BIOCYPHER

Technical Guide and Protocol Specification

DNA-Based Data Encoding for Secure Communication and Long-Term Storage

BioCypher Project

Protocol Version 1.0

Open Protocol Specification

December 21, 2025

Abstract

BioCypher is an open-source application and protocol for encoding digital data into DNA sequences. This technical guide provides comprehensive documentation for implementing, deploying, and extending the BioCypher protocol. The system supports three encoding modes optimized for different use cases: basic binary-to-DNA mapping, nanopore sequencing optimization, and AES-256 encryption. BioCypher enables secure communication through biological carriers, long-term archival storage, and surveillance-resistant information transfer.

Document Information

Version:	1.0
Status:	Formal Specification
License:	Open Protocol
Repository:	https://github.com/SampleBias/biocypher

Contents

Part I
Introduction and Architecture

1 Introduction

1.1 What is BioCypher?

BioCypher is both an application and an open protocol for encoding digital information into DNA sequences. Unlike traditional digital storage, DNA-based encoding offers:

- **Unprecedented Storage Density**: 215 petabytes per gram (theoretical)
- **Extreme Longevity**: Data stability for thousands of years
- **Surveillance Resistance**: Physical carriers bypass digital interception
- **No Power Requirements**: DNA requires no electricity for storage
- **Biological Compatibility**: Integration with living organisms

1.2 Protocol Philosophy

BioCypher is designed as an *open protocol*, meaning:

1. **Interoperability**: Any compliant implementation can encode/decode BioCypher sequences
2. **Transparency**: All encoding algorithms and cryptographic methods are publicly documented
3. **No Vendor Lock-In**: Multiple implementations can coexist
4. **Community Extensibility**: Protocol can be extended through community proposals
5. **Scientific Rigor**: All methods are based on peer-reviewed research

1.3 Use Cases

1.3.1 Secure Communication

Encode messages into DNA, embed in biological carriers (bacteria, plants, living tissue), transport physically, and decode at destination. Bypasses digital surveillance entirely.

1.3.2 Long-Term Archival

Store critical data (cultural heritage, scientific knowledge, personal records) in DNA format for multi-generational preservation.

1.3.3 Distributed Backup

Replicate encoded DNA across multiple geographic locations and biological substrates for redundancy.

1.3.4 Steganographic Applications

Hide data within natural-appearing DNA sequences (genomic steganography).

1.3.5 Space Exploration

Include biological archives in interstellar missions for knowledge preservation across cosmic timescales.

1.4 System Requirements

1.4.1 For Encoding

- Computer with BioCypher application or compatible implementation
- Text editor or data source
- DNA synthesis service (e.g., IDT, Twist Bioscience) or access to synthesis equipment

1.4.2 For Decoding

- DNA sequencing capability (Oxford Nanopore MinION recommended)
- Computer with BioCypher application
- DNA extraction capabilities (if decoding from carriers)

2 Protocol Architecture

2.1 System Overview

The BioCypher protocol consists of four primary layers:

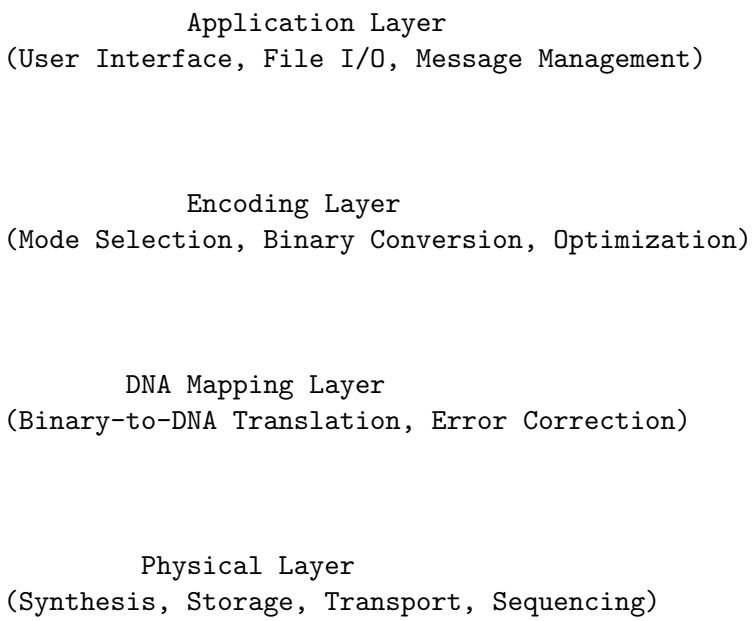

Figure 1: BioCypher Protocol Stack

Algorithm 1 BioCypher Encoding Pipeline

Input: Text message M, mode *mode*, optional password P
Output: DNA sequence D

if *mode* = secure and P is provided **then**
    $M \leftarrow \text{AES\_Encrypt}(M, P)$
    $M \leftarrow \text{Base64\_Encode}(M)$
end if

$B \leftarrow \text{Text\_To\_Binary}(M)$

if *mode* = nanopore **then**
    $B \leftarrow \text{Add\_Parity\_Bits}(B)$
    $B \leftarrow \text{Apply\_Error\_Correction}(B)$
    $D \leftarrow \text{Triplet\_Encode}(B)$
    $D \leftarrow \text{Optimize\_For\_Nanopore}(D)$
    $D \leftarrow \text{Add\_Markers}(D)$
else if *mode* = secure **then**
    $D \leftarrow \text{Binary\_To\_DNA}(B)$
    $D \leftarrow \text{Add\_Markers}(D)$
else ▷ basic mode
    $D \leftarrow \text{Binary\_To\_DNA}(B)$
end if

return D

2.2 Data Flow

2.2.1 Encoding Pipeline

2.2.2 Decoding Pipeline

2.3 Protocol Versioning

BioCypher uses semantic versioning: `MAJOR.MINOR.PATCH`

- **MAJOR**: Incompatible protocol changes
- **MINOR**: Backward-compatible functionality additions
- **PATCH**: Backward-compatible bug fixes

Current version: **1.0.0**

Algorithm 2 BioCypher Decoding Pipeline

Input: DNA sequence D, mode *mode*, optional password P
Output: Text message M

if $mode \in \{\text{nanopore}, \text{secure}\}$ **then**
    $D \leftarrow \text{Remove\_Markers}(D)$
end if

if $mode = \text{nanopore}$ **then**
    $D \leftarrow \text{Remove\_Padding}(D)$
    $B \leftarrow \text{Triplet\_Decode}(D)$
    $B \leftarrow \text{Apply\_Error\_Correction}(B)$
    $B \leftarrow \text{Check\_Parity}(B)$
else
    $B \leftarrow \text{DNA\_To\_Binary}(D)$
end if

$M \leftarrow \text{Binary\_To\_Text}(B)$

if $mode = \text{secure}$ and P is provided **then**
    $M \leftarrow \text{Base64\_Decode}(M)$
    $M \leftarrow \text{AES\_Decrypt}(M, P)$
end if

return M

Part II
Encoding Modes and Specifications

3 Encoding Mode Overview

BioCypher supports three encoding modes, each optimized for specific use cases:

Mode	Use Case	Markers	Error Correction
`basic`	Simple encoding, max density	No	No
`nanopore`	Sequencing optimization	Yes	Yes
`secure`	Encrypted storage	Yes	No

Table 1: BioCypher Encoding Modes

4 Binary-to-DNA Mapping

4.1 Standard Mapping Table

The fundamental encoding used in Basic and Secure modes:

Binary Pair	DNA Base	Chemical Name
`00`	`A`	Adenine
`01`	`T`	Thymine
`10`	`C`	Cytosine
`11`	`G`	Guanine

Table 2: Standard Binary-to-DNA Mapping

4.2 Mapping Properties

This mapping was chosen for:

- **Simplicity**: Direct 2-bit to 1-base correspondence
- **Efficiency**: 2 bits per nucleotide = 4 characters per byte
- **Universality**: Uses only standard DNA bases
- **Reversibility**: Unambiguous bidirectional mapping

4.3 Encoding Rules

1. Binary strings are processed in 2-bit pairs
2. If binary length is odd, pad with `0` before final pair
3. Output is concatenated DNA sequence
4. No delimiters between bases in basic mode

4.4 Decoding Rules

1. Each DNA base maps to a 2-bit binary pair
2. Invalid bases (not A, T, C, G) are skipped with warning
3. Binary string is chunked into 8-bit bytes
4. Each byte is converted to ASCII character
5. Non-printable characters are filtered (optional)

5 Basic Mode Specification

5.1 Overview

Basic mode provides the simplest encoding: direct binary-to-DNA conversion with no overhead. This mode is optimal for:

- Maximum storage density
- Simple encoding/decoding workflows
- Applications where error correction is handled externally
- Testing and validation

5.2 Encoding Algorithm

```python
def encode_basic(message: str) -> str:
    """
    Encode message to DNA using basic mode.

    Args:
        message: Input text message

    Returns:
        DNA sequence (string of A, T, C, G)
    """
    # Convert message to binary
    binary = ''.join(format(ord(char), '08b') for char in message)

    # Map binary pairs to DNA bases
    dna_map = {'00': 'A', '01': 'T', '10': 'C', '11': 'G'}
    dna = ''

    for i in range(0, len(binary), 2):
        pair = binary[i:i+2]
        if len(pair) == 2:
            dna += dna_map[pair]
        else:  # Odd length, pad with 0
            pair += '0'
            dna += dna_map[pair]

    return dna
```

Listing 1: Basic Mode Encoding

5.3 Decoding Algorithm

```python
def decode_basic(dna: str) -> str:
    """
    Decode DNA sequence to message using basic mode.

    Args:
        dna: DNA sequence (A, T, C, G)

    Returns:
        Decoded text message
    """
    # Map DNA bases to binary
    binary_map = {'A': '00', 'T': '01', 'C': '10', 'G': '11'}
    binary = ''

    for base in dna.upper():
        if base in binary_map:
            binary += binary_map[base]
        else:
            # Skip invalid bases
            continue

    # Convert binary to text
    message = ''
```

```
    for i in range(0, len(binary), 8):
        byte = binary[i:i+8]
        if len(byte) == 8:
            ascii_val = int(byte, 2)
            if 0 <= ascii_val <= 255:
                message += chr(ascii_val)

    return message
```

Listing 2: Basic Mode Decoding

5.4 Format Specification

Basic Mode Format

`[DNA Sequence]`

Components:

- No start/stop markers
- No padding or delimiters
- Raw DNA sequence only

5.5 Examples

Input	Binary	DNA
"A"	01000001	TAAA
"Hi"	01001000 01101001	TAAATAAA TATA
"DNA"	01000100 01001110 01000001	TAAATAGC TAACATAA TAAA

Table 3: Basic Mode Examples

5.6 Performance Characteristics

- **Encoding Speed**: $O(n)$ where n is message length
- **Decoding Speed**: $O(m)$ where m is DNA sequence length
- **Memory Usage**: $O(n)$ for both encoding and decoding
- **Storage Efficiency**: 4 DNA bases per character (optimal)

6 Nanopore Mode Specification

6.1 Overview

Nanopore mode optimizes DNA sequences for Oxford Nanopore sequencing technology. This mode addresses specific challenges in nanopore sequencing:

- **Homopolymer Errors**: Consecutive identical bases cause signal compression

- **GC Content Bias**: Extreme GC ratios affect translocation speed
- **Secondary Structures**: Hairpins and G-quadruplexes impede sequencing
- **Read Dropouts**: Missing data requires redundancy

6.2 Triplet Encoding

Nanopore mode uses triplet encoding (3 bits → 3 bases) instead of binary pairs:

Binary Triplet	DNA Triplet	Properties
000	ATC	No homopolymers, balanced GC
001	ATG	No homopolymers, balanced GC
010	ACT	No homopolymers, balanced GC
011	ACG	No homopolymers, high GC
100	TAG	No homopolymers, balanced GC
101	TAC	No homopolymers, balanced GC
110	TCG	No homopolymers, high GC
111	TCA	No homopolymers, balanced GC

Table 4: Nanopore Triplet Encoding Table

Important Note

Triplet encoding eliminates all homopolymer runs by design. Each triplet contains exactly 3 different bases or carefully controlled alternation.

6.3 Error Correction Scheme

Nanopore mode implements triple redundancy error correction:

Algorithm 3 Triple Redundancy Encoding

Input: Binary string B
Output: Error-corrected binary string $B_{corrected}$

for each bit b in B **do**
    $B_{corrected} \leftarrow B_{corrected} + b + b + b$
end for
return $B_{corrected}$

Algorithm 4 Triple Redundancy Decoding

Input: Error-corrected binary string $B_{corrected}$
Output: Corrected binary string B

for each triplet (b_1, b_2, b_3) in $B_{corrected}$ **do**
    $majority \leftarrow \text{most\_common}(b_1, b_2, b_3)$
    $B \leftarrow B + majority$
end for
return B

This scheme can correct single-bit errors within each triplet:

- `000` → `0` (correct)
- `100` → `0` (single error corrected)
- `110` → `1` (two errors, incorrect majority)

6.4 Parity Bits

Each character (8 bits) gets an additional parity bit for error detection:

```python
def add_parity(binary: str) -> str:
    """Add parity bit to each 8-bit chunk."""
    result = ''
    for i in range(0, len(binary), 8):
        byte = binary[i:i+8]
        if len(byte) == 8:
            parity = str(byte.count('1') % 2)  # Even parity
            result += byte + parity  # 9 bits total
    return result
```

Listing 3: Parity Bit Addition

During decoding, parity is checked:

- If parity matches: Accept byte
- If parity fails: Reject byte or flag error

6.5 Sequence Optimization

6.5.1 GC Content Balancing

Optimal GC content for nanopore sequencing: 40-60%

```python
def calculate_gc_content(dna: str) -> float:
    """Calculate GC percentage."""
    gc_count = dna.count('G') + dna.count('C')
    return (gc_count / len(dna)) * 100 if len(dna) > 0 else 0.0

def is_gc_balanced(dna: str) -> bool:
    """Check if GC content is in optimal range."""
    gc = calculate_gc_content(dna)
    return 40.0 <= gc <= 60.0
```

Listing 4: GC Content Calculation

If GC content is outside range, padding is added to balance.

6.5.2 Homopolymer Detection

Banned patterns (detected via regex):

- `AA+` (2+ consecutive A)
- `TT+` (2+ consecutive T)
- `CC+` (2+ consecutive C)

- `GG+` (2+ consecutive G)
- `ATATAT` (repetitive alternating)
- `GCGCGC` (repetitive GC dinucleotides)

```python
import re

def has_homopolymers(dna: str) -> bool:
    """Check for problematic homopolymer runs."""
    patterns = [
        r'AA+',       # 2+ consecutive A
        r'TT+',       # 2+ consecutive T
        r'CC+',       # 2+ consecutive C
        r'GG+',       # 2+ consecutive G
        r'(AT){3,}', # Repetitive AT
        r'(GC){3,}'  # Repetitive GC
    ]

    for pattern in patterns:
        if re.search(pattern, dna):
            return True
    return False
```

Listing 5: Homopolymer Detection

6.6 Padding Mechanism

If optimization is needed (GC imbalance or homopolymers), padding is added:

```python
def generate_padding(dna: str) -> str:
    """
    Generate padding to balance GC content.

    Padding uses a repeating pattern designed to:
    - Balance GC content
    - Avoid homopolymers
    - Be easily identifiable for removal
    """
    padding_pattern = "ATCATGACTACG"
    padding_length = max(6, len(dna) // 10)

    # Repeat pattern to reach desired length
    repeats = (padding_length // len(padding_pattern)) + 1
    padding = (padding_pattern * repeats)[:padding_length]

    return padding
```

Listing 6: Padding Generation

Padding format:

`[padding] TACGTA [core sequence] TACGTA [padding]`

Where `TACGTA` is the padding delimiter.

Marker Type	Sequence
Start Marker	`ATCGATCG` (8 bases)
Stop Marker	`CGATATCG` (8 bases)
Padding Delimiter	`TACGTA` (6 bases)

Table 5: Nanopore Mode Markers

6.7 Marker Sequences

Nanopore mode uses standardized markers:

These markers were chosen to:

- Be easily identifiable
- Avoid homopolymers
- Be unlikely to occur naturally in data
- Have balanced GC content

6.8 Complete Nanopore Encoding Algorithm

```
def encode_nanopore(message: str, error_correction: bool = True) -> str
    :
    """
    Encode message using nanopore-optimized mode.

    Args:
        message: Input text
        error_correction: Enable triple redundancy

    Returns:
        Nanopore-optimized DNA sequence
    """
    # Step 1: Convert to binary with parity
    binary = ''
    for char in message:
        byte = format(ord(char), '08b')
        parity = str(byte.count('1') % 2)
        binary += byte + parity  # 9 bits per char

    # Step 2: Apply error correction if enabled
    if error_correction:
        corrected = ''
        for bit in binary:
            corrected += bit * 3  # Triple each bit
        binary = corrected

    # Step 3: Pad to multiple of 3
    while len(binary) % 3 != 0:
        binary += '0'

    # Step 4: Triplet encoding
    triplet_map = {
        '000': 'ATC', '001': 'ATG', '010': 'ACT', '011': 'ACG',
```

```
        '100': 'TAG', '101': 'TAC', '110': 'TCG', '111': 'TCA'
    }

    dna = ''
    for i in range(0, len(binary), 3):
        triplet = binary[i:i+3]
        dna += triplet_map[triplet]

    # Step 5: Check optimization needs
    needs_optimization = (
        has_homopolymers(dna) or
        not is_gc_balanced(dna)
    )

    if needs_optimization:
        padding = generate_padding(dna)
        delimiter = "TACGTA"
        dna = padding + delimiter + dna + delimiter + padding

    # Step 6: Add markers
    dna = "ATCGATCG" + dna + "CGATATCG"

    return dna
```

Listing 7: Complete Nanopore Encoding

6.9 Complete Nanopore Decoding Algorithm

```
def decode_nanopore(dna: str, error_correction: bool = True) -> str:
    """
    Decode nanopore-optimized DNA sequence.

    Args:
        dna: DNA sequence with markers
        error_correction: Expect triple redundancy

    Returns:
        Decoded text message
    """
    # Step 1: Remove markers
    if dna.startswith("ATCGATCG"):
        dna = dna[8:]
    if dna.endswith("CGATATCG"):
        dna = dna[:-8]

    # Step 2: Remove padding if present
    delimiter = "TACGTA"
    first_delim = dna.find(delimiter)
    last_delim = dna.rfind(delimiter)

    if first_delim != -1 and last_delim != -1 and first_delim !=
   last_delim:
        dna = dna[first_delim + 6:last_delim]

    # Step 3: Triplet decoding
    triplet_map_reverse = {
```

```
        'ATC': '000', 'ATG': '001', 'ACT': '010', 'ACG': '011',
        'TAG': '100', 'TAC': '101', 'TCG': '110', 'TCA': '111'
    }

    binary = ''
    for i in range(0, len(dna), 3):
        triplet = dna[i:i+3]
        if triplet in triplet_map_reverse:
            binary += triplet_map_reverse[triplet]

    # Step 4: Apply error correction if enabled
    if error_correction:
        corrected = ''
        for i in range(0, len(binary), 3):
            group = binary[i:i+3]
            if len(group) == 3:
                # Majority vote
                ones = group.count('1')
                corrected += '1' if ones >= 2 else '0'
        binary = corrected

    # Step 5: Convert binary to text with parity check
    message = ''
    for i in range(0, len(binary), 9):
        chunk = binary[i:i+9]
        if len(chunk) == 9:
            data_bits = chunk[0:8]
            parity_bit = chunk[8]

            # Check parity
            expected_parity = str(data_bits.count('1') % 2)
            if parity_bit == expected_parity:
                ascii_val = int(data_bits, 2)
                if 0 <= ascii_val <= 255:
                    message += chr(ascii_val)

    return message
```

Listing 8: Complete Nanopore Decoding

6.10 Format Specification

Nanopore Mode Format

Without Padding:

`ATCGATCG [Core Sequence] CGATATCG`

With Padding:

`ATCGATCG [Padding] TACGTA [Core Sequence] TACGTA [Padding] CGATATCG`

Components:

- Start marker: `ATCGATCG` (always present)
- Padding delimiter: `TACGTA` (only if padding added)
- Core sequence: Triplet-encoded data
- Stop marker: `CGATATCG` (always present)

6.11 Performance Characteristics

- **Encoding Speed**: $O(n)$ with higher constant than basic mode
- **Decoding Speed**: $O(m)$ with error correction overhead
- **Memory Usage**: $O(3n)$ due to triple redundancy
- **Storage Efficiency**: 3 DNA bases per bit (with error correction)
- **Error Correction Capability**: Single-bit errors within triplets

7 Secure Mode Specification

7.1 Overview

Secure mode combines AES-256-CBC encryption with DNA encoding for cryptographically secure data storage. This mode is designed for:

- Highly sensitive information
- Data requiring confidentiality over decades
- Multi-party secure communication
- Compliance with security standards (FIPS 140-2)

7.2 Cryptographic Specification

Important Note

All cryptographic parameters follow NIST recommendations and industry best practices. The 100,000 PBKDF2 iterations provide strong protection against brute-force attacks while maintaining reasonable performance.

Parameter	Value/Algorithm
Encryption Algorithm	AES-256-CBC
Key Size	256 bits (32 bytes)
Block Size	128 bits (16 bytes)
IV Size	128 bits (16 bytes)
Salt Size	128 bits (16 bytes)
Key Derivation	PBKDF2-HMAC-SHA256
KDF Iterations	100,000
Padding Scheme	PKCS#7
Random Number Generator	`os.urandom()` (CSPRNG)

Table 6: Secure Mode Cryptographic Parameters

7.3 Encryption Process

Algorithm 5 Secure Mode Encryption

Input: Message M, password P
Output: Encrypted data structure E

$salt \leftarrow \text{random\_bytes}(16)$
$iv \leftarrow \text{random\_bytes}(16)$
$key \leftarrow \text{PBKDF2}(P, salt, 100000, 32)$
$ciphertext \leftarrow \text{AES256\_CBC\_Encrypt}(M, key, iv)$

$E.salt \leftarrow salt$
$E.iv \leftarrow iv$
$E.ciphertext \leftarrow ciphertext$

return E

7.4 Serialization Format

Encrypted data is serialized before DNA encoding:

Offset	Length	Field
0-1	2 bytes	Salt length (big-endian uint16)
2-17	16 bytes	Salt
18-19	2 bytes	IV length (big-endian uint16)
20-35	16 bytes	IV
36-39	4 bytes	Ciphertext length (big-endian uint32)
40+	Variable	Ciphertext

Table 7: Secure Mode Serialization Format

```
import struct

def serialize_crypto_data(salt, iv, ciphertext):
    """
    Serialize cryptographic data for DNA encoding.
```

```
    Returns bytes in format:
    [salt_len(2)][salt(16)][iv_len(2)][iv(16)][ct_len(4)][ciphertext(n)]
    """
    salt_len = struct.pack('>H', len(salt))       # 2-byte big-endian
    iv_len = struct.pack('>H', len(iv))           # 2-byte big-endian
    ct_len = struct.pack('>I', len(ciphertext))   # 4-byte big-endian

    return salt_len + salt + iv_len + iv + ct_len + ciphertext

def deserialize_crypto_data(data):
    """
    Deserialize cryptographic data from bytes.

    Returns (salt, iv, ciphertext)
    """
    offset = 0

    # Extract salt
    salt_len = struct.unpack('>H', data[offset:offset+2])[0]
    offset += 2
    salt = data[offset:offset+salt_len]
    offset += salt_len

    # Extract IV
    iv_len = struct.unpack('>H', data[offset:offset+2])[0]
    offset += 2
    iv = data[offset:offset+iv_len]
    offset += iv_len

    # Extract ciphertext
    ct_len = struct.unpack('>I', data[offset:offset+4])[0]
    offset += 4
    ciphertext = data[offset:offset+ct_len]

    return salt, iv, ciphertext
```

Listing 9: Crypto Data Serialization

7.5 Complete Secure Encoding Algorithm

```
from Crypto.Cipher import AES
from Crypto.Protocol.KDF import PBKDF2
from Crypto.Random import get_random_bytes
import base64

def encode_secure(message: str, password: str) -> str:
    """
    Encode message using secure mode with AES-256-CBC.

    Args:
        message: Plaintext message
        password: Encryption password

    Returns:
```

```
        DNA sequence containing encrypted data
    """
    # Step 1: Generate cryptographic materials
    salt = get_random_bytes(16)
    iv = get_random_bytes(16)

    # Step 2: Derive key from password
    key = PBKDF2(
        password.encode('utf-8'),
        salt,
        dkLen=32,         # 256 bits
        count=100000,     # Iterations
        hmac_hash_module=None  # Default SHA-256
    )

    # Step 3: Encrypt message
    cipher = AES.new(key, AES.MODE_CBC, iv)
    padded_message = pad_pkcs7(message.encode('utf-8'))
    ciphertext = cipher.encrypt(padded_message)

    # Step 4: Serialize crypto data
    crypto_data = serialize_crypto_data(salt, iv, ciphertext)

    # Step 5: Base64 encode
    crypto_string = base64.b64encode(crypto_data).decode('ascii')

    # Step 6: DNA encode using basic mode
    dna = encode_basic(crypto_string)

    # Step 7: Add markers
    dna = "ATCGATCG" + dna + "CGATATCG"

    return dna

def pad_pkcs7(data: bytes) -> bytes:
    """Apply PKCS#7 padding."""
    padding_length = 16 - (len(data) % 16)
    padding = bytes([padding_length] * padding_length)
    return data + padding
```

Listing 10: Secure Mode Encoding

7.6 Complete Secure Decoding Algorithm

```
def decode_secure(dna: str, password: str) -> str:
    """
    Decode secure mode DNA sequence.

    Args:
        dna: DNA sequence with encrypted data
        password: Decryption password

    Returns:
        Decrypted plaintext message
    """
    # Step 1: Remove markers
```

```
    if dna.startswith("ATCGATCG"):
        dna = dna[8:]
    if dna.endswith("CGATATCG"):
        dna = dna[:-8]

    # Step 2: DNA decode using basic mode
    crypto_string = decode_basic(dna)

    # Step 3: Base64 decode
    crypto_data = base64.b64decode(crypto_string)

    # Step 4: Deserialize crypto data
    salt, iv, ciphertext = deserialize_crypto_data(crypto_data)

    # Step 5: Derive key from password
    key = PBKDF2(
        password.encode('utf-8'),
        salt,
        dkLen=32,
        count=100000
    )

    # Step 6: Decrypt
    cipher = AES.new(key, AES.MODE_CBC, iv)
    padded_plaintext = cipher.decrypt(ciphertext)

    # Step 7: Remove PKCS#7 padding
    plaintext = unpad_pkcs7(padded_plaintext)

    return plaintext.decode('utf-8')

def unpad_pkcs7(data: bytes) -> bytes:
    """Remove PKCS#7 padding."""
    padding_length = data[-1]
    return data[:-padding_length]
```

Listing 11: Secure Mode Decoding

7.7 Password Requirements

Warning

Strong passwords are critical for security. Weak passwords can be brute-forced despite strong encryption.

Recommended password policy:

- **Minimum Length**: 12 characters (16+ recommended)
- **Character Classes**: Uppercase, lowercase, digits, special characters
- **Avoid**: Dictionary words, common patterns, personal information
- **Entropy**: Minimum 60 bits (80+ recommended)

```python
import re

def validate_password_strength(password: str) -> tuple[bool, str]:
    """
    Validate password meets security requirements.

    Returns (is_valid, message)
    """
    if len(password) < 12:
        return False, "Password must be at least 12 characters"

    checks = {
        'uppercase': r'[A-Z]',
        'lowercase': r'[a-z]',
        'digit': r'[0-9]',
        'special': r'[^A-Za-z0-9]'
    }

    for name, pattern in checks.items():
        if not re.search(pattern, password):
            return False, f"Password must contain {name} character"

    # Check for common patterns
    common_patterns = ['12345', 'password', 'qwerty', 'abc']
    password_lower = password.lower()
    for pattern in common_patterns:
        if pattern in password_lower:
            return False, "Password contains common pattern"

    return True, "Password meets requirements"
```

Listing 12: Password Strength Validation

7.8 Security Considerations

7.8.1 Key Derivation

PBKDF2 with 100,000 iterations provides:

- Protection against dictionary attacks
- Computational cost for attackers
- Reasonable performance (¡ 1 second on modern hardware)

7.8.2 Randomness

All random values (salt, IV) use cryptographically secure random number generator:

```python
import os
salt = os.urandom(16)  # Uses /dev/urandom on Unix
```

7.8.3 Data Integrity

While AES-CBC provides confidentiality, it does not provide authentication. Future versions may include HMAC for integrity verification.

7.9 Format Specification

Secure Mode Format

`ATCGATCG [Base64-encoded encrypted data in DNA] CGATATCG`

Components:

- Start marker: `ATCGATCG` (always present)
- Core sequence: Encrypted data (salt + IV + ciphertext)
- Stop marker: `CGATATCG` (always present)

7.10 Performance Characteristics

- **Encryption Time**: $O(n + k)$ where k is KDF cost (dominant)
- **Decryption Time**: $O(n + k)$ where k is KDF cost (dominant)
- **Memory Usage**: $O(n)$ for plaintext/ciphertext
- **Storage Overhead**: 40 bytes (salt + IV + padding) + Base64 expansion (33%)
- **KDF Time**: 500ms-1s per encode/decode on modern CPU

Part III
Implementation and Validation

8 Error Handling

8.1 Error Taxonomy

BioCypher defines standardized error codes for interoperability:

Error Code	Severity	Description
INVALID_SEQUENCE	Error	DNA sequence contains invalid bases
MISSING_MARKERS	Error	Required markers not found
DECODE_FAILED	Error	Unable to decode sequence
ENCRYPTION_ERROR	Error	Cryptographic operation failed
PASSWORD_REQUIRED	Error	Secure mode requires password
PASSWORD_WEAK	Warning	Password doesn't meet strength requirements
CHECKSUM_MISMATCH	Error	Parity or integrity check failed
GC_IMBALANCE	Warning	GC content outside optimal range
HOMOPOLYMER_DETECTED	Warning	Homopolymer runs detected
SEQUENCE_TOO_SHORT	Error	Sequence length below minimum

Table 8: BioCypher Error Codes

8.2 Error Handling Strategies

8.2.1 Recoverable Errors

Some errors can be handled gracefully:

```
def decode_with_error_handling(dna: str, mode: str) -> tuple[str, list]:
    """
    Decode DNA with comprehensive error handling.

    Returns (decoded_message, warnings)
    """
    warnings = []

    # Clean input
    dna = dna.upper().strip()
    dna = ''.join(c for c in dna if c in 'ATCG')

    # Check for invalid bases (already removed)
    if len(dna) == 0:
        raise ValueError("INVALID_SEQUENCE: No valid DNA bases")

    # Mode-specific validation
    if mode in ['nanopore', 'secure']:
        if not (dna.startswith("ATCGATCG") and dna.endswith("CGATATCG")):
            warnings.append("MISSING_MARKERS: Markers not found, attempting decode anyway")
```

```
21
22      # Check GC content
23      gc_content = calculate_gc_content(dna)
24      if not (20 <= gc_content <= 80):
25          warnings.append(f"GC_IMBALANCE: GC content {gc_content:.1f}%
        outside range")
26
27      # Attempt decode
28      try:
29          if mode == 'basic':
30              message = decode_basic(dna)
31          elif mode == 'nanopore':
32              message = decode_nanopore(dna)
33          elif mode == 'secure':
34              message = decode_secure(dna, password)
35          else:
36              raise ValueError(f"Unknown mode: {mode}")
37
38          return message, warnings
39
40      except Exception as e:
41          raise ValueError(f"DECODE_FAILED: {str(e)}")
```

Listing 13: Recoverable Error Handling

8.2.2 Cryptographic Errors

Cryptographic errors must be handled carefully to avoid leaking information:

```
1  def decode_secure_safe(dna: str, password: str) -> str:
2      """
3      Decode secure mode with safe error handling.
4
5      Never reveals whether password is correct or data is corrupt.
6      """
7      try:
8          return decode_secure(dna, password)
9      except Exception as e:
10         # Generic error - don't leak info about failure cause
11         raise ValueError("DECRYPTION_FAILED: Unable to decrypt data")
```

Listing 14: Secure Error Handling

Warning

Never expose detailed error messages for cryptographic failures. Generic errors prevent information leakage to attackers.

9 Validation and Testing

9.1 Sequence Validation

9.1.1 DNA Base Validation

```python
def validate_dna_sequence(dna: str) -> tuple[bool, list]:
    """
    Validate DNA sequence.

    Returns (is_valid, errors)
    """
    errors = []

    # Check for empty sequence
    if len(dna) == 0:
        errors.append("Empty sequence")
        return False, errors

    # Check for invalid characters
    valid_bases = set('ATCGatcg')
    invalid_chars = set(dna) - valid_bases
    if invalid_chars:
        errors.append(f"Invalid bases: {invalid_chars}")

    # Check length constraints
    if len(dna) > 1000000:  # 1M bases
        errors.append("Sequence exceeds maximum length")

    return len(errors) == 0, errors
```

Listing 15: DNA Sequence Validation

9.1.2 Nanopore-Specific Validation

```python
def validate_nanopore_sequence(dna: str) -> tuple[bool, list]:
    """
    Validate sequence for nanopore sequencing.

    Returns (is_valid, warnings)
    """
    warnings = []

    # Check markers
    if not dna.startswith("ATCGATCG"):
        warnings.append("Missing start marker")
    if not dna.endswith("CGATATCG"):
        warnings.append("Missing stop marker")

    # Check GC content
    gc = calculate_gc_content(dna)
    if not (40 <= gc <= 60):
        warnings.append(f"GC content {gc:.1f}% outside optimal range")

    # Check homopolymers
    if has_homopolymers(dna):
        warnings.append("Homopolymer runs detected")

    # Check length alignment
    core = dna[8:-8] if len(dna) > 16 else dna
    if len(core) % 3 != 0:
```

```
        warnings.append("Sequence not aligned to triplet boundary")

    return True, warnings  # Warnings don't prevent use
```

Listing 16: Nanopore Validation

9.2 Test Vectors

9.2.1 Basic Mode Test Vectors

Input	Binary	Expected DNA
"A"	01000001	TAAA
"B"	01000010	TAAAC
"Hi"	01001000 01101001	TAAATAAATATA
" " (space)	00100000	ACAA
"123"	00110001 00110010 00110011	AGCAAT AGCATC AGCATG

Table 9: Basic Mode Test Vectors

9.2.2 Nanopore Mode Test Vectors

Due to randomized padding and optimization, nanopore mode test vectors focus on structure:

```
def test_nanopore_roundtrip():
    """Test nanopore encoding/decoding roundtrip."""
    test_cases = [
        "Hello",
        "BioCypher Protocol v1.0",
        "The quick brown fox jumps over the lazy dog",
        "!@#$%^&*()_+-=[]{}|;':,.<>?/"  # Special chars
    ]

    for message in test_cases:
        encoded = encode_nanopore(message, error_correction=True)

        # Verify structure
        assert encoded.startswith("ATCGATCG"), "Missing start marker"
        assert encoded.endswith("CGATATCG"), "Missing stop marker"

        # Verify no homopolymers
        assert not has_homopolymers(encoded), "Homopolymers detected"

        # Verify GC content
        gc = calculate_gc_content(encoded)
        assert 20 <= gc <= 80, f"GC content {gc} out of range"

        # Roundtrip decode
        decoded = decode_nanopore(encoded, error_correction=True)
        assert decoded == message, f"Roundtrip failed: {message} != {
    decoded}"
```

Listing 17: Nanopore Test Vector Validation

9.2.3 Secure Mode Test Vectors

Secure mode uses randomized salt/IV, so test vectors verify structure and properties:

```python
def test_secure_mode():
    """Test secure mode encryption/decryption."""
    message = "Top Secret Message"
    password = "StrongP@ssw0rd123!"

    # Encode
    encoded = encode_secure(message, password)

    # Verify structure
    assert encoded.startswith("ATCGATCG"), "Missing start marker"
    assert encoded.endswith("CGATATCG"), "Missing stop marker"

    # Verify DNA validity
    core = encoded[8:-8]
    assert all(c in 'ATCG' for c in core), "Invalid DNA bases"

    # Decode with correct password
    decoded = decode_secure(encoded, password)
    assert decoded == message, "Decryption failed"

    # Verify wrong password fails
    try:
        wrong_decode = decode_secure(encoded, "WrongPassword")
        assert False, "Should have failed with wrong password"
    except:
        pass  # Expected failure
```

Listing 18: Secure Mode Test

9.3 Unit Test Suite

A complete implementation should include:

1. **Encoding Tests**:
   - ASCII character coverage (32-126)
   - Extended characters and Unicode handling
   - Empty strings and edge cases
   - Maximum length sequences
2. **Decoding Tests**:
   - Valid sequences
   - Sequences with invalid bases
   - Truncated sequences
   - Sequences without markers
3. **Roundtrip Tests**:
   - Encode then decode yields original
   - All three modes

- Various message lengths

4. **Error Correction Tests**:
   - Introduce single-bit errors
   - Verify correction in nanopore mode
   - Test correction limits (2+ bit errors)
5. **Cryptographic Tests**:
   - Password strength validation
   - Encryption/decryption roundtrip
   - Wrong password detection
   - Salt/IV uniqueness
6. **Performance Tests**:
   - Encoding speed benchmarks
   - Decoding speed benchmarks
   - Memory usage profiling

10 Conformance Levels

The BioCypher protocol defines four conformance levels for implementations:

10.1 Level 1: Basic Conformance

Required Features:

- Support Basic Mode encoding
- Support Basic Mode decoding
- Implement standard binary-to-DNA mapping
- Handle ASCII text (printable range 32-126)
- Skip invalid DNA bases during decoding

Validation:

- Pass all basic mode test vectors
- Roundtrip encode/decode correctly

10.2 Level 2: Nanopore Conformance

Required Features:

- All Level 1 requirements
- Support Nanopore Mode encoding
- Support Nanopore Mode decoding

- Implement triplet encoding table
- Support error correction (triple redundancy)
- Handle nanopore markers
- Support padding removal
- Parity bit checking

Validation:

- Pass nanopore mode roundtrip tests
- Verify homopolymer avoidance
- Demonstrate error correction

10.3 Level 3: Secure Conformance

Required Features:

- All Level 1 requirements
- Support Secure Mode encoding
- Support Secure Mode decoding
- Implement AES-256-CBC encryption
- Implement PBKDF2-HMAC-SHA256 (100,000 iterations)
- Password strength validation
- Handle secure mode markers
- Proper cryptographic error handling

Validation:

- Pass secure mode roundtrip tests
- Verify encryption with test vectors
- Demonstrate wrong password detection

10.4 Level 4: Full Conformance

Required Features:

- All Level 1, 2, and 3 requirements
- Comprehensive error handling
- Complete validation suite
- Performance optimization
- Statistics and analysis

- Documentation compliance

Validation:

- Pass all test vectors
- Complete unit test suite (¿90% coverage)
- Performance benchmarks meet standards

Part IV
Practical Application and Deployment

11 Installation and Setup

11.1 Reference Implementation

The official BioCypher reference implementation is available at:

https://github.com/syndicate-labs/biocipher

11.2 Installation Methods

11.2.1 From Source

```
# Clone repository
git clone https://github.com/syndicate-labs/biocipher.git
cd biocipher

# Install dependencies
pip install -r requirements.txt

# Install BioCypher
pip install -e .

# Verify installation
biocipher --version
```

Listing 19: Installation from Source

11.2.2 Using pip

```
pip install biocipher

# Verify
biocipher --version
```

Listing 20: Installation via pip

11.3 Dependencies

Package	Version	Purpose
Python	3.8+	Runtime environment
pycryptodome	3.15+	AES encryption
numpy	1.20+	Numerical operations
click	8.0+	CLI interface

Table 10: BioCypher Dependencies

12 Command-Line Interface

12.1 Basic Usage

12.1.1 Encoding

```
# Basic mode
biocipher encode --mode basic --input message.txt --output sequence.
    fasta

# Nanopore mode with error correction
biocipher encode --mode nanopore --error-correction \
    --input message.txt --output sequence.fasta

# Secure mode with password
biocipher encode --mode secure --password "MyPassword123!" \
    --input secret.txt --output encrypted.fasta

# Pipe input
echo "Hello World" | biocipher encode --mode basic
```

Listing 21: CLI Encoding Examples

12.1.2 Decoding

```
# Basic mode
biocipher decode --mode basic --input sequence.fasta

# Nanopore mode
biocipher decode --mode nanopore --error-correction \
    --input sequence.fasta --output decoded.txt

# Secure mode
biocipher decode --mode secure --password "MyPassword123!" \
    --input encrypted.fasta --output secret.txt
```

Listing 22: CLI Decoding Examples

12.2 Analysis and Validation

```
# Validate DNA sequence
biocipher validate --input sequence.fasta

# Analyze GC content and properties
biocipher analyze --input sequence.fasta

# Check for homopolymers
biocipher check-homopolymers --input sequence.fasta

# Calculate statistics
biocipher stats --input sequence.fasta
```

Listing 23: Analysis Commands

13 Python API

13.1 Basic Usage

```python
from biocipher import encode, decode

# Basic encoding
message = "Hello, BioCypher!"
dna = encode(message, mode='basic')
print(f"DNA: {dna}")

# Basic decoding
decoded = decode(dna, mode='basic')
print(f"Decoded: {decoded}")

# Nanopore encoding with error correction
dna_nanopore = encode(message, mode='nanopore', error_correction=True)

# Secure encoding
dna_secure = encode(message, mode='secure', password='MyPassword!')
decoded_secure = decode(dna_secure, mode='secure', password='MyPassword
    !')
```

Listing 24: Python API Examples

13.2 Advanced API Usage

```python
from biocipher import BioCypher
from biocipher.validators import validate_sequence
from biocipher.analyzers import analyze_sequence

# Create BioCypher instance
bc = BioCypher()

# Encode with custom settings
dna = bc.encode(
    message="Secret data",
    mode='nanopore',
    error_correction=True,
    optimize_gc=True,
    redundancy=5
)

# Validate sequence
is_valid, errors = validate_sequence(dna, mode='nanopore')
if not is_valid:
    print(f"Validation errors: {errors}")

# Analyze sequence
stats = analyze_sequence(dna)
print(f"GC Content: {stats['gc_content']:.1f}%")
print(f"Length: {stats['length']} bases")
print(f"Homopolymers: {stats['homopolymer_count']}")

# Decode with error handling
try:
```

```
    decoded = bc.decode(dna, mode='nanopore', error_correction=True)
    print(f"Successfully decoded: {decoded}")
except Exception as e:
    print(f"Decoding failed: {e}")
```

Listing 25: Advanced API

14 Integration with DNA Synthesis

14.1 Synthesis Workflow

1. **Encode Message**: Use BioCypher to generate DNA sequence
2. **Validate Sequence**: Check for synthesis compatibility
3. **Format for Synthesis**: Export in FASTA or vendor-specific format
4. **Submit to Vendor**: Send to DNA synthesis service
5. **Receive Synthesized DNA**: Obtain physical DNA (plasmid, oligo, etc.)

14.2 Synthesis Provider Integration

```
from biocipher import BioCypher
from biocipher.export import export_fasta, export_genbank

bc = BioCypher()
dna = bc.encode("Important data", mode='nanopore')

# Export as FASTA for synthesis
fasta = export_fasta(
    sequence=dna,
    name="BioCypher_Message_001",
    description="BioCypher encoded message"
)

with open('for_synthesis.fasta', 'w') as f:
    f.write(fasta)

# Example FASTA output:
# >BioCypher_Message_001 BioCypher encoded message
# ATCGATCGATCATGACTACGTAGATCATGACTACGATCATGACTACG
# ATCATGACTACGATCATGACTACGCGATATCG
```

Listing 26: Export for Synthesis

14.3 Recommended Synthesis Providers

15 Integration with Nanopore Sequencing

15.1 Oxford Nanopore MinION Workflow

1. **DNA Extraction**: Extract DNA from carrier using DNeasy or similar kit
2. **Quality Control**: Quantify DNA (NanoDrop/Qubit), aim for 1-5 µg

Provider	Max Length	Turnaround	Cost/base
IDT	3 kb	5-10 days	$0.07-0.10
Twist Bioscience	10 kb	7-14 days	$0.07-0.09
GenScript	5 kb	10-15 days	$0.09-0.12
Eurofins	2 kb	5-10 days	$0.08-0.11

Table 11: DNA Synthesis Providers (approximate as of 2025)

3. **Library Preparation**: Use Rapid Sequencing Kit (10 min) or Ligation Kit (1 hr)
4. **Load Flow Cell**: Prime MinION flow cell, load library
5. **Run Sequencing**: Start run via MinKNOW software
6. **Real-Time Basecalling**: Monitor for target sequences
7. **Extract FASTQ**: Export sequenced reads
8. **Decode**: Use BioCypher to decode DNA sequences

15.2 MinKNOW Integration

```python
from biocipher import BioCypher
from Bio import SeqIO  # Biopython for FASTQ parsing

bc = BioCypher()

# Parse FASTQ file from MinKNOW
fastq_file = "sequencing_run_001.fastq"
found_sequences = []

for record in SeqIO.parse(fastq_file, "fastq"):
    sequence = str(record.seq)

    # Check if sequence has BioCypher markers
    if "ATCGATCG" in sequence and "CGATATCG" in sequence:
        # Extract BioCypher sequence
        start = sequence.find("ATCGATCG")
        end = sequence.find("CGATATCG", start) + 8
        biocipher_seq = sequence[start:end]

        # Attempt decode
        try:
            message = bc.decode(
                biocipher_seq,
                mode='nanopore',
                error_correction=True
            )
            found_sequences.append({
                'read_id': record.id,
                'sequence': biocipher_seq,
                'message': message
            })
            print(f"Decoded from {record.id}: {message}")
        except Exception as e:
```

```
            print(f"Failed to decode {record.id}: {e}")

# Save results
import json
with open('decoded_messages.json', 'w') as f:
    json.dump(found_sequences, f, indent=2)
```

Listing 27: Process MinKNOW Output

16 Physical Transport Methods

16.1 Bacterial Carrier Protocol

Bacterial Spore Transport

Materials:

- *Bacillus subtilis* 168 (non-pathogenic)
- High-copy plasmid with BioCypher sequence
- Sporulation medium
- Sterile vials or filter paper

Protocol:

1. Transform bacteria with plasmid
2. Culture 24h at 37°C
3. Induce sporulation (48-72h)
4. Heat-kill vegetative cells (80°C, 20 min)
5. Lyophilize spores or air-dry on filter paper
6. Package for transport

Recovery:

1. Germinate spores in LB medium (2-4h)
2. Extract plasmid DNA
3. Sequence using MinION
4. Decode with BioCypher

16.2 Plant Seed Carrier Protocol

Plant Seed Transport

Materials:

- *Arabidopsis thaliana* or tobacco
- *Agrobacterium* with BioCypher sequence in T-DNA
- Growth facilities

Protocol:

1. Floral dip transformation
2. Grow T1 generation plants
3. Select transformants
4. Collect seeds (stable integration)
5. Package seeds for transport

Recovery:

1. Germinate seeds
2. Extract genomic DNA
3. PCR amplify BioCypher region
4. Sequence amplicon
5. Decode with BioCypher

17 Security Best Practices

17.1 Operational Security

1. **Password Management**:
   - Use password manager for secure mode passwords
   - Never store passwords in plain text
   - Use unique passwords per message
   - Consider passphrase generation (diceware)
2. **Data Handling**:
   - Securely delete plaintext after encoding
   - Wipe temporary files
   - Use encrypted filesystems when possible
   - Clear terminal history if sensitive
3. **Physical Security**:

- Protect synthesized DNA samples
- Secure transport containers
- Destroy carriers after decoding if sensitive
- Consider tamper-evident packaging

4. **Operational Practices**:
   - Verify DNA sequence integrity before synthesis
   - Use multiple redundant encodings for critical data
   - Maintain operational security during transport
   - Document chain of custody if needed

17.2 Threat Model

17.2.1 Adversary Capabilities

BioCypher secure mode protects against:

- **Passive Surveillance**: Interception of DNA carriers
- **Sequence Analysis**: Adversary can sequence DNA
- **Computational Attacks**: Brute-force and dictionary attacks

BioCypher does NOT protect against:

- **Coercion**: Physical threats to reveal password
- **Endpoint Compromise**: Malware on encoding/decoding system
- **Side-Channel Attacks**: Timing, power analysis, etc.
- **Quantum Computers**: AES-256 may be vulnerable to Grover's algorithm

17.3 Future Security Enhancements

Proposed for future protocol versions:

- **Authenticated Encryption**: Add HMAC for integrity verification
- **Post-Quantum Cryptography**: Integrate lattice-based or hash-based algorithms
- **Multi-Party Encryption**: Support for threshold cryptography
- **Steganographic Improvements**: Better hiding in natural sequences

Part V
Protocol Extensions and Development

18 Extending the Protocol

18.1 Community Extension Process

BioCypher is an open protocol designed for community extension. The formal process:

1. **Proposal Submission**:
   - Submit BioCypher Improvement Proposal (BIP)
   - Include rationale, specification, and test vectors
   - Post to GitHub discussions or mailing list
2. **Community Review**:
   - Open discussion period (minimum 30 days)
   - Technical review by maintainers
   - Public comment and feedback
3. **Implementation**:
   - Reference implementation in official codebase
   - Comprehensive test suite
   - Documentation updates
4. **Adoption**:
   - Merged into next protocol version
   - Announced to community
   - Backward compatibility maintained when possible

18.2 Example Extensions

18.2.1 Compression Extension

Add data compression before encoding:

```
import zlib
import base64

def encode_compressed(message: str, mode: str = 'basic') -> str:
    """Encode with compression for better storage efficiency."""
    # Compress
    compressed = zlib.compress(message.encode('utf-8'), level=9)

    # Base64 encode (for DNA encoding)
    b64 = base64.b64encode(compressed).decode('ascii')

    # DNA encode
    dna = encode(b64, mode=mode)

```

```
    # Add compression marker
    return "ATCGATCG_COMPRESSED_" + dna

def decode_compressed(dna: str, mode: str = 'basic') -> str:
    """Decode compressed sequence."""
    # Remove compression marker
    if dna.startswith("ATCGATCG_COMPRESSED_"):
        dna = dna[20:]

    # DNA decode
    b64 = decode(dna, mode=mode)

    # Base64 decode
    compressed = base64.b64decode(b64)

    # Decompress
    message = zlib.decompress(compressed).decode('utf-8')

    return message
```

Listing 28: Compression Extension Example

18.2.2 Checksum Extension

Add CRC32 checksums for integrity:

```
import struct
import binascii

def encode_with_checksum(message: str, mode: str = 'basic') -> str:
    """Add CRC32 checksum for integrity verification."""
    # Calculate checksum
    checksum = binascii.crc32(message.encode('utf-8'))

    # Prepend checksum (4 bytes)
    data = struct.pack('>I', checksum) + message.encode('utf-8')

    # Encode to DNA
    return encode(data.decode('latin-1'), mode=mode)

def decode_with_checksum(dna: str, mode: str = 'basic') -> str:
    """Decode and verify checksum."""
    # Decode
    data = decode(dna, mode=mode).encode('latin-1')

    # Extract checksum and message
    checksum_stored = struct.unpack('>I', data[:4])[0]
    message_bytes = data[4:]

    # Verify
    checksum_calculated = binascii.crc32(message_bytes)
    if checksum_stored != checksum_calculated:
        raise ValueError("CHECKSUM_MISMATCH")

    return message_bytes.decode('utf-8')
```

Listing 29: Checksum Extension

19 Contributing to BioCypher

19.1 Development Setup

```
# Clone repository
git clone https://github.com/syndicate-labs/biocipher.git
cd biocipher

# Create virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

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

# Install pre-commit hooks
pre-commit install

# Run tests
pytest tests/ -v

# Check code coverage
pytest --cov=biocipher --cov-report=html

# Run linter
flake8 biocipher/

# Format code
black biocipher/
```

Listing 30: Development Environment Setup

19.2 Contribution Guidelines

1. **Code Quality**:
   - Follow PEP 8 style guide
   - Maintain test coverage ¿90%
   - Add docstrings to all public functions
   - Type hints for function signatures
2. **Testing**:
   - Write unit tests for new features
   - Include integration tests
   - Add test vectors for new encoding modes
   - Test edge cases and error conditions
3. **Documentation**:
   - Update README for user-facing changes
   - Add technical documentation for new features
   - Include examples and use cases

- Update protocol specification if needed

4. **Pull Request Process**:
   - Create feature branch from `main`
   - Write clear commit messages
   - Reference related issues
   - Request review from maintainers

20 Performance Optimization

20.1 Profiling

```python
import cProfile
import pstats
from biocipher import encode, decode

# Profile encoding
def profile_encoding():
    message = "A" * 10000  # 10KB message
    for _ in range(100):
        encode(message, mode='nanopore', error_correction=True)

# Run profiler
cProfile.run('profile_encoding()', 'encode_stats')

# Analyze results
p = pstats.Stats('encode_stats')
p.sort_stats('cumulative')
p.print_stats(10)  # Top 10 functions
```

Listing 31: Performance Profiling

20.2 Optimization Strategies

1. **Algorithmic Improvements**:
   - Use lookup tables for encoding/decoding
   - Batch process sequences
   - Parallelize independent operations

2. **Data Structure Optimization**:
   - Use bytearray for mutable binary data
   - Preallocate buffers when size is known
   - Avoid string concatenation in loops

3. **Caching**:
   - Cache encryption keys (PBKDF2 is expensive)
   - Memoize frequently computed values
   - Use LRU cache for pure functions

```python
from functools import lru_cache

# Cache triplet mapping for faster lookups
@lru_cache(maxsize=8)
def get_triplet_map():
    return {
        '000': 'ATC', '001': 'ATG', '010': 'ACT', '011': 'ACG',
        '100': 'TAG', '101': 'TAC', '110': 'TCG', '111': 'TCA'
    }

def encode_nanopore_optimized(message: str) -> str:
    """Optimized nanopore encoding."""
    # Preallocate buffer
    binary_length = len(message) * 9  # 9 bits per char
    binary = bytearray(binary_length)

    # Use cached triplet map
    triplet_map = get_triplet_map()

    # Build binary representation
    offset = 0
    for char in message:
        byte = format(ord(char), '08b')
        parity = str(byte.count('1') % 2)
        chunk = byte + parity

        for bit in chunk:
            binary[offset] = ord(bit)
            offset += 1

    # Convert to DNA using list comprehension
    dna_parts = [
        triplet_map[binary[i:i+3].decode('ascii')]
        for i in range(0, len(binary), 3)
    ]

    return ''.join(dna_parts)
```

Listing 32: Optimized Encoding

21 Reference Documentation

21.1 API Documentation

Complete API documentation is generated from source code docstrings using Sphinx:

```bash
# Install Sphinx
pip install sphinx sphinx-rtd-theme

# Build HTML documentation
cd docs/
make html

# View documentation
open _build/html/index.html
```

Listing 33: Build Documentation

21.2 Protocol Specification Updates

This document represents Protocol Version 1.0. Updates follow semantic versioning:

- **Major Version**: Breaking changes to encoding format
- **Minor Version**: New features, backward compatible
- **Patch Version**: Bug fixes, clarifications

22 Roadmap

22.1 Short-Term (6 months)

- Implement compression extension
- Add integrity checking (HMAC)
- Improve nanopore optimization algorithms
- Expand test vector coverage
- Performance optimizations

22.2 Medium-Term (12 months)

- Post-quantum cryptography integration
- Multi-mode hybrid encoding
- Advanced steganography features
- Web-based encoder/decoder
- Mobile applications (iOS/Android)

22.3 Long-Term (24+ months)

- Hardware integration (MinION direct support)
- Distributed storage protocol
- Blockchain-based verification
- AI-powered sequence optimization
- Space mission integration

A Glossary

AES Advanced Encryption Standard - Symmetric encryption algorithm

Base64 Binary-to-text encoding scheme

Basecalling Converting nanopore current signals to DNA sequences

Flow Cell Consumable containing nanopores for sequencing

GC Content Percentage of guanine and cytosine bases

Homopolymer Run of consecutive identical nucleotides

PBKDF2 Password-Based Key Derivation Function 2

Triplet Three-nucleotide sequence

B Mathematical Notation

Symbol	Meaning
M	Message (plaintext)
D	DNA sequence
B	Binary representation
E	Encrypted data
P	Password
n	Message length (characters)
m	DNA sequence length (bases)
$O(n)$	Big-O complexity notation

C References

1. NIST FIPS 197: "Advanced Encryption Standard (AES)"
2. RFC 2898: "PKCS #5: Password-Based Cryptography Specification Version 2.0"
3. RFC 4648: "The Base16, Base32, and Base64 Data Encodings"
4. Oxford Nanopore Technologies: "Nanopore Sequencing Accuracy"
5. Church et al. (2012): "Next-Generation Digital Information Storage in DNA", *Science*
6. Goldman et al. (2013): "Towards practical, high-capacity, low-maintenance information storage in synthesized DNA", *Nature*

D License

E Contact Information

Resource	Location
Project Website	https://syndicate-labs.io/biocipher
GitHub Repository	https://github.com/syndicate-labs/biocipher
Documentation	https://docs.syndicate-labs.io
Email	contact@syndicate-labs.io
Matrix Chat	#biocypher:matrix.org

F Acknowledgments

BioCypher builds upon decades of research in DNA data storage, cryptography, and nanopore sequencing. We acknowledge:

- The DNA data storage research community
- Oxford Nanopore Technologies for sequencing innovation
- The open-source cryptography community
- All contributors to the BioCypher project

BioCypher Technical Guide v1.0
Open Protocol for DNA-Based Data Encoding

Remember the Game Genie.