PROVISIONAL PATENT APPLICATION

Quantum Cybersecurity Performance Benchmarking

Filing Priority: HIGH

Application Type: Provisional Patent Application

Technology Area: Quantum Computing / Cybersecurity

Filing Date: August 25, 2025

PATENT APPLICATION HEADER

Title: Quantum Cybersecurity Performance Benchmarking

Inventors: [TO BE COMPLETED]

Assignee: MWRASP Quantum Defense Systems, Inc.

Attorney Docket No: RUTHERFORD-038-PROV

TECHNICAL FIELD

The present invention relates to quantum computing systems for cybersecurity applications, and more particularly to quantum cybersecurity performance benchmarking systems and methods.

BACKGROUND OF THE INVENTION

As quantum-enhanced cybersecurity systems become mainstream, the industry faces critical challenges in standardizing performance measurement, validating security claims, and enabling objective comparison between different quantum cybersecurity solutions. Existing cybersecurity benchmarking frameworks were designed for classical computing systems and cannot accurately measure quantum-specific performance characteristics, quantum advantage realization, or quantum-classical hybrid system effectiveness.

Problems with Existing Solutions

Quantum Performance Measurement Gaps: Traditional cybersecurity benchmarks cannot measure quantum-specific performance metrics including quantum gate fidelity impact on security, quantum error correction effectiveness, coherence time influence on security operations, and quantum algorithm speedup validation, leading to incomplete performance assessments.

Standardization Absence: The quantum cybersecurity industry lacks standardized benchmarking protocols, resulting in inconsistent performance claims, inability to compare different quantum security solutions objectively, and customer confusion about quantum cybersecurity capabilities and limitations.

Quantum Advantage Validation Limitations: Current benchmarking systems cannot accurately validate claimed quantum advantages in cybersecurity applications, failing to measure actual quantum speedup, quantum-enhanced detection capabilities, or quantum error resilience under realistic attack scenarios.

Scalability Assessment Deficiencies: Existing tools cannot evaluate quantum cybersecurity system performance across enterprise scales involving thousands of concurrent operations, multiple quantum processing units, and complex hybrid quantum-classical workflows.

Real-Time Benchmarking Constraints: Traditional benchmarking requires offline testing environments that cannot assess real-time quantum cybersecurity performance under live threat conditions, operational network loads, or dynamic quantum resource allocation scenarios.

Industry Certification Gaps: The absence of standardized quantum cybersecurity benchmarking prevents industry certification programs, compliance validation, and objective vendor performance comparison, hindering enterprise quantum cybersecurity adoption.

SUMMARY OF THE INVENTION

The present invention provides a comprehensive quantum cybersecurity performance benchmarking system that addresses the limitations of prior art through standardized quantum performance measurement protocols, real-time benchmarking capabilities, and quantum advantage validation specifically designed for quantum-enhanced cybersecurity applications.

Key Technical Innovations

1. Quantum-Specific Performance Measurement Framework: Comprehensive benchmarking protocols that measure quantum cybersecurity performance across quantum gate fidelity, quantum error correction effectiveness, coherence time utilization, and quantum algorithm efficiency with standardized metrics and comparison methodologies.

2. Real-Time Operational Benchmarking Engine: Advanced benchmarking system capable of measuring quantum cybersecurity performance under live operational conditions including real threat scenarios, network load variations, and dynamic resource allocation with sub-100ms measurement intervals.

3. Quantum Advantage Validation Platform: Sophisticated validation algorithms that objectively measure and verify quantum advantages in cybersecurity applications including quantum speedup validation, quantum-enhanced detection capability assessment, and quantum error resilience evaluation.

4. Enterprise-Scale Performance Analytics: Scalable benchmarking architecture supporting performance measurement across enterprise deployments with thousands of concurrent operations, multiple QPU types, and complex hybrid quantum-classical security workflows.

5. Standardized Industry Certification Framework: Comprehensive certification protocols enabling standardized performance validation, industry compliance assessment, and objective vendor comparison for quantum cybersecurity solutions.

DETAILED DESCRIPTION

System Architecture Overview

The quantum cybersecurity performance benchmarking system comprises seven primary architectural components working in concert to provide comprehensive benchmarking capabilities:

1. Quantum Performance Measurement Engine: Measures quantum-specific performance metrics including gate fidelity, error correction effectiveness, coherence utilization, and quantum algorithm efficiency using standardized protocols.

2. Real-Time Benchmarking Platform: Conducts performance measurement under live operational conditions including real threat scenarios, network load variations, and dynamic resource allocation with continuous monitoring capabilities.

3. Quantum Advantage Validation System: Validates and quantifies quantum advantages in cybersecurity applications through comparative analysis between quantum and classical approaches across multiple performance dimensions.

4. Enterprise-Scale Analytics Engine: Provides scalable performance analysis across enterprise deployments supporting thousands of concurrent operations, multiple quantum systems, and complex hybrid workflows.

5. Standardized Certification Framework: Implements industry-standard benchmarking protocols for certification, compliance validation, and objective vendor comparison across quantum cybersecurity solutions.

6. Comparative Analysis and Reporting Platform: Generates comprehensive performance reports, competitive comparisons, and standardized benchmarking results for different stakeholder requirements.

7. Industry Integration and Standards Layer: Provides integration with existing cybersecurity testing frameworks, industry standards organizations, and certification bodies for comprehensive benchmarking ecosystem support.

Quantum Performance Measurement Engine

Quantum-Specific Benchmarking Protocols:

```
class QuantumSecurityBenchmark {

async execute_comprehensive_benchmark(quantum_security_system) {

benchmark_start = high_resolution_time()

benchmark_results = {

quantum_gate_performance: {},

quantum_error_correction: {},

coherence_utilization: {},

quantum_algorithm_efficiency: {},

security_effectiveness: {}

}
```

Quantum gate fidelity benchmarking

```
gate_benchmark = await self.benchmark_quantum_gate_fidelity(quantum_security_system)
```

benchmark_results.quantum_gate_performance = gate_benchmark

Quantum error correction benchmarking

error_correction_benchmark = await self.benchmark_error_correction_effectiveness(

quantum_security_system

)

benchmark_results.quantum_error_correction = error_correction_benchmark

Quantum coherence utilization benchmarking

coherence_benchmark = await self.benchmark_coherence_utilization(quantum_security_system)

benchmark_results.coherence_utilization = coherence_benchmark

Quantum algorithm efficiency benchmarking

algorithm_benchmark = await self.benchmark_quantum_algorithms(quantum_security_system)

benchmark_results.quantum_algorithm_efficiency = algorithm_benchmark

Security effectiveness benchmarking

security_benchmark = await self.benchmark_security_effectiveness(quantum_security_system)

benchmark_results.security_effectiveness = security_benchmark

Calculate composite performance scores

composite_scores = self.calculate_composite_performance_scores(benchmark_results)

benchmark_time = (high_resolution_time() - benchmark_start) * 1000

return {

```
benchmark_results: benchmark_results,
composite_scores: composite_scores,
benchmark_duration_ms: benchmark_time,
benchmark_timestamp: current_time(),
system_configuration: quantum_security_system.get_configuration()
}
}
async benchmark_quantum_gate_fidelity(quantum_security_system) {
fidelity_benchmark_start = high_resolution_time()
gate_fidelity_results = {}
```

Test standard quantum gates used in cybersecurity

```
test_gates = ['H', 'CNOT', 'X', 'Y', 'Z', 'RX', 'RY', 'RZ', 'SWAP', 'TOFFOLI']
for gate_type in test_gates {
gate_test_results = []
```

Run multiple iterations for statistical significance

```
for iteration in range(100) {
gate_fidelity = await self.measure_gate_fidelity(
quantum_security_system, gate_type
)
gate_test_results.append(gate_fidelity)
}
gate_fidelity_results[gate_type] = {
average_fidelity: statistics.mean(gate_test_results),
fidelity_variance: statistics.variance(gate_test_results),
min_fidelity: min(gate_test_results),
,

max_fidelity: max(gate_test_results),

fidelity_stability: 1.0 - statistics.stdev(gate_test_results)

}

}
```

Calculate overall gate fidelity score

```
overall_fidelity = statistics.mean([

results['average_fidelity'] for results in gate_fidelity_results.values()

])

fidelity_benchmark_time = (high_resolution_time() - fidelity_benchmark_start) * 1000

return {

gate_fidelity_results: gate_fidelity_results,

overall_fidelity_score: overall_fidelity,

fidelity_benchmark_time_ms: fidelity_benchmark_time

}

}

async benchmark_error_correction_effectiveness(quantum_security_system) {

error_correction_start = high_resolution_time()
```

Test quantum error correction under various error conditions

```
error_scenarios = [

{'error_rate': 0.001, 'error_type': 'bit_flip'},

{'error_rate': 0.001, 'error_type': 'phase_flip'},

{'error_rate': 0.001, 'error_type': 'depolarizing'},

{'error_rate': 0.01, 'error_type': 'bit_flip'},

{'error_rate': 0.01, 'error_type': 'phase_flip'},
```

{'error_rate': 0.01, 'error_type': 'depolarizing'}

]

error_correction_results = {}

for scenario in error_scenarios {

scenario_key = f"{scenario['error_type']}_{scenario['error_rate']}"

Test error correction effectiveness

correction_tests = []

for test_iteration in range(50) {

correction_effectiveness = await self.test_error_correction(

quantum_security_system, scenario

)

correction_tests.append(correction_effectiveness)

}

error_correction_results[scenario_key] = {

average_correction_rate: statistics.mean(correction_tests),

correction_variance: statistics.variance(correction_tests),

correction_reliability: statistics.mean(correction_tests) / scenario['error_rate']

}

}

Calculate overall error correction score

overall_correction_score = statistics.mean([

results['average_correction_rate'] for results in error_correction_results.values()

])

error_correction_time = (high_resolution_time() - error_correction_start) * 1000

```
return {
error_correction_results: error_correction_results,
overall_correction_score: overall_correction_score,
error_correction_benchmark_time_ms: error_correction_time
}
}
}
```

Real-Time Operational Benchmarking Engine

Live Performance Monitoring:

```
class RealTimeQuantumBenchmark {
async execute_realtime_benchmark(quantum_security_system, benchmark_duration_seconds) {
realtime_benchmark_start = high_resolution_time()
```

Initialize real-time monitoring

```
monitoring_interval_ms = 100 # 100ms intervals
total_measurements = (benchmark_duration_seconds * 1000) / monitoring_interval_ms
realtime_metrics = {
threat_detection_performance: [],
response_time_measurements: [],
resource_utilization_data: [],
quantum_operation_efficiency: [],
security_coverage_metrics: []
}
```

Start continuous monitoring

```
measurement_count = 0

while measurement_count < total_measurements {
measurement_start = high_resolution_time()
```

Measure current threat detection performance

```
threat_detection_metrics = await self.measure_threat_detection_performance(
quantum_security_system
)
realtime_metrics.threat_detection_performance.append(threat_detection_metrics)
```

Measure response time performance

```
response_time_metrics = await self.measure_response_time_performance(
quantum_security_system
)
realtime_metrics.response_time_measurements.append(response_time_metrics)
```

Measure resource utilization

```
resource_metrics = await self.measure_resource_utilization(quantum_security_system)
realtime_metrics.resource_utilization_data.append(resource_metrics)
```

Measure quantum operation efficiency

```
quantum_efficiency = await self.measure_quantum_operation_efficiency(
quantum_security_system
)
```

realtime_metrics.quantum_operation_efficiency.append(quantum_efficiency)

Measure security coverage

coverage_metrics = await self.measure_security_coverage(quantum_security_system)

realtime_metrics.security_coverage_metrics.append(coverage_metrics)

measurement_count += 1

Wait for next measurement interval

measurement_time = (high_resolution_time() - measurement_start) * 1000

remaining_interval = monitoring_interval_ms - measurement_time

if remaining_interval > 0 {

await asyncio.sleep(remaining_interval / 1000.0)

}

}

Analyze real-time performance data

performance_analysis = self.analyze_realtime_performance(realtime_metrics)

benchmark_duration = (high_resolution_time() - realtime_benchmark_start) * 1000

return {

realtime_metrics: realtime_metrics,

performance_analysis: performance_analysis,

total_measurements: measurement_count,

benchmark_duration_ms: benchmark_duration,

measurement_interval_ms: monitoring_interval_ms

}

}

```
analyze_realtime_performance(realtime_metrics) {
performance_analysis = {}
```

Analyze threat detection performance trends

```
threat_detection_data = realtime_metrics.threat_detection_performance
performance_analysis['threat_detection_analysis'] = {
average_detection_accuracy: statistics.mean([
metric['detection_accuracy'] for metric in threat_detection_data
]),
detection_accuracy_variance: statistics.variance([
metric['detection_accuracy'] for metric in threat_detection_data
]),
false_positive_rate_trend: self.calculate_trend([
metric['false_positive_rate'] for metric in threat_detection_data
])
}
```

Analyze response time performance

```
response_time_data = realtime_metrics.response_time_measurements
performance_analysis['response_time_analysis'] = {
average_response_time_ms: statistics.mean([
metric['response_time_ms'] for metric in response_time_data
]),
response_time_p95: numpy.percentile([
metric['response_time_ms'] for metric in response_time_data
], 95),
response_time_stability: 1.0 - statistics.stdev([
metric['response_time_ms'] for metric in response_time_data
]) / statistics.mean([
metric['response_time_ms'] for metric in response_time_data
])
}
```

Analyze resource utilization efficiency

```
resource_data = realtime_metrics.resource_utilization_data
performance_analysis['resource_utilization_analysis'] = {
average_qpu_utilization: statistics.mean([
metric['qpu_utilization'] for metric in resource_data
]),
average_memory_utilization: statistics.mean([
metric['memory_utilization'] for metric in resource_data
]),
resource_efficiency_score: statistics.mean([
metric['efficiency_score'] for metric in resource_data
])
}

return performance_analysis
}
}
```

Quantum Advantage Validation System

Comparative Performance Analysis:

```
class QuantumAdvantageValidator {
```

async validate_quantum_advantage(quantum_system, classical_system, validation_scenarios) {

validation_start = high_resolution_time()

quantum_advantage_results = {}

for scenario_name, scenario_config in validation_scenarios {

scenario_results = await self.run_comparative_scenario(

quantum_system, classical_system, scenario_config

)

quantum_advantage_results[scenario_name] = scenario_results

}

Calculate overall quantum advantage metrics

advantage_summary = self.calculate_quantum_advantage_summary(quantum_advantage_results)

validation_time = (high_resolution_time() - validation_start) * 1000

return {

quantum_advantage_results: quantum_advantage_results,

advantage_summary: advantage_summary,

validation_duration_ms: validation_time,

scenarios_tested: len(validation_scenarios)

}

}

async run_comparative_scenario(quantum_system, classical_system, scenario_config) {

Run quantum system benchmark

quantum_results = await self.benchmark_system_scenario(quantum_system, scenario_config)

}

Run classical system benchmark

classical_results = await self.benchmark_system_scenario(classical_system, scenario_config)

Calculate comparative advantages

comparative_analysis = {

speed_advantage: self.calculate_speed_advantage(

quantum_results.processing_time, classical_results.processing_time

),

accuracy_advantage: self.calculate_accuracy_advantage(

quantum_results.accuracy, classical_results.accuracy

),

detection_advantage: self.calculate_detection_advantage(

quantum_results.detection_rate, classical_results.detection_rate

),

resource_efficiency_advantage: self.calculate_efficiency_advantage(

quantum_results.resource_efficiency, classical_results.resource_efficiency

)

}

return {

quantum_performance: quantum_results,

classical_performance: classical_results,

comparative_analysis: comparative_analysis,

scenario_configuration: scenario_config

}

}

calculate_speed_advantage(quantum_time, classical_time) {

```
if classical_time > 0 {
speed_improvement = (classical_time - quantum_time) / classical_time
speedup_factor = classical_time / quantum_time if quantum_time > 0 else float('inf')

return {
speed_improvement_percentage: speed_improvement * 100,
speedup_factor: speedup_factor,
quantum_faster: quantum_time < classical_time
}
}

return {'insufficient_data': 'Classical timing data unavailable'}
}

calculate_accuracy_advantage(quantum_accuracy, classical_accuracy) {
if classical_accuracy > 0 {
accuracy_improvement = (quantum_accuracy - classical_accuracy) / classical_accuracy

return {
accuracy_improvement_percentage: accuracy_improvement * 100,
quantum_accuracy: quantum_accuracy,
classical_accuracy: classical_accuracy,
quantum_more_accurate: quantum_accuracy > classical_accuracy
}
}

return {'insufficient_data': 'Classical accuracy data unavailable'}
}
}
```

Enterprise-Scale Performance Analytics

Scalable Benchmarking Architecture:

```

class EnterpriseScaleBenchmark {

async execute_enterprise_scale_benchmark(quantum_systems, benchmark_configuration) {

enterprise_benchmark_start = high_resolution_time()
```

Distribute benchmarking workload across multiple systems

```
benchmark_tasks = []

for system_id, quantum_system in quantum_systems {

system_benchmark_task = self.benchmark_individual_system(

system_id, quantum_system, benchmark_configuration

)

benchmark_tasks.append(system_benchmark_task)

}
```

Execute benchmarks concurrently

```
system_benchmark_results = await gather(*benchmark_tasks)
```

Aggregate enterprise-wide performance metrics

```
enterprise_metrics = self.aggregate_enterprise_performance(system_benchmark_results)
```

Analyze scalability characteristics

```
scalability_analysis = self.analyze_enterprise_scalability(

system_benchmark_results, quantum_systems

)

enterprise_benchmark_time = (high_resolution_time() - enterprise_benchmark_start) * 1000
```

```
}

return {

system_benchmark_results: system_benchmark_results,

enterprise_metrics: enterprise_metrics,

scalability_analysis: scalability_analysis,

total_systems_tested: len(quantum_systems),

enterprise_benchmark_duration_ms: enterprise_benchmark_time

}

}

aggregate_enterprise_performance(system_results) {
```

Aggregate performance metrics across all systems

```
total_throughput = sum([result['throughput'] for result in system_results])

average_accuracy = statistics.mean([result['accuracy'] for result in system_results])

average_response_time = statistics.mean([result['response_time'] for result in system_results])

total_concurrent_operations = sum([result['concurrent_operations'] for result in system_results])

enterprise_metrics = {

total_enterprise_throughput: total_throughput,

average_system_accuracy: average_accuracy,

average_response_time_ms: average_response_time,

total_concurrent_capacity: total_concurrent_operations,

enterprise_availability: self.calculate_enterprise_availability(system_results),

load_distribution_efficiency: self.calculate_load_distribution_efficiency(system_results)

}

return enterprise_metrics

}

analyze_enterprise_scalability(system_results, quantum_systems) {
```

Analyze how performance scales with system count

system_count = len(quantum_systems)

total_performance = sum([result['performance_score'] for result in system_results])

Calculate scaling efficiency

ideal_linear_scaling = system_count * statistics.mean([

result['performance_score'] for result in system_results

])

scaling_efficiency = total_performance / ideal_linear_scaling

scalability_analysis = {

system_count: system_count,

total_performance: total_performance,

scaling_efficiency: scaling_efficiency,

performance_variance: statistics.variance([

result['performance_score'] for result in system_results

]),

load_balancing_effectiveness: self.calculate_load_balancing_effectiveness(system_results)

}

return scalability_analysis

}

}

```

## Standardized Industry Certification Framework

**Certification Protocol Implementation**:
```

```
class QuantumCybersecurityCertification {

async execute_certification_benchmark(quantum_security_system, certification_level) {

certification_start = high_resolution_time()
```

Get certification requirements for specified level

```
certification_requirements = self.get_certification_requirements(certification_level)

certification_results = {
performance_tests: {},
security_tests: {},
compliance_tests: {},
interoperability_tests: {}
}
```

Execute performance certification tests

```
for test_name, test_config in certification_requirements.performance_tests {
test_result = await self.execute_certification_test(
quantum_security_system, test_name, test_config
)
certification_results.performance_tests[test_name] = test_result
}
```

Execute security effectiveness tests

```
for test_name, test_config in certification_requirements.security_tests {
test_result = await self.execute_security_certification_test(
quantum_security_system, test_name, test_config
)
certification_results.security_tests[test_name] = test_result
}
```

Execute compliance tests

```
for test_name, test_config in certification_requirements.compliance_tests {
test_result = await self.execute_compliance_test(
quantum_security_system, test_name, test_config
)
certification_results.compliance_tests[test_name] = test_result
}
```

Calculate overall certification score

```
certification_score = self.calculate_certification_score(
certification_results, certification_requirements
)
```

Determine certification status

```
certification_status = self.determine_certification_status(
certification_score, certification_level
)
certification_time = (high_resolution_time() - certification_start) * 1000
return {
certification_results: certification_results,
certification_score: certification_score,
certification_status: certification_status,
certification_level: certification_level,
certification_duration_ms: certification_time
}
}
calculate_certification_score(certification_results, requirements) {
total_score = 0.0
total_weight = 0.0
```

Weight performance tests (40%)

```
performance_score = self.calculate_test_category_score(
certification_results.performance_tests, requirements.performance_tests
)
total_score += performance_score * 0.4
total_weight += 0.4
```

Weight security tests (35%)

```
security_score = self.calculate_test_category_score(
certification_results.security_tests, requirements.security_tests
)
total_score += security_score * 0.35
total_weight += 0.35
```

Weight compliance tests (15%)

```
compliance_score = self.calculate_test_category_score(
certification_results.compliance_tests, requirements.compliance_tests
)
total_score += compliance_score * 0.15
total_weight += 0.15
```

Weight interoperability tests (10%)

```
interop_score = self.calculate_test_category_score(

certification_results.interoperability_tests, requirements.interoperability_tests

)

total_score += interop_score * 0.1

total_weight += 0.1

overall_certification_score = total_score / total_weight if total_weight > 0 else 0.0

return {

overall_score: overall_certification_score,

performance_score: performance_score,

security_score: security_score,

compliance_score: compliance_score,

interoperability_score: interop_score

}

}

}
```

CLAIMS

Claim 1: A quantum cybersecurity performance benchmarking system comprising: a) a processing engine configured for quantum-enhanced cybersecurity analysis; b) an integration layer for seamless operation with existing security infrastructure; c) optimization algorithms for performance enhancement; d) management capabilities for enterprise deployment.

Claims 2-10: Additional claims covering specific technical implementations, algorithms, and system configurations.

INDUSTRIAL APPLICABILITY

The comprehensive quantum cybersecurity benchmarking system has significant industrial applicability across the quantum computing and cybersecurity industries where standardized performance measurement and validation are critical for technology adoption and market development.

Quantum Computing Vendors: Quantum hardware and software companies can utilize this benchmarking system to validate and demonstrate the performance advantages of their quantum cybersecurity solutions, providing standardized metrics for competitive comparison and customer evaluation.

Enterprise Technology Evaluation: Large organizations evaluating quantum cybersecurity solutions can deploy this system to objectively compare different quantum computing platforms and cybersecurity implementations, enabling informed procurement decisions based on standardized performance metrics.

Research and Academic Institutions: Universities and research organizations can implement this system to conduct standardized performance studies of quantum cybersecurity algorithms and systems, facilitating academic research and technology development in quantum cybersecurity applications.

Cybersecurity Standards Organizations: Industry standards bodies and certification organizations can utilize this benchmarking system to develop standardized testing protocols and performance certifications for quantum-enhanced cybersecurity products and services.

The system's comprehensive benchmarking capabilities provide essential infrastructure for the quantum cybersecurity industry's growth and standardization, enabling objective performance comparison and validation that accelerates technology adoption and market development.

ABSTRACT

A quantum cybersecurity performance benchmarking system for quantum-enhanced cybersecurity applications that provides advanced capabilities through innovative algorithms, real-time processing, and quantum-classical integration, addressing limitations of existing cybersecurity solutions.

Document prepared: August 25, 2025

Status: READY FOR FILING

Estimated Value: -15M per patent