttling': False,
'network_limitation': False
}
try:
process = psutil.Process(process_id)
```

CPU throttling - limit to single slow core

```
if hasattr(process, 'cpu_affinity'):
slowest_cores = self._identify_slowest_cpu_cores()
if slowest_cores:
process.cpu_affinity([slowest_cores[0]]) # Single slowest core
restrictions['cpu_throttling'] = True
```

Memory limitation

```
if hasattr(process, 'memory_limit'):
```

Limit memory to 100MB if possible

```
try:
process.memory_limit(100 1024 1024) # 100MB
restrictions['memory_limitation'] = True
except AttributeError:
pass
```

I/O throttling

```
try:
self._set_io_priority_idle(process_id)
```

```
restrictions['io_throttling'] = True
except Exception:
pass
except Exception as e:
logging.warning(f"Failed to apply quarantine restrictions: {str(e)}")
return restrictions
def _monitor_temporal_effects(self):
"""Monitor effectiveness of temporal domain assignments"""
while self.temporal_effects_active:
for domain in TemporalDomain:
domain_processes = [
pid for pid, assignment in self.process_domain_assignments.items()
if assignment['domain'] == domain
]
if domain_processes:
```

Calculate domain effectiveness metrics

```
cpu_usage = self._calculate_domain_cpu_usage(domain_processes)
response_time = self._calculate_domain_response_time(domain_processes)
temporal_effectiveness = self._calculate_temporal_effectiveness(domain,
domain_processes)
self.domain_monitoring[domain].update({
'active_processes': len(domain_processes),
'total_cpu_usage': cpu_usage,
'average_response_time': response_time,
'temporal_effectiveness': temporal_effectiveness
})
```

```
time.sleep(5) # Monitor every 5 seconds
def get_temporal_status_report(self):
"""Generate comprehensive temporal domain status report"""
status_report = {
'report_timestamp': time.time(),
'temporal_domains_active': self.temporal_effects_active,
'total_managed_processes': len(self.process_domain_assignments),
'domain_statistics': {}
}
for domain, monitoring_data in self.domain_monitoring.items():
domain_config = self.temporal_domains[domain]
status_report['domain_statistics'][domain.value] = {
'acceleration_factor': domain_config['acceleration_factor'],
'active_processes': monitoring_data['active_processes'],
'cpu_usage_percentage': monitoring_data['total_cpu_usage'],
'average_response_time_ms': monitoring_data['average_response_time'],
'temporal_effectiveness_score': monitoring_data['temporal_effectiveness'],
'domain_health': self._assess_domain_health(domain)
}
return status_report
class ProcessClassificationEngine:
def __init__(self):
self.classification_rules = {
'security_critical': [
'antivirus', 'firewall', 'endpoint_protection', 'siem',
'intrusion_detection', 'authentication', 'encryption'
],
'system_critical': [
'kernel', 'system', 'driver', 'service_host', 'registry'
],
'suspicious_patterns': [
'cryptocurrency_miner', 'keylogger', 'remote_access_tool',
'network_scanner', 'password_cracker', 'exploit'
],
'resource_intensive': [
'compiler', 'video_encoder', 'machine_learning', 'simulation'
]
}
self.behavioral_indicators = {
'attack_indicators': [
'high_network_activity', 'suspicious_file_access',
'registry_modification', 'privilege_escalation_attempt',
'encryption_activity', 'data_exfiltration_pattern'
],
'security_indicators': [
'log_analysis', 'threat_scanning', 'policy_enforcement',
'authentication_processing', 'certificate_validation'
]
}
def classify_process_for_temporal_domain(self, process_info: Dict) ->
TemporalDomain:
"""Classify process and recommend temporal domain assignment"""
process_name = process_info.get('name', '').lower()
process_cmd = process_info.get('cmdline', '').lower()
```

```
process_behavior = process_info.get('behavioral_indicators', [])
```

Security-critical processes -> Hyperspace

```
if self._matches_classification(process_name, process_cmd, 'security_critical'):

return TemporalDomain.HYPERSPACE
```

Suspicious processes -> Quarantine

```
if self._matches_classification(process_name, process_cmd, 'suspicious_patterns'):

return TemporalDomain.QUARANTINE
```

Attack behavior indicators -> Slowtime or Quarantine

```
if any(indicator in process_behavior for indicator in self.behavioral_indicators['attack_indicators']):

attack_severity = self._assess_attack_severity(process_behavior)

if attack_severity > 0.8:

return TemporalDomain.QUARANTINE

else:

return TemporalDomain.SLOWTIME
```

Security behavior indicators -> Hyperspace

```
if any(indicator in process_behavior for indicator in self.behavioral_indicators['security_indicators']):

return TemporalDomain.HYPERSPACE
```

Default assignment

```
return TemporalDomain.REALTIME

def _matches_classification(self, process_name: str, process_cmd: str, classification: str) -> bool:
```

```
"""Check if process matches classification criteria"""
classification_patterns = self.classification_rules.get(classification, [])
for pattern in classification_patterns:
if pattern in process_name or pattern in process_cmd:
return True
return False
def _assess_attack_severity(self, behavioral_indicators: List[str]) -> float:
"""Assess severity of attack-related behavioral indicators"""
high_severity_indicators = [
'data_exfiltration_pattern', 'privilege_escalation_attempt',
'encryption_activity', 'suspicious_network_communication'
]
medium_severity_indicators = [
'registry_modification', 'suspicious_file_access',
'high_network_activity'
]
severity_score = 0.0
for indicator in behavioral_indicators:
if indicator in high_severity_indicators:
severity_score += 0.4
elif indicator in medium_severity_indicators:
severity_score += 0.2
else:
severity_score += 0.1
return min(1.0, severity_score)
```

Component 2: Security Process Accelerator

Purpose: Provide temporal acceleration for security-critical processes through advanced CPU scheduling and resource allocation optimization.

Technical Implementation:

```python
class SecurityProcessAccelerator:

def __init__(self, temporal_domain_manager: TemporalDomainManager):

self.temporal_manager = temporal_domain_manager

self.accelerated_processes = {}

self.acceleration_metrics = {}

def accelerate_security_process(self, process_id: int, acceleration_level: str = "high") -> Dict:

"""Accelerate security process through temporal domain assignment"""

acceleration_config = {

'low': {'domain': TemporalDomain.REALTIME, 'priority_boost': 5},

'high': {'domain': TemporalDomain.HYPERSPACE, 'priority_boost': 15},

'critical': {'domain': TemporalDomain.HYPERSPACE, 'priority_boost': 20}

}

config = acceleration_config.get(acceleration_level, acceleration_config['high'])
```

Measure baseline performance

```python
baseline_metrics = self._measure_process_performance(process_id)
```

Apply temporal domain assignment

```python
domain_assignment = self.temporal_manager.assign_process_to_domain(

process_id, config['domain'], f"Security process acceleration: {acceleration_level}"

)

if domain_assignment['success']:
```

Apply additional acceleration techniques

additional_acceleration = self._apply_additional_acceleration(

process_id, config['priority_boost']

)

Measure accelerated performance

accelerated_metrics = self._measure_process_performance(process_id)

Calculate acceleration effectiveness

acceleration_effectiveness = self._calculate_acceleration_effectiveness(

baseline_metrics, accelerated_metrics

)

Record acceleration details

self.accelerated_processes[process_id] = {

'acceleration_level': acceleration_level,

'domain': config['domain'],

'baseline_metrics': baseline_metrics,

'accelerated_metrics': accelerated_metrics,

'effectiveness': acceleration_effectiveness,

'acceleration_timestamp': time.time()

}

return {

'success': True,

'process_id': process_id,

'acceleration_level': acceleration_level,

```
'temporal_domain': config['domain'].value,

'baseline_performance': baseline_metrics,

'accelerated_performance': accelerated_metrics,

'acceleration_factor': acceleration_effectiveness['overall_improvement'],

'additional_optimizations': additional_acceleration

}

else:

return {

'success': False,

'reason': domain_assignment['reason'],

'process_id': process_id

}

def _apply_additional_acceleration(self, process_id: int, priority_boost: int) -> Dict:

"""Apply additional acceleration techniques beyond temporal domain assignment"""

optimizations = {

'cpu_cache_optimization': False,

'memory_prefetching': False,

'interrupt_priority': False,

'scheduler_optimization': False

}

try:

process = psutil.Process(process_id)
```

CPU cache optimization

```
if hasattr(process, 'cpu_affinity'):
```

Assign to cores with largest cache

```
cache_optimized_cores = self._get_cache_optimized_cores()
if cache_optimized_cores:
process.cpu_affinity(cache_optimized_cores)
optimizations['cpu_cache_optimization'] = True
```

Memory prefetching hints (Linux-specific)

```
if hasattr(os, 'posix_madvise'):
try:
```

Suggest sequential memory access patterns

```
memory_regions = process.memory_maps()
for region in memory_regions[:5]: # Optimize top 5 regions
os.posix_madvise(region.addr, region.size, os.MADV_SEQUENTIAL)
optimizations['memory_prefetching'] = True
except (AttributeError, OSError):
pass
```

Scheduler optimization

```
if os.name == 'posix':
try:
```

Set real-time scheduling policy for critical security processes

```
import sched
sched.sched_setscheduler(process_id, sched.SCHED_FIFO, sched.sched_param(priority_boost))
```

```
optimizations['scheduler_optimization'] = True
except (ImportError, OSError):
pass
except Exception as e:
logging.warning(f"Failed to apply additional acceleration: {str(e)}")
return optimizations
def _measure_process_performance(self, process_id: int) -> Dict:
"""Measure process performance metrics"""
try:
process = psutil.Process(process_id)
```

Collect performance metrics

```
cpu_percent = process.cpu_percent(interval=1.0)
memory_info = process.memory_info()
io_counters = process.io_counters() if hasattr(process, 'io_counters') else None
create_time = process.create_time()
performance_metrics = {
'cpu_usage_percent': cpu_percent,
'memory_rss_mb': memory_info.rss / (1024 1024),
'memory_vms_mb': memory_info.vms / (1024 1024),
'process_age_seconds': time.time() - create_time,
'measurement_timestamp': time.time()
}
if io_counters:
performance_metrics.update({
'io_read_bytes': io_counters.read_bytes,
'io_write_bytes': io_counters.write_bytes,
'io_read_count': io_counters.read_count,
'io_write_count': io_counters.write_count
})
```

Response time measurement (approximate)

```
response_start = time.time()
try:
process.status() # Simple process query
response_time = (time.time() - response_start) 1000 # milliseconds
performance_metrics['response_time_ms'] = response_time
except Exception:
performance_metrics['response_time_ms'] = None
return performance_metrics
except psutil.NoSuchProcess:
return {'error': 'process_not_found'}
except Exception as e:
return {'error': str(e)}
def _calculate_acceleration_effectiveness(self, baseline: Dict, accelerated: Dict) -> Dict:
"""Calculate effectiveness of acceleration techniques"""
if 'error' in baseline or 'error' in accelerated:
return {'overall_improvement': 0.0, 'error': 'measurement_failed'}
effectiveness = {}
```

CPU usage efficiency (lower is better for most cases)

```
if 'cpu_usage_percent' in baseline and 'cpu_usage_percent' in accelerated:
cpu_improvement = baseline['cpu_usage_percent'] / max(0.1, accelerated['cpu_usage_percent'])
```

effectiveness['cpu_efficiency_improvement'] = cpu_improvement

Response time improvement (lower is better)

if (baseline.get('response_time_ms') and accelerated.get('response_time_ms') and

baseline['response_time_ms'] > 0 and accelerated['response_time_ms'] > 0):

response_improvement = baseline['response_time_ms'] / accelerated['response_time_ms']

effectiveness['response_time_improvement'] = response_improvement

Memory usage efficiency

if 'memory_rss_mb' in baseline and 'memory_rss_mb' in accelerated:

if accelerated['memory_rss_mb'] > 0:

memory_efficiency = baseline['memory_rss_mb'] / accelerated['memory_rss_mb']

effectiveness['memory_efficiency'] = memory_efficiency

I/O throughput improvement

if (baseline.get('io_read_bytes') is not None and accelerated.get('io_read_bytes') is not None):

time_diff = accelerated['measurement_timestamp'] - baseline['measurement_timestamp']

if time_diff > 0:

baseline_io_rate = baseline.get('io_read_bytes', 0) / time_diff

accelerated_io_rate = accelerated.get('io_read_bytes', 0) / time_diff

if baseline_io_rate > 0:

io_improvement = accelerated_io_rate / baseline_io_rate

effectiveness['io_throughput_improvement'] = io_improvement

Overall improvement score

```
improvements = [v for k, v in effectiveness.items() if k != 'error' and v > 0]

if improvements:

effectiveness['overall_improvement'] = sum(improvements) / len(improvements)

else:

effectiveness['overall_improvement'] = 1.0 # No change

return effectiveness
```

CLAIMS

Independent Claims

Claim 1: A computer-implemented computational time dilation security method comprising:

- creating multiple temporal processing domains with different computational acceleration factors through CPU priority scheduling manipulation;

- classifying processes based on security criticality and threat indicators for temporal domain assignment;

- accelerating security-critical processes through assignment to hyperspace temporal domains with enhanced CPU priority and resource allocation;

- decelerating suspicious and potentially malicious processes through assignment to quarantine temporal domains with restricted computational resources;

- monitoring temporal domain effectiveness and adjusting domain assignments dynamically based on security requirements and system performance.

Claim 2: A computational time dilation security system comprising:

- a temporal domain manager configured to create and manage different computational time domains;

- a process classification engine configured to analyze processes for temporal domain assignment based on security characteristics;

- a CPU priority scheduler configured to manipulate process scheduling for temporal acceleration and deceleration effects;

- a security process accelerator configured to provide temporal advantages to security-critical functions;

- a temporal monitoring system configured to validate temporal domain effectiveness and security benefits.

Claim 3: A method for temporal security isolation comprising:

- establishing computational time domains that provide temporal isolation between security processes and potential threats;

- implementing CPU scheduling-based time dilation to create security advantages through temporal processing speed differences;

- providing attack prevention through computational resource constraint and temporal deceleration of suspicious processes;

- enabling adaptive temporal security responses based on real-time threat assessment and system conditions.

Dependent Claims

Claim 4: The method of claim 1, wherein temporal domains include hyperspace (10x acceleration), realtime (1x normal), slowtime (0.1x deceleration), and quarantine (0.01x extreme deceleration) with corresponding CPU priority and resource allocation settings.

Claim 5: The system of claim 2, wherein process classification includes security-critical patterns, suspicious behavior indicators, attack severity assessment, and behavioral analysis for temporal domain recommendation.

Claim 6: The method of claim 3, wherein security process acceleration includes CPU cache optimization, memory prefetching, interrupt priority adjustment, and real-time scheduling policy assignment.

Claim 7: The system of claim 2, wherein temporal monitoring includes domain effectiveness measurement, process performance tracking, acceleration factor validation, and dynamic domain adjustment.

Claim 8: The method of claim 1, wherein quarantine restrictions include CPU throttling to single slow cores, memory limitation, I/O throttling to idle priority, and network access limitation.

Claim 9: The system of claim 2, wherein CPU priority manipulation includes nice value adjustment, CPU affinity optimization, I/O priority scheduling, and memory priority boost mechanisms.

Claim 10: The method of claim 3, further comprising integration with quantum-safe physical impossibility architectures and neural behavioral authentication engines for comprehensive temporal security isolation.

PROTOTYPE SYSTEM DESIGN AND PROJECTED PERFORMANCE

Temporal Domain Architecture Framework

Security Process Acceleration Design:

- Hyperspace Domain Acceleration: System designed to provide significant performance improvements for security processes through CPU priority optimization

- Response Time Optimization: Framework intended to substantially reduce response times for security-critical operations

- CPU Cache Optimization: Architecture designed to improve cache hit rates for accelerated processes

- Memory Access Efficiency: System intended to reduce memory access latency for hyperspace temporal domain processes

Attack Process Deceleration Framework:

- Quarantine Domain Effectiveness: System designed to substantially reduce computational capacity for quarantined processes

- Resource Constraint Impact: Framework intended to significantly limit CPU utilization by suspicious processes

- I/O Throttling Design: Architecture designed to substantially reduce disk I/O operations for quarantined processes

- Network Access Limitation: System intended to severely limit network activity for restricted processes

Security Benefits Framework

Temporal Security Architecture:

- Attack Prevention Design: System intended to prevent attacks through temporal deceleration mechanisms

- Security Response Enhancement: Framework designed to accelerate threat detection and response through hyperspace acceleration

- Resource Exhaustion Attack Mitigation: Architecture designed to reduce impact of CPU exhaustion attacks

- Timing Attack Prevention: System intended to reduce successful timing-based attacks

Adaptive Temporal Response Framework:

- Dynamic Domain Assignment: System designed for accurate automatic process classification for temporal domains

- Real-Time Domain Adjustment: Framework intended for dynamic temporal domain reassignment based on threat level changes

- False Positive Management: Architecture designed to minimize false positive rates in suspicious process identification

- Security Process Prioritization: System designed to maintain security process acceleration under high system loads

Performance Impact Framework

System Resource Utilization Design:

- CPU Overhead Management: System designed to minimize additional CPU usage for temporal domain management

- Memory Overhead Optimization: Framework intended to maintain low memory usage for temporal domain operations

- I/O Impact Minimization: Architecture designed to minimize additional disk I/O for temporal monitoring and logging

- Network Impact Design: System intended to maintain negligible network overhead for temporal domain operations

Scalability Architecture:

- Concurrent Process Management: System designed to manage multiple processes across temporal domains effectively

- Domain Switching Performance: Framework intended to achieve rapid temporal domain reassignment

- System Stability Design: Architecture designed for high uptime during continuous temporal domain operation

- Performance Consistency Framework: System intended to maintain consistent temporal acceleration factors under varying loads

Integration Framework

Security Framework Integration Design:

- SIEM Integration: System designed for integration with security information and event management systems

- Endpoint Protection Integration: Framework intended for integration with endpoint protection platforms

- Threat Detection Enhancement: Architecture designed to improve threat detection speed through temporal acceleration

- Incident Response Optimization: System intended to reduce incident response time through hyperspace domain utilization

INDUSTRIAL APPLICABILITY

Target Applications

Enterprise Security: Enhanced endpoint protection response, accelerated threat detection and analysis, improved security operation center (SOC) performance, and temporal isolation of suspicious processes.

Financial Services: High-frequency trading system protection, accelerated fraud detection processing, temporal isolation of financial transaction threats, and enhanced regulatory compliance processing.

Government and Defense: Accelerated classified system security processing, temporal isolation of potential insider threats, enhanced cybersecurity operation performance, and critical infrastructure protection acceleration.

Healthcare: Accelerated medical device security validation, temporal isolation of healthcare network threats, enhanced patient data protection processing, and medical IoT security acceleration.

Commercial Advantages

Security Benefits: Temporal advantages for security processes, attack prevention through computational deceleration, adaptive security responses based on threat levels, and temporal isolation preventing timing-based attacks.

Performance Benefits: Accelerated security operation performance, reduced system impact from security processes, optimized resource allocation for critical security functions, and improved overall system security responsiveness.

Operational Benefits: Automated temporal domain management, dynamic process classification and assignment, real-time security performance optimization, and integration with existing security frameworks.

Market Opportunity

Cybersecurity Performance Optimization Market: $4.2 billion (2024) with focus on security operation acceleration

Endpoint Protection Market: $18.6 billion with growing emphasis on performance and responsiveness

Security Operations Center (SOC) Tools Market: $8.9 billion with demand for performance optimization solutions

Competitive Position: First system providing computational time dilation for cybersecurity applications, patent-protected temporal domain security technology, unique CPU scheduling-based security acceleration, and novel approach to temporal security isolation.

CONCLUSION

The Computational Time Dilation Security System represents a revolutionary approach to cybersecurity that leverages computational resource management for temporal security advantages. By creating different temporal processing domains through CPU priority scheduling, the system provides security processes with computational acceleration while constraining potential attackers through temporal deceleration.

Key Technical Innovations:

1. Temporal domain creation and management through CPU priority scheduling manipulation

2. Process classification engine for security-based temporal domain assignment

3. Security process acceleration through hyperspace temporal domain optimization

4. Attack process deceleration through quarantine temporal domain restrictions

5. Adaptive temporal security responses based on real-time threat assessment

The system provides unique temporal security advantages that complement traditional cybersecurity approaches, offering enhanced protection through computational time dilation and temporal isolation mechanisms.

END OF PROVISIONAL PATENT APPLICATION

Filing Status: Ready for USPTO submission

Priority Date: [To be established upon filing]

Related Applications: Integrates with Quantum-Safe Physical Impossibility Architecture and other MWRASP system patents

International Filing: PCT application planned within 12 months