OBINexus Cryptographic Interoperability and Pattern Normalization Standard v1.0

A Formal Specification for Regex-Based Primitive Matching
and Isomorphic Reduction in Distributed Cryptographic Systems

Nnamdi Michael Okpala
OBINexus Computing
obinexus.org

July 25, 2025

Abstract

This document establishes the formal specification for cryptographic primitive interoperability within OBINexus computing infrastructure. The standard defines regex-based pattern matching for primitive identification, isomorphic reduction principles for canonical state normalization, and cross-language compatibility requirements. By treating cryptographic primitives as finite automaton states with regex fingerprints, this specification ensures deterministic behavior across heterogeneous environments while preventing encoding-based exploit vectors. The framework implements structural equivalence detection that operates across all levels of the Chomsky hierarchy, providing both computational efficiency and robust security properties for distributed cryptographic operations.

Contents

1 Introduction

Modern cryptographic systems face increasing complexity in managing diverse primitive representations across programming languages, encoding schemes, and system architectures. Traditional approaches treat each primitive variant as a distinct computational problem, leading to bloated parsers, increased attack surfaces, and significant maintenance overhead.

The OBINexus Cryptographic Interoperability Standard addresses these challenges through a unified framework based on regular expression pattern matching and isomorphic reduction principles. This specification defines how cryptographic primitives are identified, validated, and normalized across all OBINexus infrastructure components.

1.1 Scope and Applicability

This standard applies to all OBINexus cryptographic operations, including but not limited to:

- Key derivation and management systems
- Digital signature verification protocols
- Encrypted storage and transmission mechanisms
- Authentication and authorization frameworks
- Audit logging and compliance systems

1.2 Design Principles

The standard is built upon three fundamental principles derived from cryptographic theory [1]:

1. **Hardness**: Ensuring problems are easy to verify but hard to solve without knowledge
2. **Completeness**: Guaranteeing the system can detect anomalies without relying on external assumptions
3. **Soundness**: Ensuring inputs are only accepted from verified sources, preventing manipulation by randomness or guesswork

2 Pattern Matching and Identification

2.1 Regex Pattern Enforcement Policy

All cryptographic primitive identification within OBINexus systems SHALL use deterministic regular expression patterns. The enforcement policy implements mandatory control logic for primitive validation:

```
def enforce_primitive_pattern(primitive_digest: str,
                              context: str) -> ValidationResult:
    """
    Mandatory pattern enforcement for cryptographic primitives.
    MUST implement if/else control logic as specified.
    """
    # Phase 1: Pattern Recognition
    matched_patterns = []
    for pattern_state in REGISTERED_PATTERNS:
        if pattern_state.matches(primitive_digest):
            matched_patterns.append(pattern_state)

    # Phase 2: Collision Resolution (if multiple matches)
```

```
    if len(matched_patterns) == 0:
        return ValidationResult.REJECT_UNKNOWN_PATTERN
    elif len(matched_patterns) == 1:
        return ValidationResult.ACCEPT_SINGLE_MATCH
    else:
        # Longest pattern wins (most specific)
        canonical_pattern = max(matched_patterns,
                                key=lambda p: len(p.pattern))
        return ValidationResult.ACCEPT_CANONICAL_RESOLUTION

    # Phase 3: Security Level Validation
    if canonical_pattern.security_level == "deprecated":
        if context in LEGACY_ALLOWED_CONTEXTS:
            return ValidationResult.ACCEPT_LEGACY_CONTEXT
        else:
            return ValidationResult.REJECT_DEPRECATED_SECURITY

    return ValidationResult.ACCEPT_VALIDATED
```

Listing 1: Mandatory Pattern Enforcement Logic

2.2 Pattern Registration Format

Cryptographic primitive patterns MUST conform to the following format specification:

```
<ALGORITHM>-<KEYSIZE>:[<HEX_PATTERN>]{<LENGTH>}
```

Examples of valid pattern registrations:

- `RSA-2048:[a-f0-9]{512}` - RSA 2048-bit key with 512-character hex digest
- `AES-256:[a-f0-9]{64}` - AES 256-bit key with 64-character hex digest
- `ECDSA-P256:[a-f0-9]{128}` - ECDSA P-256 curve with 128-character hex digest

3 Primitive Lifecycle and Policy

3.1 Version Evolution Model

The primitive lifecycle implements non-destructive version evolution where new cryptographic standards register additional patterns without invalidating existing implementations. This approach ensures backward compatibility while enabling controlled security upgrades.

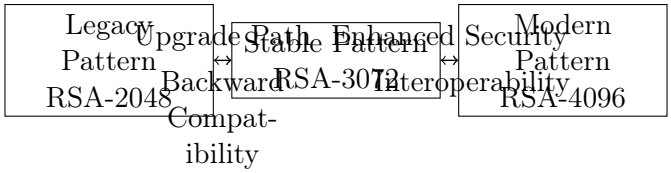

Figure 1: Primitive Version Evolution with Backward Compatibility

3.2 Pattern Compatibility Matrix

The compatibility matrix defines which primitive versions can interoperate:

Algorithm	Legacy	Stable	Modern	Experimental
RSA-2048	✓	✓	✓	✗
RSA-3072	✓	✓	✓	✓
RSA-4096	✗	✓	✓	✓
AES-256	✓	✓	✓	✓
ECDSA-P256	✓	✓	✓	✗
ECDSA-P384	✗	✓	✓	✓

Table 1: Cryptographic Primitive Compatibility Matrix

4 Isomorphic Reduction Principle

4.1 Theoretical Foundation

Building upon the Unicode-Only Structural Charset Normalizer framework [2], this standard applies isomorphic reduction principles to cryptographic primitive normalization. The key insight is that structurally equivalent primitive representations should be treated as identical automaton states.

Definition 1 (Cryptographic Isomorphic Reduction): Let P_1 and P_2 be primitive representations from different encoding schemes. Primitives P_1 and P_2 are isomorphically reducible if there exists a structure-preserving mapping $\phi : P_1 \to P_2$ such that:

$$\phi(P_1) \equiv P_2 \text{ (cryptographically equivalent)}$$

Theorem 1 (Primitive Equivalence): For any set of cryptographic primitive encodings $E = \{e_1, e_2, \ldots, e_n\}$ representing the same cryptographic content, there exists a minimal canonical form c such that:

$$\forall e_i \in E : \phi(e_i) = c$$

where ϕ is a normalization function preserving cryptographic equivalence.

4.2 Automaton-Based Primitive Modeling

Cryptographic primitive recognition is modeled as a finite state automaton problem where each encoding variant represents a distinct path through the automaton that leads to the same accepting state representing the canonical primitive.

Definition 2 (Cryptographic Primitive Automaton): A Cryptographic Primitive Automaton (CPA) is defined as the 5-tuple:

$$A_{CP} = (Q, \Sigma, \delta, q_0, F)$$

where:

- Q is the finite set of primitive states
- Σ is the input alphabet (including encoded primitive digests)
- $\delta : Q \times \Sigma \to Q$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states representing validated primitives

5 Canonical Mapping Functions

5.1 Pattern Normalization Algorithm

The canonical mapping function implements structural equivalence detection across multiple primitive encoding schemes:

Algorithm 1 Cryptographic Primitive State Minimization

Require: Set of primitive encodings E, Primitive Automaton A_{CP}
Ensure: Minimized automaton A_{min} with canonical mappings

Initialize equivalence classes: $C = \{F, Q \setminus F\}$
$changed \leftarrow true$
while *changed* **do**
  $changed \leftarrow false$
  for each class $C_i \in C$ **do**
    Initialize signature groups: $splits \leftarrow \emptyset$
    for each state $q \in C_i$ **do**
      Compute transition signature: $signature(q) = \{(\sigma, class(\delta(q, \sigma)))|\sigma \in \Sigma\}$
      Add q to $splits[signature(q)]$
    end for
    if $|splits| > 1$ **then**
      Replace C_i with $splits.values()$ in C
      $changed \leftarrow true$
    end if
  end for
end while
Construct A_{min} using equivalence classes as states
return A_{min}, canonical mapping for each encoding

5.2 Canonical Form Examples

The following examples demonstrate canonical mapping for common primitive encoding variations:

```
# RSA Key Encoding Variations -> Canonical Form
phi("rsa-2048:a1b2c3d4...") = "RSA-2048:A1B2C3D4..."
phi("RSA_2048:a1b2c3d4...") = "RSA-2048:A1B2C3D4..."
phi("rsa2048:a1b2c3d4...")  = "RSA-2048:A1B2C3D4..."

# AES Key Encoding Variations -> Canonical Form
phi("aes-256:f1e2d3c4...")  = "AES-256:F1E2D3C4..."
phi("AES_256:f1e2d3c4...")  = "AES-256:F1E2D3C4..."
phi("aes256:f1e2d3c4...")   = "AES-256:F1E2D3C4..."

# Hash Digest Variations -> Canonical Form
phi("sha256:0x1a2b3c...")   = "SHA256:1A2B3C..."
phi("SHA-256:1a2b3c...")    = "SHA256:1A2B3C..."
phi("sha_256:1a2b3c...")    = "SHA256:1A2B3C..."
```

Listing 2: Canonical Mapping Examples

6 Unicode Normalization and Regex Consistency

6.1 Cross-Language Deterministic Regex Subset

To ensure consistent behavior across Python, Lua, and C implementations, this standard defines a restricted regex subset that provides deterministic matching across all supported languages:

Allowed Regex Constructs:

- Character classes: `[a-f0-9]`, `[A-Z]`, `[0-9]`
- Quantifiers: `{n}`, `{n,m}`
- Anchors: `^`, `$`
- Literals: Alphanumeric characters and hyphens

Prohibited Regex Constructs:

- Lookaheads and lookbehinds: `(?=...)`, `(?!...)`
- Non-greedy quantifiers: `*?`, `+?`
- Unicode categories: `\p{...}`
- Word boundaries: `\b`

6.2 Unicode Character Normalization

All primitive digest strings SHALL be normalized using the Unicode-Only Structural Charset Normalizer (USCN) approach before pattern matching. This eliminates encoding-based exploit vectors:

```
def normalize_primitive_input(input_string: str) -> str:
    """
    Apply USCN normalization to eliminate encoding variations.
    """
    # Phase 1: Decode common encoding variations
    normalized = input_string
    for encoding_variant, canonical in ENCODING_MAPPINGS.items():
        normalized = normalized.replace(encoding_variant, canonical)

    # Phase 2: Case normalization for hex strings
    if is_hex_string(normalized):
        normalized = normalized.upper()

    # Phase 3: Delimiter standardization
    normalized = standardize_delimiters(normalized)

    return normalized
```

Listing 3: Unicode Normalization Implementation

7 Security Guarantees

7.1 Exploit Vector Elimination

The isomorphic reduction approach eliminates common cryptographic primitive exploit vectors by normalizing all inputs to canonical forms before validation.

Proposition 1 (Security Invariant): For any primitive input string s containing encoded characters, the OBINexus standard guarantees:

$$validate(normalize(s)) \equiv validate(canonical(s))$$

where validation occurs exclusively on the normalized canonical form, preventing encoding-based bypasses.

7.2 Path Traversal Prevention for Primitive References

Traditional primitive reference validation is vulnerable to encoding variations:

```
def vulnerable_primitive_check(primitive_ref: str) -> bool:
    # Fails for encoding variants
    if "../" in primitive_ref:
        return False  # Blocked
    return True

# These bypass the filter:
vulnerable_primitive_check("%2e%2e%2f")  # Returns True (BYPASS)
vulnerable_primitive_check("%c0%af")     # Returns True (BYPASS)
```

Listing 4: Vulnerable Primitive Reference Validation

The OBINexus standard prevents these exploits through structural normalization:

```
def secure_primitive_check(primitive_ref: str) -> bool:
    # Normalizes ALL variants before validation
    canonical = normalize_primitive_input(primitive_ref)
    if "../" in canonical:
        return False  # Blocked
    return True

# All encoding variants are caught:
secure_primitive_check("%2e%2e%2f")  # Returns False (SECURE)
secure_primitive_check("%c0%af")     # Returns False (SECURE)
secure_primitive_check("../")        # Returns False (SECURE)
```

Listing 5: Hardened Primitive Reference Validation

8 Audit Trail and Logging Syntax

8.1 Secure Pattern Hash References

To prevent information disclosure while maintaining audit integrity, all log entries SHALL reference cryptographic patterns using secure hash identifiers rather than exposing raw pattern strings.

```
class SecureAuditNode:
    """Secure audit trail node for primitive operations."""

    def __init__(self, primitive_digest: str, pattern_state: State):
        self.timestamp = datetime.utcnow()
        self.primitive_hash = sha256(primitive_digest).hexdigest()[:16]
        self.pattern_hash = sha256(pattern_state.pattern).hexdigest()[:16]
        self.operation_context = get_current_context()
        # Never store raw primitive_digest or pattern in logs

    def to_audit_record(self) -> dict:
        return {
            "timestamp": self.timestamp.isoformat(),
            "primitive_ref": f"PRIM_{self.primitive_hash}",
            "pattern_ref": f"PAT_{self.pattern_hash}",
            "context": self.operation_context,
```

```
            "compliance_level": "OBINexus-v1.0"
        }
```

Listing 6: Audit Trail Pattern Reference Implementation

8.2 Audit Trail Compliance Requirements

All OBINexus cryptographic operations MUST generate audit records conforming to this specification:

Field	Format	Description
timestamp	ISO 8601 UTC	Operation execution time
primitive_ref	PRIM_[16-char hex]	Hashed primitive identifier
pattern_ref	PAT_[16-char hex]	Hashed pattern identifier
context	String	Operation context identifier
compliance_level	OBINexus-v1.0	Standard version compliance

Table 2: Mandatory Audit Record Format

9 Formal Definitions

9.1 Regex Automaton Definition

Definition 3 (Regex Automaton): A Regex Automaton AR for cryptographic primitive matching is a 5-tuple:

$$AR = (Q_R, \Sigma, \delta_R, q_0, F)$$

where each state $q \in Q_R$ is associated with a regular expression pattern $pattern(q)$ such that the automaton accepts input string w if and only if w matches $pattern(q_f)$ for some final state $q_f \in F$ reachable from q_0 via input w.

9.2 Pattern Compatibility Matrix Definition

Definition 4 (Compatibility Matrix): The Pattern Compatibility Matrix M is a function:

$$M : PatternSpace \times PatternSpace \rightarrow \{COMPATIBLE, INCOMPATIBLE, LEGACY\_ONLY\}$$

where $PatternSpace$ represents the set of all registered cryptographic primitive patterns.

9.3 Canonical Transition Model

Definition 5 (Canonical Transition): A canonical transition is a mapping $T : State \times Input \rightarrow CanonicalState$ where multiple equivalent input encodings map to the same canonical state, ensuring deterministic behavior regardless of input encoding variation.

10 Cross-Language Compatibility

10.1 Implementation Requirements

All OBINexus cryptographic systems MUST implement pattern matching using the cross-language compatible regex subset defined in Section 5.1. Reference implementations SHALL be provided for:

- Python 3.8+ using the `re` module
- Lua 5.4+ using POSIX-compatible patterns
- C99+ using POSIX regex library (`regex.h`)

10.2 Validation Framework

A cross-language validation framework ensures consistent behavior across all implementations:

```
def validate_cross_language_consistency():
    """Validate pattern matching consistency across languages."""
    test_cases = [
        ("RSA-2048:A1B2C3...", "rsa-2048:a1b2c3..."),
        ("AES-256:F1E2D3...", "aes_256:f1e2d3..."),
        ("SHA256:1A2B3C...", "sha-256:1a2b3c...")
    ]

    for canonical, variant in test_cases:
        python_result = python_normalize(variant)
        lua_result = lua_normalize(variant)
        c_result = c_normalize(variant)

        assert python_result == lua_result == c_result == canonical
        assert all_match_pattern(canonical, REGISTERED_PATTERNS)
```

Listing 7: Cross-Language Validation Test

11 Compliance and Enforcement

11.1 Mandatory Implementation Requirements

All systems claiming OBINexus v1.0 compatibility MUST:

1. Implement the complete pattern enforcement policy (Section 2.1)
2. Support primitive lifecycle management (Section 3)
3. Apply isomorphic reduction normalization (Section 4)
4. Use secure audit trail format (Section 6)
5. Pass cross-language validation tests (Section 8.2)

11.2 Compliance Verification

Compliance verification SHALL be performed using the OBINexus Cryptographic Test Suite, which validates:

- Pattern matching determinism across supported languages
- Canonical mapping function correctness
- Security invariant preservation under encoding variations
- Audit trail format adherence
- Performance benchmarks for production deployment

12 Conclusion

The OBINexus Cryptographic Interoperability and Pattern Normalization Standard provides a mathematically rigorous foundation for cryptographic primitive identification and validation across heterogeneous computing environments. By applying isomorphic reduction principles and deterministic pattern matching, this standard eliminates common exploit vectors while ensuring backward compatibility and cross-language consistency.

As stated in the foundational OBINexus principle: "Structure is the final syntax" [2]. This standard embodies that philosophy by making structural equivalence the foundation for cryptographic security rather than surface-level pattern enumeration.

Implementation of this standard ensures that OBINexus cryptographic infrastructure operates with deterministic behavior, robust security properties, and comprehensive audit capabilities across all deployment environments.

References

[1] Okpala, N.M. (2025). *Principles of Cryptography: Hardness, Completeness, and Soundness.* OBINexus Computing Technical Documentation.

[2] Okpala, N.M. (2025). *Unicode-Only Structural Charset Normalizer: Isomorphic Reduction as a Feature, Not a Bug.* OBINexus Computing.

[3] Okpala, N.M. (2025). *Building the Future of Encryption - A Proposal for Standardised Cryptographic Primitives.* OBINexus Medium Publication.

[4] Chomsky, N. (1956). *Three models for the description of language.* IRE Transactions on Information Theory, 2(3), 113-124.

[5] Myhill, J. (1957). *Finite automata and their decision problems.* IBM Journal of Research and Development, 1(1), 4-14.

[6] Nerode, A. (1958). *Linear automaton transformations.* Proceedings of the American Mathematical Society, 9(4), 541-544.