Quantum-Resistant Graph Security Analyzer

Purpose: MAANG-level Python script demonstrating:

- Quantum-resistant cryptographic hashing (SHA3-512)
- Graph theory and network analysis
- AI-assisted anomaly detection in graph structures
- Blockchain-inspired verification of graph integrity
- Parallel and asynchronous processing
- Serialization, logging, and modular architecture

Keywords: quantum-resistant, blockchain, graph-theory, anomaly-detection, parallel-processing, async-python, hashing, MAANG-ready

"""

```
# -----------------------------
# Imports
# -----------------------------
import hashlib
import networkx as nx
import asyncio
import concurrent.futures
import pickle
import random
import logging
import time
import numpy as np
from sklearn.ensemble import IsolationForest  # AI anomaly detection
```

```
# ------------------------------
# Logging setup
# ------------------------------
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s | %(levelname)s | %(message)s',
    handlers=[logging.FileHandler("quantum_blockchain_graph.log"), logging.StreamHandler()]
)

# ------------------------------
# Quantum-resistant hashing
# ------------------------------
def sha3_512_hash(data: str) -> str:
    """Generate a SHA3-512 quantum-resistant hash."""
    return hashlib.sha3_512(data.encode('utf-8')).hexdigest()

# ------------------------------
# Random Graph Generation
# ------------------------------
def generate_random_graph(nodes: int, edge_probability: float) -> nx.Graph:
    """Generate a random undirected graph for analysis."""
    logging.info(f"Generating graph with {nodes} nodes...")
    return nx.erdos_renyi_graph(n=nodes, p=edge_probability)
```

```
# ------------------------------
# Graph Hashing Pipeline
# ------------------------------
def graph_hashing_pipeline(graph: nx.Graph) -> dict:
    """Hash nodes and edges for blockchain verification."""
    node_hashes = {f"node_{n}": sha3_512_hash(str(n)) for n in graph.nodes()}
    edge_hashes = {f"edge_{u}_{v}": sha3_512_hash(f"{u}-{v}") for u, v in graph.edges()}
    return {**node_hashes, **edge_hashes}

# ------------------------------
# Parallel Centrality Analysis
# ------------------------------
def node_centrality(graph: nx.Graph) -> dict:
    """Compute eigenvector centrality in parallel."""
    with concurrent.futures.ThreadPoolExecutor() as executor:
        future = executor.submit(nx.eigenvector_centrality_numpy, graph)
        return future.result()

# ------------------------------
# Async Node Hashing
# ------------------------------
async def async_hash_nodes(graph: nx.Graph) -> dict:
    """Asynchronously hash nodes to optimize performance."""
    tasks = [asyncio.to_thread(sha3_512_hash, str(n)) for n in graph.nodes()]
    hashed_nodes = await asyncio.gather(*tasks)
```

```
    return {n: h for n, h in zip(graph.nodes(), hashed_nodes)}


# ------------------------------
# AI-Assisted Anomaly Detection
# ------------------------------
def detect_anomalies(graph: nx.Graph) -> list:
    """Detect anomalous nodes using IsolationForest (AI/ML)."""
    logging.info("Detecting anomalies with AI...")
    degrees = np.array([graph.degree(n) for n in graph.nodes()]).reshape(-1, 1)
    model = IsolationForest(contamination=0.05, random_state=42)
    preds = model.fit_predict(degrees)
    anomalies = [n for n, p in zip(graph.nodes(), preds) if p == -1]
    logging.info(f"Detected {len(anomalies)} anomalous nodes")
    return anomalies


# ------------------------------
# Blockchain-inspired verification
# ------------------------------
def blockchain_verification(graph: nx.Graph, hashes: dict) -> str:
    """Simulate blockchain verification of the graph."""
    logging.info("Verifying graph integrity (blockchain simulation)...")
    concatenated_hash = ''.join(sorted(hashes.values()))
    return sha3_512_hash(concatenated_hash)


# ------------------------------
```

```python
# Serialization
# ------------------------------
def save_graph_data(graph: nx.Graph, filename: str = "graph_blockchain.pkl"):
    """Serialize graph, hashes, anomalies, and verification hash."""
    data = {
        "graph": graph,
        "hashes": graph_hashing_pipeline(graph),
        "centrality": node_centrality(graph),
        "anomalies": detect_anomalies(graph)
    }
    data["verification_hash"] = blockchain_verification(graph, data["hashes"])
    with open(filename, "wb") as f:
        pickle.dump(data, f)
    logging.info(f"Serialized all data to {filename}")


# ------------------------------
# Main Execution Pipeline
# ------------------------------
async def main():
    """End-to-end MAANG-ready pipeline"""
    start_time = time.time()

    # 1. Generate graph
    G = generate_random_graph(nodes=1000, edge_probability=0.02)
```

```
    # 2. Parallel centrality
    centrality = node_centrality(G)

    # 3. Async node hashing
    node_hashes = await async_hash_nodes(G)

    # 4. Full graph hash pipeline
    full_hashes = graph_hashing_pipeline(G)

    # 5. AI anomaly detection
    anomalies = detect_anomalies(G)

    # 6. Blockchain verification
    verification_hash = blockchain_verification(G, full_hashes)

    # 7. Save all data
    save_graph_data(G)

    end_time = time.time()
    logging.info(f"Pipeline completed in {end_time - start_time:.2f} seconds")
    logging.info(f"Blockchain verification hash: {verification_hash}")
    logging.info(f"Anomalous nodes: {anomalies[:10]} (showing first 10)")

# ------------------------------
# Entry point
```

```
# ------------------------------
if __name__ == "__main__":
    asyncio.run(main())
```

Script Summary:

- MAANG-level showcase: quantum-resistant cryptography + blockchain + graph theory + AI anomaly detection + async + parallel

- Demonstrates expertise in cutting-edge technologies

- Highly modular, well-commented, scalable

- Keywords: quantum-resistant, anomaly-detection, blockchain, graph-theory, parallel-processing, async-python, MAANG-ready