PyAuraSeal514: Cryptographic Integrity with The Hidden Cipher

> ***"When cryptography meets algorithm, we seal them with aura - and I did just that."***

License MIT (https://opensource.org/licenses/MIT) python 3.8+ (https://www.python.org/downloads/) OBINexus Computing (https://github.com/obinexus)

Table of Contents

Overview

PyAuraSeal514 is a cryptographic algorithm implementation that combines:

- **The Hidden Cipher** : One-way functions based on perfect number theory
- **Huffman-AVL Trie** : Optimal compression with structural integrity
- **514 Hex Digest** : Distributed entropy across 6 components
- **2:1 Public-Private Key Architecture** : Vector-based public keys, scalar private key
- **Consciousness-Aware Security** : Prevents tampering through entropy distribution

Core Innovation

AuraSeal514 treats cryptographic integrity as a **perfect number problem** - where the sum of cryptographic components must equal the whole system for validation to succeed.

The Hidden Cipher Foundation

Based on the research paper "The Hidden Cipher - Odd Perfect Numbers and Cryptographic Integrity" by Nnamdi Michael Okpala, this implementation applies perfect number theory to cryptographic systems.

Perfect Number Analogy

```
 # Perfect Number: 6 = 1 + 2 + 3 (sum of proper divisors)
# AuraSeal514: System = Component₁ + Component₂ + ... + Component₆

def verify_perfect_integrity(components):
    """
    Verify that the sum of components equals the expected system hash
    Following the perfect number cryptographic principle
    """
    component_sum = sum(int(comp, 16) for comp in components)
    system_hash = generate_system_signature(components)

    return component_sum == int(system_hash[:8], 16)  # First 32 bits compar
```

Cryptographic Key Pair Harmony

Just as perfect numbers maintain harmony between their divisors:

- **GCD(6, d) = d** for all proper divisors d
- **LCM(6, d) = 6** for all proper divisors d

AuraSeal514 maintains harmony between public and private keys:

- **Verify(pub_1, message) ⊕ Verify(pub_2, message) = private_key_validation**
- **Entropy(pub_1) + Entropy(pub_2) = 2 × Entropy(private_key)**

Technical Architecture

PhenoAVL Data Structure

```
class PhenoAVLNode:
    """
    Phenomenological AVL Node with Huffman integration
    Maintains balance while preserving cryptographic properties
    """
    def __init__(self, key: str, huffman_code: str = "", freq: int = 0):
        self.key = key
        self.huffman_code = huffman_code           # Huffman compression
        self.frequency = freq                      # Usage frequency
        self.height = 1                           # AVL tree height
        self.balance_factor = 0                   # AVL balance [-1, 0, 1]
        self.checksum = self._calculate_checksum() # Integrity hash
```

Component Architecture

The system is built on **six fundamental components** :

1. **Data Model Encoder** - Huffman compression
2. **Algorithm Encoder** - Context-aware encoding
3. **Validation Engine** - Isomorphic verification
4. **Binary Processor** - Checksum validation
5. **Recovery System** - Self-healing capabilities
6. **Coordinate Mapper** - Spatial distribution

AuraSeal514 Algorithm

514 Hex Decimal Digest Specification

The **514** comes from the mathematical relationship:

- **514 ÷ 4 = 128.5**
- **128** hex digits per component × **4** primary components = **512**
- **+2** additional digits for entropy validation = **514**

```
# AuraSeal514 Structure
AURASEAL_PATTERN = [
    "128 hex digits",    # Component 1: [A-F0-9]
    "128 hex digits",    # Component 2: [A-F0-9]
    "128 hex digits",    # Component 3: [A-F0-9]
    "128 hex digits",    # Component 4: [A-F0-9]
    "128 hex digits",    # Component 5: [A-F0-9]
    "126 hex digits",    # Component 6: [A-F0-9] (514 - 512 = 2, so 128-2=12
]

# Total: 128×5 + 126 = 640 + 126 = 766...
# Wait, let me recalculate: 514 ÷ 6 components ≈ 85.67 per component
```

Corrected 514 Distribution

```
# Proper 514 Hex Distribution across 6 components
COMPONENT_SIZES = [
    86,  # Component 1: 86 hex digits
    86,  # Component 2: 86 hex digits
    86,  # Component 3: 86 hex digits
    86,  # Component 4: 86 hex digits
    85,  # Component 5: 85 hex digits
    85,  # Component 6: 85 hex digits
]
# Total: 86×4 + 85×2 = 344 + 170 = 514 ✓
```

Component Generation

```
def generate_auraseal514_components(data: str) -> List[str]:
    """
    Generate 514 hex digits distributed across 6 components
    Each component represents a cryptographic aspect of the data
    """
    # Generate base hash
    base_hash = hashlib.sha512(data.encode()).hexdigest()  # 128 chars

    components = []
    offset = 0

    for i, size in enumerate(COMPONENT_SIZES):
        # Generate component-specific hash
        component_data = f"{data}:component_{i}:{base_hash[offset:offset+32]
        component_hash = hashlib.sha256(component_data.encode()).hexdigest()

        # Extend to required length using recursive hashing
        while len(component_hash) < size:
            component_hash += hashlib.md5(component_hash.encode()).hexdigest

        components.append(component_hash[:size].upper())
        offset = (offset + 32) % len(base_hash)

    return components
```

Hex Digest Specification

GUID/UUID Integration

```python
import uuid

def generate_sealed_guid(data: str) -> str:
    """
    Generate GUID with AuraSeal514 integration
    Format: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX (sealed)
    """
    # Generate base GUID
    guid = str(uuid.uuid4())

    # Generate AuraSeal components
    components = generate_auraseal514_components(f"{data}:{guid}")

    # Create sealed GUID using first component
    sealed_guid = f"{components[0][:8]}-{components[0][8:12]}-{components[0]

    return sealed_guid

def verify_sealed_guid(guid: str, original_data: str) -> bool:
    """
    Verify that GUID was generated from original data
    """
    # Reconstruct expected GUID
    expected_guid = generate_sealed_guid(original_data)
    return guid == expected_guid
```

Public Key 2:1 Private Key Relationship

```python
def generate_auraseal_keys() -> Tuple[Dict[int, str], str]:
    """
    Generate dual public keys (2:1 ratio) and single private key

    Architecture:
    - Public Key 1: Vector derived from private key + entropy₁
    - Public Key 2: Vector derived from private key + entropy₂
    - Private Key: Scalar generated first (O(log n) complexity)

    Returns:
        Tuple of (public_keys_dict, private_key_string)
    """
    # Step 1: Generate private key (scalar) - O(log n)
    private_entropy = os.urandom(64)
    private_key = hashlib.sha512(private_entropy).hexdigest()

    # Step 2: Generate public keys (vectors) - O(n)
    pub_key_1_data = f"{private_key}:vector1:{time.time()}"
    pub_key_2_data = f"{private_key}:vector2:{time.time()}"

    pub_key_1_components = generate_auraseal514_components(pub_key_1_data)
    pub_key_2_components = generate_auraseal514_components(pub_key_2_data)

    # Concatenate components for full public keys
    public_key_1 = "".join(pub_key_1_components)  # 514 hex chars
    public_key_2 = "".join(pub_key_2_components)  # 514 hex chars

    return {1: public_key_1, 2: public_key_2}, private_key

def sign_with_auraseal(data: str, private_key: str) -> str:
    """
    Sign data using AuraSeal514 algorithm
    Creates signature that can be verified with either public key
    """
    # Generate signature components
    signature_data = f"{data}:{private_key}:auraseal514"
    signature_components = generate_auraseal514_components(signature_data)
```

```
    # Create composite signature
    signature = hashlib.sha256(":".join(signature_components).encode()).hexd

    return signature

def verify_auraseal_signature(data: str, signature: str, public_key: str) ->
    """
    Verify signature using public key
    Note: In practice, this would use proper asymmetric cryptography
    """
    # Extract components from public key
    pub_components = [
        public_key[i*86:(i+1)*86] if i < 4 else public_key[i*85+4:(i+1)*85+4
        for i in range(6)
    ]

    # Generate expected signature pattern
    verification_data = f"{data}:{':'.join(pub_components)}:verify"
    expected_hash = hashlib.sha256(verification_data.encode()).hexdigest()

    # Check signature validity (simplified for demonstration)
    return len(signature) == len(expected_hash) and signature[:16] == expect
```

Component Analysis

6-Component Architecture

Each of the 6 components serves a specific cryptographic purpose:

```python
COMPONENT_PURPOSES = {
    0: "Data Integrity Hash",     # Validates data hasn't been tampered
    1: "Structural Checksum",     # Verifies file/archive structure
    2: "Temporal Signature",      # Time-based validation
    3: "Entropy Distribution",    # Ensures proper randomness
    4: "Access Control Hash",     # Permission validation
    5: "Recovery Verification"    # Self-healing capability
}

def analyze_components(components: List[str]) -> Dict[str, any]:
    """
    Analyze the cryptographic strength of each component
    """
    analysis = {}

    for i, component in enumerate(components):
        analysis[f"component_{i}"] = {
            "purpose": COMPONENT_PURPOSES[i],
            "length": len(component),
            "entropy": calculate_entropy(component),
            "hex_distribution": analyze_hex_distribution(component),
            "strength_score": calculate_strength_score(component)
        }

    return analysis

def calculate_entropy(hex_string: str) -> float:
    """
    Calculate Shannon entropy of hex string
    Perfect entropy = 4.0 for hex (16 possible values)
    """
    char_counts = {}
    for char in hex_string:
        char_counts[char] = char_counts.get(char, 0) + 1

    entropy = 0
    length = len(hex_string)
```

```
    for count in char_counts.values():
        probability = count / length
        entropy -= probability * math.log2(probability)

    return entropy

def analyze_hex_distribution(hex_string: str) -> Dict[str, int]:
    """
    Analyze distribution of hex characters [0-9A-F]
    Good distribution is crucial for cryptographic strength
    """
    distribution = {char: 0 for char in "0123456789ABCDEF"}

    for char in hex_string.upper():
        if char in distribution:
            distribution[char] += 1

    return distribution
```

Network Distribution

Index.html Integration

For secure distribution over networks (banking apps, mobile devices):

```
<!DOCTYPE html>
<html>
<head>
    <title>AuraSeal514 Secure Download</title>
    <meta charset="UTF-8">
</head>
<body>
    <script>
        // AuraSeal514 Network Verification
        class AuraSealNetworkVerifier {
            constructor() {
                this.expectedComponents = 6;
                this.totalHexLength = 514;
            }

            async verifyDownload(downloadUrl, expectedSignature) {
                try {
                    // Download file
                    const response = await fetch(downloadUrl);
                    const data = await response.arrayBuffer();

                    // Generate AuraSeal components from downloaded data
                    const components = this.generateComponents(data);

                    // Verify signature
                    const isValid = this.verifySignature(components, expecte

                    if (!isValid) {
                        throw new Error("AuraSeal verification failed - file
                    }

                    return {
                        success: true,
                        components: components,
                        integrity: "VERIFIED"
                    };
```

```
        } catch (error) {
            // Report to server and block user if necessary
            this.reportTamperAttempt(error);
            throw error;
        }
    }

    generateComponents(data) {
        // JavaScript implementation of component generation
        // (Simplified - in practice would use proper crypto librari
        const hash = this.sha256(data);
        const components = [];

        const sizes = [86, 86, 86, 86, 85, 85];
        let offset = 0;

        for (let i = 0; i < 6; i++) {
            const componentData = hash + i.toString();
            const componentHash = this.sha256(componentData).substri
            components.push(componentHash.toUpperCase());
        }

        return components;
    }

    verifySignature(components, expectedSignature) {
        const combined = components.join("");
        const calculatedSignature = this.sha256(combined);
        return calculatedSignature.substring(0, 16) === expectedSign
    }

    async reportTamperAttempt(error) {
        // Report tampering attempt to server
        await fetch('/api/security/tamper-report', {
            method: 'POST',
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify({
```

```
                        error: error.message,
                        timestamp: Date.now(),
                        userAgent: navigator.userAgent
                    })
                });
            }

            sha256(data) {
                // Simplified SHA-256 (use proper crypto library in producti
                return "simplified_hash_" + data.toString().length;
            }
        }

        // Usage example
        const verifier = new AuraSealNetworkVerifier();

        // Verify ZIP file download
        verifier.verifyDownload('/secure/data.zip', 'expected_signature_here
            .then(result => {
                console.log('Download verified:', result);
                // Proceed with file usage
            })
            .catch(error => {
                console.error('Security violation:', error);
                alert('Security error: File verification failed. Download bl
            });
    </script>
</body>
</html>
```

Mobile Device Security

```
// Mobile-specific AuraSeal verification
class MobileAuraSealVerifier {
    constructor() {
        this.deviceId = this.generateDeviceId();
        this.networkType = this.detectNetworkType();
    }

    generateDeviceId() {
        // Generate unique device identifier
        return btoa(navigator.userAgent + screen.width + screen.height).subs
    }

    detectNetworkType() {
        const connection = navigator.connection || navigator.mozConnection |
        return connection ? connection.effectiveType : 'unknown';
    }

    async secureDownload(url, metadata) {
        // Enhanced security for mobile networks
        const securityHeaders = {
            'X-Device-ID': this.deviceId,
            'X-Network-Type': this.networkType,
            'X-AuraSeal-Version': '514',
            'X-Timestamp': Date.now()
        };

        const response = await fetch(url, {
            headers: securityHeaders
        });

        if (!response.ok) {
            throw new Error(`Download failed: ${response.status}`);
        }

        // Verify using AuraSeal514
        return this.verifyMobileDownload(response, metadata);
    }
```

```
    async verifyMobileDownload(response, metadata) {
        const data = await response.arrayBuffer();

        // Mobile-optimized verification
        const components = this.generateMobileComponents(data, this.deviceId
        const isValid = this.verifyMobileSignature(components, metadata.sign

        if (!isValid) {
            // Enhanced mobile security response
            this.blockDeviceAccess();
            throw new Error("Mobile security violation - access blocked");
        }

        return {
            verified: true,
            deviceSecure: true,
            components: components
        };
    }

    blockDeviceAccess() {
        // Implement device-level security response
        localStorage.setItem('auraseal_security_violation', Date.now());
        // Additional mobile security measures...
    }
}
```

Error Handling & Rollback

State Preservation & Recovery

```python
class AuraSealStateManager:
    """
    Manages system state and implements rollback capabilities
    """

    def __init__(self):
        self.state_history = []
        self.max_states = 50
        self.current_state = None

    def save_state(self, operation_type: str, data: Dict[str, any]):
        """
        Save current state before potentially dangerous operations
        """
        state_snapshot = {
            'timestamp': time.time(),
            'operation': operation_type,
            'data': copy.deepcopy(data),
            'components': self.generate_state_components(data),
            'integrity_hash': self.calculate_state_hash(data)
        }

        self.state_history.append(state_snapshot)

        # Maintain history size
        if len(self.state_history) > self.max_states:
            self.state_history.pop(0)

        self.current_state = state_snapshot
        return state_snapshot

    def verify_state_integrity(self) -> bool:
        """
        Verify current state hasn't been corrupted
        """
        if not self.current_state:
            return False
```

```
        # Recalculate integrity hash
        expected_hash = self.calculate_state_hash(self.current_state['data']
        actual_hash = self.current_state['integrity_hash']

        return expected_hash == actual_hash

    def rollback_to_safe_state(self) -> Dict[str, any]:
        """
        Rollback to the last verified safe state
        """
        safe_states = []

        # Find all states with valid integrity
        for state in reversed(self.state_history):
            test_hash = self.calculate_state_hash(state['data'])
            if test_hash == state['integrity_hash']:
                safe_states.append(state)

        if not safe_states:
            raise Exception("No safe state found for rollback")

        # Use most recent safe state
        safe_state = safe_states[0]
        self.current_state = safe_state

        return {
            'rollback_successful': True,
            'rolled_back_to': safe_state['timestamp'],
            'operation': safe_state['operation']
        }

    def handle_entropy_violation(self, violation_type: str, details: Dict[st
        """
        Handle various types of entropy violations
        """
        violation_handlers = {
```

```
            'component_corruption': self._handle_component_corruption,
            'signature_mismatch': self._handle_signature_mismatch,
            'temporal_anomaly': self._handle_temporal_anomaly,
            'network_tampering': self._handle_network_tampering
        }

        handler = violation_handlers.get(violation_type, self._handle_unknow
        return handler(details)

    def _handle_component_corruption(self, details: Dict[str, any]):
        """Handle component-level corruption"""
        corrupted_component = details.get('component_id')

        # Attempt self-healing
        if self.attempt_component_recovery(corrupted_component):
            return {'status': 'recovered', 'action': 'component_healed'}

        # Fall back to rollback
        rollback_result = self.rollback_to_safe_state()
        return {'status': 'rolled_back', 'action': 'state_restored', 'detail

    def _handle_signature_mismatch(self, details: Dict[str, any]):
        """Handle signature verification failures"""
        # Critical security violation - immediate lockdown
        return {
            'status': 'blocked',
            'action': 'security_lockdown',
            'message': 'Signature mismatch detected - system locked'
        }

    def _handle_temporal_anomaly(self, details: Dict[str, any]):
        """Handle time-based security issues"""
        # Check if this is a replay attack
        timestamp_diff = details.get('timestamp_difference', 0)

        if timestamp_diff > 300:  # 5 minutes
            return {
```

```
            'status': 'blocked',
            'action': 'temporal_lockdown',
            'message': 'Temporal anomaly detected - possible replay atta
        }

    # Minor temporal issue - log and continue
    return {'status': 'warning', 'action': 'logged'}

def _handle_network_tampering(self, details: Dict[str, any]):
    """Handle network-level tampering attempts"""
    tamper_type = details.get('tamper_type', 'unknown')

    if tamper_type in ['mitm', 'injection', 'modification']:
        # Serious network attack - block and report
        return {
            'status': 'blocked',
            'action': 'network_security_violation',
            'message': 'Network tampering detected - connection terminat
        }

    return {'status': 'monitored', 'action': 'enhanced_monitoring'}
```

Production vs Development Behavior

```
class AuraSealEnvironmentManager:
    """
    Handle different behaviors in development vs production
    """

    def __init__(self, environment: str = 'development'):
        self.environment = environment
        self.debug_mode = environment == 'development'

    def handle_verification_failure(self, failure_type: str, details: Dict[s
        """
        Different handling based on environment
        """
        if self.environment == 'development':
            return self._handle_development_failure(failure_type, details)
        elif self.environment == 'production':
            return self._handle_production_failure(failure_type, details)
        else:
            return self._handle_staging_failure(failure_type, details)

    def _handle_development_failure(self, failure_type: str, details: Dict[s
        """
        Development environment: More lenient, detailed debugging
        """
        print(f"[DEV] AuraSeal Failure: {failure_type}")
        print(f"[DEV] Details: {json.dumps(details, indent=2)}")

        # Allow operation to continue with warning
        return {
            'action': 'continue_with_warning',
            'debug_info': details,
            'can_rollback': True,
            'can_retry': True
        }

    def _handle_production_failure(self, failure_type: str, details: Dict[st
        """
```

```
        Production environment: Strict security, minimal information disclos
        """
        # Log security event (without sensitive details)
        security_log = {
            'timestamp': time.time(),
            'failure_type': failure_type,
            'severity': 'high',
            'action': 'blocked'
        }

        # Immediate security response
        return {
            'action': 'block_and_report',
            'message': 'Security violation detected',
            'user_blocked': True,
            'report_to_admin': True,
            'security_log': security_log
        }

    def _handle_staging_failure(self, failure_type: str, details: Dict[str,
        """
        Staging environment: Balance of security and debugging
        """
        return {
            'action': 'block_with_details',
            'debug_info': details,
            'can_retry': True,
            'security_check': True
        }
```

Installation & Usage

Prerequisites

```
pip install hashlib uuid zipfile dataclasses heapq base64 json os time copy
```

Basic Usage

```
from pyauraseal514 import PhenoAVL, AuraSealStateManager

# Initialize AuraSeal system
auraseal = PhenoAVL(coherence_threshold=0.954)
state_manager = AuraSealStateManager()

# Generate cryptographic keys
public_keys, private_key = auraseal.generate_key_pair()
print(f"Public Key 1: {public_keys[1][:32]}...")
print(f"Public Key 2: {public_keys[2][:32]}...")

# Create secure archive with versioning
project_data = {
    'name': 'SecureProject',
    'version': '1.0.0',
    'files': ['main.py', 'config.json'],
    'checksum': 'abc123def456'
}

# Save state before operation
state_manager.save_state('archive_creation', project_data)

# Create AuraSeal-protected archive
archive_name = auraseal.create_version_archive('./project', project_data)

# Verify archive integrity
is_valid = auraseal.verify_archive_integrity(archive_name)
print(f"Archive integrity: {'VERIFIED' if is_valid else 'FAILED'}")

# Generate 514 hex components for analysis
components = generate_auraseal514_components(json.dumps(project_data))
analysis = analyze_components(components)

print("\n=== Component Analysis ===")
for comp_name, details in analysis.items():
    print(f"{comp_name}: {details['purpose']}")
    print(f"  Length: {details['length']} | Entropy: {details['entropy']:.2f
```

Advanced Usage with Network Security

```
import asyncio
from pyauraseal514 import MobileAuraSealVerifier

async def secure_mobile_download():
    """
    Example of secure mobile download with AuraSeal514
    """
    verifier = MobileAuraSealVerifier()

    try:
        # Download with AuraSeal verification
        result = await verifier.secureDownload(
            'https://secure.example.com/app.zip',
            {'signature': 'expected_auraseal_signature'}
        )

        print("Mobile download verified successfully")
        return result

    except Exception as e:
        print(f"Security violation: {e}")
        # Device access blocked automatically
        return None

# Run secure download
result = asyncio.run(secure_mobile_download())
```

API Reference

Core Classes

PhenoAVL

Main AuraSeal514 cryptographic system.

```
class PhenoAVL:
    def __init__(self, coherence_threshold: float = 0.954)
    def generate_key_pair(self) -> Tuple[Dict[int, str], str]
    def create_version_archive(self, folder_path: str, version_data: Dict) -
    def verify_archive_integrity(self, archive_path: str) -> bool
    def get_system_status(self) -> Dict[str, any]
```

PhenoAVLTrie

Huffman-AVL tree implementation with cryptographic properties.

```
class PhenoAVLTrie:
    def build_huffman_tree(self, text: str) -> None
    def insert(self, key: str, freq: int = 1) -> Optional[PhenoAVLNode]
    def compress_data(self, data: str) -> Tuple[str, float]
    def verify_integrity(self) -> bool
```

Utility Functions

generate_auraseal514_components(data: str) -> List[str]

Generate 514 hex digits distributed across 6 components.

analyze_components(components: List[str]) -> Dict[str, any]

Analyze cryptographic strength of components.

```
verify_sealed_guid(guid: str,
original_data: str) -> bool
```

Verify GUID was generated from original data.

Examples

Example 1: Basic File Protection

```python
#!/usr/bin/env python3

from pyauraseal514 import PhenoAVL
import json

def protect_configuration_file():
    """
    Protect a configuration file with AuraSeal514
    """
    # Configuration data
    config = {
        "database_url": "postgresql://localhost:5432/mydb",
        "api_key": "secret_api_key_12345",
        "debug_mode": False,
        "max_connections": 100
    }

    # Initialize AuraSeal
    auraseal = PhenoAVL()
    keys = auraseal.generate_key_pair()

    # Create protected archive
    archive = auraseal.create_version_archive(
        "./config",
        {"config": config, "protected": True}
    )

    # Generate signature for verification
    signature = auraseal.create_archive_signature(archive, {"config": config

    print(f"Configuration protected in: {archive}")
    print(f"AuraSeal signature: {signature[:32]}...")

    # Verify protection
    is_secure = auraseal.verify_archive_integrity(archive)
    print(f"Protection verified: {is_secure}")
```

```
    return archive, signature


if __name__ == "__main__":
    protect_configuration_file()
```

Example 2: Banking Application Security

```python
#!/usr/bin/env python3

from pyauraseal514 import PhenoAVL, generate_auraseal514_components
import time

class BankingTransactionSecure:
    """
    Banking application with AuraSeal514 security
    """

    def __init__(self):
        self.auraseal = PhenoAVL(coherence_threshold=0.954)
        self.public_keys, self.private_key = self.auraseal.generate_key_pair

    def create_secure_transaction(self, transaction_data: dict) -> dict:
        """
        Create cryptographically secure banking transaction
        """
        # Add temporal component
        transaction_data['timestamp'] = time.time()
        transaction_data['transaction_id'] = self.generate_transaction_id()

        # Generate AuraSeal components
        tx_string = json.dumps(transaction_data, sort_keys=True)
        components = generate_auraseal514_components(tx_string)

        # Create secure transaction package
        secure_transaction = {
            'data': transaction_data,
            'auraseal_components': components,
            'signature': self.auraseal.create_archive_signature(
                f"tx_{transaction_data['transaction_id']}",
                transaction_data
            ),
            'public_key_1': self.public_keys[1],
            'public_key_2': self.public_keys[2],
            'verification_code': self.generate_verification_code(components)
```

```python
        }

        return secure_transaction

    def verify_transaction(self, secure_transaction: dict) -> bool:
        """
        Verify banking transaction integrity
        """
        try:
            # Extract data
            tx_data = secure_transaction['data']
            components = secure_transaction['auraseal_components']
            signature = secure_transaction['signature']

            # Verify components integrity
            if len(components) != 6:
                return False

            # Verify total hex length is 514
            total_hex = sum(len(comp) for comp in components)
            if total_hex != 514:
                return False

            # Verify signature
            expected_sig = self.auraseal.create_archive_signature(
                f"tx_{tx_data['transaction_id']}",
                tx_data
            )

            return signature == expected_sig

        except Exception as e:
            print(f"Transaction verification failed: {e}")
            return False

    def generate_transaction_id(self) -> str:
        """Generate secure transaction ID using AuraSeal"""
```

```
        entropy = f"{time.time()}:{self.private_key[:16]}"
        components = generate_auraseal514_components(entropy)
        return components[0][:16].upper()

    def generate_verification_code(self, components: list) -> str:
        """Generate 6-digit verification code from components"""
        # Use first 6 chars of each component to create verification
        code_chars = []
        for i, component in enumerate(components):
            if len(component) > i:
                code_chars.append(component[i])

        # Convert hex to digits
        verification_code = ""
        for char in code_chars:
            if char.isdigit():
                verification_code += char
            else:
                verification_code += str(ord(char.upper()) % 10)

        return verification_code[:6].ljust(6, '0')

# Usage example
def demo_banking_security():
    bank = BankingTransactionSecure()

    # Create transaction
    transaction = {
        'from_account': '1234567890',
        'to_account': '0987654321',
        'amount': 1500.00,
        'currency': 'USD',
        'description': 'Secure transfer'
    }

    # Secure the transaction
    secure_tx = bank.create_secure_transaction(transaction)
```

```
    print(f"Transaction ID: {secure_tx['data']['transaction_id']}")
    print(f"Verification Code: {secure_tx['verification_code']}")

    # Verify transaction
    is_valid = bank.verify_transaction(secure_tx)
    print(f"Transaction valid: {is_valid}")

    return secure_tx

if __name__ == "__main__":
    demo_banking_security()
```

Example 3: Mobile App Integrity Check

```
#!/usr/bin/env python3

import asyncio
import aiohttp
from pyauraseal514 import PhenoAVL, generate_auraseal514_components

class MobileAppSecurityChecker:
    """
    Mobile application integrity checking with AuraSeal514
    """

    def __init__(self):
        self.auraseal = PhenoAVL()
        self.device_fingerprint = self.generate_device_fingerprint()

    def generate_device_fingerprint(self) -> str:
        """Generate unique device fingerprint"""
        import platform
        import hashlib

        device_info = f"{platform.system()}:{platform.processor()}:{platform
        return hashlib.sha256(device_info.encode()).hexdigest()[:32]

    async def check_app_integrity(self, app_path: str) -> dict:
        """
        Check mobile app integrity using AuraSeal514
        """
        try:
            # Read app file
            with open(app_path, 'rb') as f:
                app_data = f.read()

            # Generate AuraSeal components
            app_string = f"{app_path}:{len(app_data)}:{self.device_fingerpri
            components = generate_auraseal514_components(app_string)

            # Analyze each component
```

```
        integrity_results = {}
        for i, component in enumerate(components):
            integrity_results[f'component_{i}'] = {
                'length': len(component),
                'entropy': self.calculate_entropy(component),
                'valid_hex': all(c in '0123456789ABCDEF' for c in compon
                'strength': self.calculate_component_strength(component)
            }

        # Overall integrity score
        total_strength = sum(result['strength'] for result in integrity_
        integrity_score = min(total_strength / (len(components) * 100),

        return {
            'app_path': app_path,
            'device_fingerprint': self.device_fingerprint,
            'components': components,
            'integrity_results': integrity_results,
            'integrity_score': integrity_score,
            'is_secure': integrity_score >= 0.954,  # AuraSeal threshold
            'total_hex_length': sum(len(c) for c in components)
        }

    except Exception as e:
        return {
            'error': str(e),
            'is_secure': False,
            'integrity_score': 0.0
        }

def calculate_entropy(self, hex_string: str) -> float:
    """Calculate Shannon entropy"""
    import math
    from collections import Counter

    counts = Counter(hex_string.upper())
    length = len(hex_string)
```

```
        entropy = 0
        for count in counts.values():
            probability = count / length
            entropy -= probability * math.log2(probability)

        return entropy

    def calculate_component_strength(self, component: str) -> float:
        """Calculate cryptographic strength of component"""
        # Multiple factors contribute to strength
        length_score = min(len(component) / 86, 1.0) * 30  # Expected ~86 ch
        entropy_score = min(self.calculate_entropy(component) / 4.0, 1.0) *
        distribution_score = self.calculate_distribution_score(component) *

        return length_score + entropy_score + distribution_score

    def calculate_distribution_score(self, component: str) -> float:
        """Calculate how evenly distributed the hex characters are"""
        from collections import Counter

        counts = Counter(component.upper())
        expected_count = len(component) / 16  # 16 possible hex chars

        # Calculate variance from expected distribution
        variance = sum((count - expected_count) ** 2 for count in counts.val
        normalized_variance = variance / (len(component) ** 2)

        # Lower variance = better distribution = higher score
        return max(0, 1.0 - normalized_variance) * 100

# Usage example
async def demo_mobile_security():
    checker = MobileAppSecurityChecker()

    # Check integrity of a mobile app file
    result = await checker.check_app_integrity('./mobile_app.apk')
```

```
    print("=== Mobile App Security Check ===")
    print(f"App: {result.get('app_path', 'N/A')}")
    print(f"Device: {result.get('device_fingerprint', 'N/A')[:16]}...")
    print(f"Integrity Score: {result.get('integrity_score', 0):.3f}")
    print(f"Is Secure: {result.get('is_secure', False)}")
    print(f"Total Hex Length: {result.get('total_hex_length', 0)}")

    if 'integrity_results' in result:
        print("\n=== Component Analysis ===")
        for comp_name, details in result['integrity_results'].items():
            print(f"{comp_name}:")
            print(f"  Length: {details['length']}")
            print(f"  Entropy: {details['entropy']:.2f}")
            print(f"  Strength: {details['strength']:.1f}/100")

    return result

if __name__ == "__main__":
    asyncio.run(demo_mobile_security())
```

Security Considerations

Critical Security Notes

1. **Development vs Production** : The current implementation is a prototype. For production use:
    - Replace simplified hashing with proper cryptographic libraries
    - Implement real asymmetric cryptography for public/private keys
    - Add proper key management and storage

    - Include rate limiting and attack detection

2. **Key Storage** : Never store private keys in plain text. Use:

    - Hardware Security Modules (HSM) for production
    - Encrypted key stores
    - Key derivation functions with salts

3. **Network Security** : Always use HTTPS/TLS when transmitting AuraSeal components

4. **Entropy Sources** : Use cryptographically secure random number generators

Production Hardening Checklist

```
# Production security checklist
PRODUCTION_REQUIREMENTS = {
    'cryptography': [
        'Use cryptographically secure libraries (e.g., cryptography, PyCrypt
        'Implement proper asymmetric cryptography',
        'Use hardware-backed key storage where possible'
    ],
    'network': [
        'Always use TLS 1.3 or higher',
        'Implement certificate pinning',
        'Add request/response integrity checks'
    ],
    'monitoring': [
        'Log all security events',
        'Implement real-time anomaly detection',
        'Set up automated incident response'
    ],
    'testing': [
        'Penetration testing',
        'Cryptographic audit',
        'Performance testing under load'
    ]
}
```

Contributing

Development Setup

```
# Clone repository
git clone https://github.com/obinexus/pyauraseal514.git
cd pyauraseal514

# Create virtual environment
python -m venv venv
source venv/bin/activate  # Linux/Mac
# or
venv\Scripts\activate  # Windows

# Install development dependencies
pip install -r requirements-dev.txt

# Run tests
python -m pytest tests/

# Run security audit
bandit -r src/
safety check
```

Code Standards

- Follow PEP 8 style guidelines
- Maintain 90%+ test coverage
- Document all cryptographic operations
- Use type hints throughout
- Include security considerations in docstrings

Security Review Process

All contributions must pass:

1. Automated security scanning

2. Cryptographic review by maintainers
3. Integration test suite
4. Performance benchmarks

License

Acknowledgments

- **Nnamdi Michael Okpala** - Creator and Lead Developer
- **OBINexus Computing** - Research and Development
- **The Hidden Cipher Research** - Mathematical foundation
- **Perfect Number Theory** - Cryptographic inspiration

Support

- **Email**: consciousness@obinexus.org
- **GitHub Issues**: github.com/obinexus/pyauraseal514/issues (https://github.com/obinexus/pyauraseal514/issues)
- **Documentation**: docs.auraseal514.obinexus.org (https://docs.auraseal514.obinexus.org)

Motto

> ***"When cryptography meets algorithm, we seal them with aura - and I did just that."***

OBINexus Computing • Services from the Heart ❤