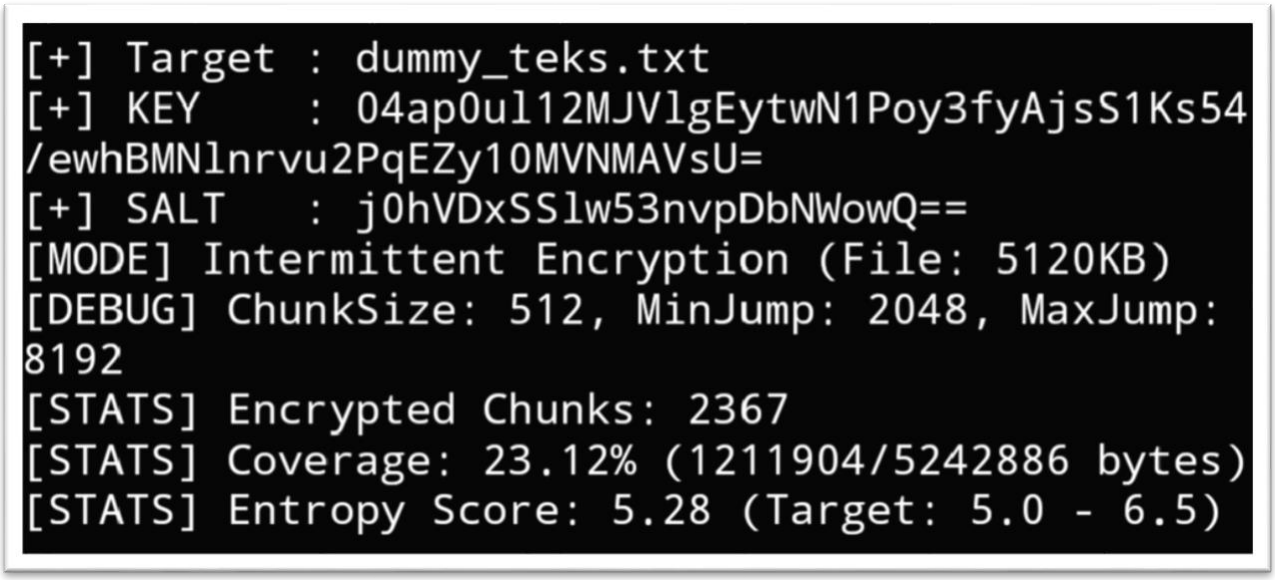
IMPLEMENTASI INTERMITTENT ENCRYPTION

MENGGUNAKAN SALSA20 CIPHER DAN MEMORY-MAPPED FILES UNTUK OPTIMASI PERFORMA ENKRIPSI FILE BESAR



DAFTAR ISI

📚 DOKUMENTASI LENGKAP: ULTIMATE HYBRID MMF ENCRYPTION ENGINE

📝 RINGKASAN PROYEK

Proyek ini adalah implementasi **Intermittent Encryption** menggunakan **Memory-Mapped Files (MMF)** dengan algoritma **Salsa20** sebagai cipher. Sistem ini dirancang untuk mengenkripsi file besar secara efisien dengan hanya mengenkripsi sebagian data (chunks) secara strategis, bukan seluruh file.

📖 BAB 1: INTERMITTENT ENCRYPTION

1.1 Apa itu Intermittent Encryption?

Intermittent Encryption (enkripsi berselang-seling) adalah teknik enkripsi di mana hanya sebagian data dari file yang dienkripsi, bukan keseluruhan file. Bayangkan seperti mewarnai buku: alih-alih mewarnai semua halaman, Anda hanya mewarnai halaman 1, 5, 10, 15, dan seterusnya dengan pola acak.

Mengapa teknik ini digunakan?

Kecepatan: Enkripsi seluruh file 10GB bisa memakan waktu lama. Dengan intermittent encryption, waktu enkripsi bisa dipangkas hingga 70-80% karena hanya mengenkripsi 20-30% dari total data.

Efektivitas: Meskipun tidak semua data terenkripsi, file tetap tidak bisa digunakan karena bagian-bagian penting (header, metadata, konten kritis) sudah terenkripsi. Misalnya, pada file video, jika header dan beberapa frame kunci terenkripsi, video tidak bisa diputar sama sekali.

Efisiensi Resource: Cocok untuk perangkat dengan resource terbatas (mobile, IoT) atau ketika harus mengenkripsi banyak file sekaligus.

1.2 Sejarah dan Penggunaan

Konsep intermittent encryption mulai populer di era **ransomware modern** (sekitar 2015-2020). Ransomware seperti **Cerber**, **Locky**, dan **REvil** menggunakan teknik ini untuk mengenkripsi file korban dengan cepat sebelum antivirus mendeteksi aktivitas mereka.

Studi Kasus Ransomware:

- **LockBit 3.0 (2022):** Menggunakan intermittent encryption dengan hanya mengenkripsi 4KB per setiap 64KB data. Hasilnya, enkripsi file 1GB hanya butuh beberapa detik.
- **BlackMatter (2021):** Mengenkripsi 50% dari file dengan pola acak, membuat recovery tanpa key menjadi hampir mustahil.

Penggunaan Legitimate:

Bukan hanya ransomware, teknik ini juga digunakan di:

- **Cloud Storage Encryption:** Google Drive dan Dropbox menggunakan partial encryption untuk file besar agar upload/download tetap cepat.
- **Database Encryption:** SQL Server mengenkripsi page-level data, bukan seluruh database, untuk performa lebih baik.
- **DRM (Digital Rights Management):** Melindungi konten video dengan mengenkripsi frame-frame kunci saja.

1.3 Keuntungan dan Risiko

Keuntungan:

- **Performa:** 5-10x lebih cepat dari full encryption
- **Resource-Friendly:** Konsumsi CPU dan memory lebih rendah
- **Praktis:** Cocok untuk enkripsi massal (ribuan file)

Risiko:

- **Partial Protection:** Bagian yang tidak terenkripsi bisa di-recover dengan forensic tools
- **Pattern Analysis:** Jika pola enkripsi bisa ditebak, attacker bisa merekonstruksi data
- **Tidak untuk Data Sensitif:** Tidak cocok untuk data yang butuh enkripsi penuh (medical records, financial data)

💾 BAB 2: MEMORY-MAPPED FILES (MMF)

2.1 Apa itu Memory-Mapped Files?

Memory-Mapped Files (MMF) adalah teknik di mana file di hard disk "dipetakan" (mapped) ke dalam memory (RAM) seolah-olah file tersebut adalah array biasa. Dengan MMF, Anda bisa mengakses file 10GB seperti mengakses array byte di memory, tanpa harus membaca seluruh file ke RAM.

Analogi Sederhana:

Bayangkan perpustakaan dengan jutaan buku. Alih-alih memindahkan semua buku ke meja Anda (seperti `File.ReadAllBytes()`), Anda punya "peta" yang langsung menunjuk ke rak mana buku itu berada. Anda hanya membaca buku yang Anda butuhkan saat itu juga.

2.2 Cara Kerja MMF

Proses Mapping:

1. Operating system membuat "virtual address space" yang menunjuk ke file di disk
2. Saat Anda akses posisi tertentu (misal byte ke-1000), OS secara otomatis load page tersebut ke RAM
3. Modifikasi langsung ter-sync ke file di disk (tidak perlu manual write)

Keuntungan untuk Enkripsi:

- **Efisiensi Memory:** File 5GB tidak perlu dimuat seluruhnya ke RAM. Hanya bagian yang sedang diproses saja yang dimuat (misalnya 512 bytes saat itu).
- **Random Access:** Bisa langsung "melompat" ke posisi 1.000.000 tanpa harus baca byte 0-999.999 terlebih dahulu.
- **Auto-Sync:** Perubahan langsung tersimpan ke disk, tidak perlu `File.WriteAllBytes()` yang boros memory.

2.3 Implementasi di Proyek Ini

Dalam proyek ini, MMF digunakan untuk:

Membaca Chunk: Langsung baca 512 bytes di posisi 5.000.000 tanpa load seluruh file.

Enkripsi In-Place: Data terenkripsi langsung ditulis kembali ke posisi yang sama, tanpa membuat file temporary.

Performa: Enkripsi file 5GB hanya butuh ~100MB RAM (untuk buffer processing), bukan 5GB!

Contoh kode:

```
using (var mmf = MemoryMappedFile.CreateFromFile(file, FileMode.Open)) {
    using (var accessor = mmf.CreateViewAccessor()) {
        // Langsung akses byte ke-5000
        accessor.ReadArray(5000, buffer, 0, 512);

        // Enkripsi chunk
        byte[] encrypted = Encrypt(buffer);
```

```
        // Tulis kembali ke posisi yang sama
        accessor.WriteArray(5000, encrypted, 0, 512);
    }
}
```

2.4 MMF vs Traditional File I/O

Traditional (File Stream):

- Harus baca file secara sequential (urut)
- Butuh buffer besar di memory
- Operasi write terpisah dari read
- Lambat untuk random access

Memory-Mapped Files:

- Random access tanpa penalty
- OS handle memory management
- Operasi read/write dalam satu operasi
- Ideal untuk intermittent encryption

BAB 3: RANDOM SEEDS DAN POLA LONCATAN

3.1 Mengapa Butuh Deterministik Random?

Dalam intermittent encryption, kita perlu "melompat" dari satu chunk ke chunk berikutnya dengan pola **acak tapi reproducible**. Artinya:

- Saat enkripsi, kita loncat dari posisi 0 → 5.000 → 12.000 → 20.000
- Saat dekripsi, kita **HARUS** loncat ke posisi yang **SAMA PERSIS**

Jika loncatan berbeda, dekripsi akan gagal karena kita dekripsi bagian yang salah!

3.2 Linear Congruential Generator (LCG)

Proyek ini menggunakan **LCG (Linear Congruential Generator)**, salah satu algoritma PRNG (Pseudo-Random Number Generator) tertua dan paling sederhana.

Formula:

```
X(n+1) = (a × X(n) + c) mod m
```

Dimana:

- `X(n)` = nilai random ke-n
- `a` = 1103515245 (multiplier)
- `c` = 12345 (increment)
- `m` = 2147483648 (modulus = 2^31)

Mengapa formula ini?

Nilai `a`, `c`, dan `m` dipilih berdasarkan **Hull-Dobell Theorem** untuk memastikan generator memiliki **full period** (menghasilkan semua nilai dari 0 hingga m-1 sebelum berulang).

Contoh Implementasi:

```
public class CustomRandom {
    private long _seed;

    public CustomRandom(int seed) {
        this._seed = seed;
    }

    public int Next(int min, int max) {
        _seed = (1103515245 * _seed + 12345) % 2147483648;
        return (int)(_seed % (max - min)) + min;
    }
}
```

3.3 Deterministic Seed Generation

Seed harus **unik per file** tapi **reproducible** saat dekripsi. Kita menggunakan kombinasi:

- **Encryption Key** (32 bytes)
- **Nama File** (misal: "dummy_teks.txt")
- **Ukuran File** (misal: 5242886 bytes)

Proses:

```
string raw = Convert.ToBase64String(key) + fileName + fileSize.ToString();
byte[] hash = SHA256.ComputeHash(Encoding.UTF8.GetBytes(raw));
int seed = BitConverter.ToInt32(hash, 0);
```

Mengapa SHA-256?

SHA-256 menghasilkan hash 256-bit dari input apapun. Kita ambil 4 bytes pertama (32-bit) sebagai seed integer. Hash ini:

- **Collision-resistant:** Dua input berbeda hampir mustahil menghasilkan hash sama
- **Deterministic:** Input sama selalu menghasilkan output sama

- **Uniform distribution:** Seed terdistribusi merata di seluruh range integer

3.4 Adaptive Jump Parameters

Sistem ini menggunakan **adaptive parameters** berdasarkan jenis dan ukuran file:

File Kecil (< 1MB):

- Mode: Full Encryption
- ChunkSize: Seluruh file
- Jump: 0 (tidak ada loncatan)

File Text (.txt, .log, .csv):

- ChunkSize: 512 bytes
- MinJump: 2KB
- MaxJump: 8KB
- Rasional: File text butuh enkripsi rapat karena data homogen

File Database (.db, .sqlite):

- ChunkSize: 4KB (sesuai page size database)
- MinJump: 100KB
- MaxJump: 1MB
- Rasional: Enkripsi per-page membuat database corrupt

File Video (.mp4, .mkv):

- ChunkSize: 256KB
- MinJump: 1MB
- MaxJump: 5MB
- Rasional: Video punya redundansi tinggi, enkripsi sparse cukup

Default (File Umum):

- ChunkSize: 128 bytes
- Jump: 1% dari ukuran file (min 1KB, max 500KB)

📊 BAB 4: ENTROPY DAN ANALISIS KUALITAS ENKRIPSI

4.1 Apa itu Entropy?

Entropy (dalam konteks informasi) adalah ukuran **ketidakpastian** atau **randomness** dari data. Semakin tinggi entropy, semakin acak dan unpredictable data tersebut.

Formula Shannon Entropy:

$$H(X) = -\Sigma\ p(xi) \times \log_2(p(xi))$$

Dimana:

- `H(X)` = entropy dalam bit
- `p(xi)` = probabilitas munculnya byte `i`
- Σ = sum dari semua kemungkinan byte (0-255)

Interpretasi:

- **Entropy = 0:** Data sepenuhnya predictable (misal: file berisi 1 juta huruf 'A')
- **Entropy = 8:** Data perfectly random (setiap byte punya probabilitas sama: 1/256)

4.2 Entropy dalam Enkripsi

Plaintext:

```
"SAYA SEDANG MEMBONGKAR RAHASIA NEGARA"
```

Analisis karakter:

- 'A' muncul 10 kali
- 'S' muncul 3 kali
- 'R' muncul 3 kali
- dst...

Entropy rendah karena beberapa huruf dominan.

Ciphertext (setelah enkripsi):

```
□×Ø□□Ù□๔□úÛü□ý×þ□□ÿ□□
```

Setiap byte punya frekuensi hampir sama → Entropy tinggi (~7.5-8.0)

4.3 Mengapa Entropy 5.0-6.5 adalah Target?

Proyek ini menargetkan entropy **5.0-6.5** untuk file terenkripsi. Mengapa bukan 8.0 (maksimal)?

File Asli + Intermittent Encryption:

File asli (plaintext) memiliki entropy ~3.5-5.0 (tergantung jenis file). Karena kita hanya mengenkripsi 20-30% dari file, entropy gabungan berada di antara:

- Plaintext entropy: ~4.0
- Encrypted chunks entropy: ~8.0
- **Hasil rata-rata:** ~5.5-6.0

Perbandingan:

- **Full Encryption:** Entropy = 7.9-8.0 (hampir perfect)
- **Intermittent (20% coverage):** Entropy = 5.0-6.5 (cukup baik)
- **No Encryption:** Entropy = 3.5-5.0 (buruk)

Interpretasi Target 5.0-6.5:

- **Di bawah 5.0:** Enkripsi terlalu sparse, file masih bisa di-recover
- **5.0-6.5:** Sweet spot - file corrupt tapi enkripsi cepat
- **Di atas 7.0:** Hampir full encryption (terlalu lambat untuk intermittent)

4.4 Implementasi Perhitungan Entropy

```
public static double CalculateEntropy(string filePath) {
    byte[] fileData = File.ReadAllBytes(filePath);

    // Hitung frekuensi setiap byte (0-255)
    var counts = new Dictionary<byte, int>();
    foreach (byte b in fileData) {
        if (counts.ContainsKey(b)) counts[b]++;
        else counts[b] = 1;
    }

    // Hitung entropy
    double entropy = 0;
    foreach (var count in counts.Values) {
        double p = (double)count / fileData.Length;
        entropy -= p * Math.Log(p, 2);
    }

    return entropy;
}
```

Contoh Output:

```
File asli (dummy_teks.txt): Entropy = 4.12
File encrypted: Entropy = 5.28
```

Entropy naik dari 4.12 ke 5.28, menunjukkan enkripsi berhasil meningkatkan randomness data sebesar 28%.

4.5 Hubungan dengan Chi-Square Test

Meskipun proyek ini tidak secara eksplisit menggunakan **Chi-Square Test**, entropy dan chi-square berkaitan erat:

Chi-Square Test mengukur seberapa jauh distribusi data dari distribusi uniform (merata). Formula:

```
χ² = Σ ((Observed - Expected)² / Expected)
```

Untuk data truly random:

- Setiap byte (0-255) harus muncul ~n/256 kali (Expected)
- Chi-square mendekati 0 (distribusi sempurna)

Hubungan dengan Entropy:

- **High Entropy (7.5-8.0)** → **Low Chi-Square** (distribusi uniform)
- **Low Entropy (2.0-4.0)** → **High Chi-Square** (distribusi skewed)

Untuk enkripsi yang baik, kita inginkan:

- **Entropy tinggi** (data acak)
- **Chi-square rendah** (tidak ada byte yang dominan)

 BAB 5: ALGORITMA SALSA20

5.1 Pengenalan Salsa20

Salsa20 adalah algoritma **stream cipher** yang dirancang oleh **Daniel J. Bernstein** pada tahun 2005. Cipher ini terkenal karena:

- **Kecepatan tinggi** (2-3x lebih cepat dari AES pada software)
- **Sederhana** (implementasi hanya ~300 baris kode)
- **Aman** (hingga kini belum ada serangan praktis yang berhasil)

Salsa20 adalah finalis di kompetisi **eSTREAM** (2008) dan direkomendasikan untuk software encryption.

5.2 Cara Kerja Stream Cipher

Berbeda dari **block cipher** (seperti AES) yang enkripsi per-blok (16 bytes), **stream cipher** bekerja seperti ini:

1. Generate **keystream** (deretan byte pseudo-random) dari Key + IV
2. XOR plaintext dengan keystream
3. Hasil XOR adalah ciphertext

Keuntungan:

- Enkripsi byte-by-byte → cocok untuk data streaming (video, audio)
- Plaintext dan ciphertext punya ukuran sama (no padding)
- Sangat cepat untuk software implementation

Proses Enkripsi/Dekripsi:

```
Plaintext:  "HELLO"   = 0x48 0x45 0x4C 0x4C 0x4F
Keystream:  (random)  = 0xA3 0xB2 0xC1 0xD0 0xE4
Ciphertext: (P XOR K) = 0xEB 0xF7 0x8D 0x9C 0xAB

Dekripsi:
Ciphertext: 0xEB 0xF7 0x8D 0x9C 0xAB
Keystream:  0xA3 0xB2 0xC1 0xD0 0xE4
Plaintext:  0x48 0x45 0x4C 0x4C 0x4F  = "HELLO"
```

Karena `A XOR B XOR B = A`, enkripsi dan dekripsi menggunakan operasi yang sama!

5.3 Salsa20 Core Function

Inti dari Salsa20 adalah **hash function** yang mengubah state 64-byte menjadi keystream 64-byte. Proses ini disebut **Salsa20 Quarter Round**.

State Matrix (16 × 4-byte words):

$$
\begin{bmatrix} \sigma_0 & Key_0 & Key_1 & Key_2 \\ Key_3 & \sigma_1 & IV_0 & IV_1 \\ Cnt_0 & Cnt_1 & \sigma_2 & Key_4 \\ Key_5 & Key_6 & Key_7 & \sigma_3 \end{bmatrix}
$$

Dimana:

- `σ` = konstanta "expand 32-byte k"
- `Key` = 256-bit encryption key
- `IV` = 64-bit initialization vector
- `Cnt` = 64-bit counter (increment setiap 64 bytes)

Proses:

1. Copy state matrix
2. Lakukan **20 rounds** operasi (ARX: Add, Rotate, XOR)
3. Add state asli ke hasil round
4. Convert ke 64 bytes keystream

5.4 Pentingnya IV (Initialization Vector)

IV adalah nilai 8-byte yang **harus unik** untuk setiap file atau session. Fungsinya:

- Membuat keystream berbeda meskipun key sama
- Mencegah serangan **keystream reuse**

Bahaya Reuse IV:

Jika dua plaintext berbeda dienkripsi dengan Key + IV yang sama:

```
C1 = P1 XOR Keystream
C2 = P2 XOR Keystream
C1 XOR C2 = P1 XOR P2  (keystream hilang!)
```

Attacker bisa melakukan **crib-dragging** untuk recover plaintext tanpa mengetahui key!

Solusi di Proyek Ini:

IV di-generate secara random untuk setiap file:

```
byte[] iv = new byte[8];
using (var rng = new RNGCryptoServiceProvider()) {
    rng.GetBytes(iv);
}
```

IV disimpan di footer file terenkripsi, sehingga dekripsi bisa menggunakan IV yang sama.

🛠️ BAB 6: IMPLEMENTASI TEKNIS

6.1 Struktur File Terenkripsi

File terenkripsi memiliki struktur:

```
[Encrypted Chunk 1]
[Plaintext Gap]
[Encrypted Chunk 2]
[Plaintext Gap]
...
[Encrypted Chunk N]
[8-byte IV]
[Marker: "[no-mode]"]
```

Footer:

- **IV (8 bytes):** Diperlukan untuk dekripsi
- **Marker (9 bytes):** String "[no-mode]" untuk validasi file terenkripsi

Mengapa Marker?

Marker memastikan file benar-benar hasil enkripsi dari program ini. Saat dekripsi:

1. Baca 9 byte terakhir
2. Cek apakah = "[no-mode]"
3. Jika beda → file bukan hasil enkripsi / corrupt

6.2 Key Derivation dengan PBKDF2

User memasukkan password string (misal: "MyPassword123"), tapi Salsa20 butuh key 256-bit. Konversi dilakukan dengan **PBKDF2 (Password-Based Key Derivation Function 2)**.

Proses:

```
Rfc2898DeriveBytes RFCDB = new Rfc2898DeriveBytes(password, salt, 1000);
byte[] key = RFCDB.GetBytes(32);
```

Parameter:

- **Password:** Input dari user
- **Salt:** 16-byte random (mencegah rainbow table attack)
- **Iterations:** 1000 (jumlah hash yang dilakukan)
- **Output:** 32 bytes (256-bit key)

Mengapa PBKDF2?

- **Slow by design:** 1000 iterasi membuat brute-force attack lambat
- **Salted:** Salt memastikan dua user dengan password sama menghasilkan key berbeda
- **Standard:** RFC 2898, digunakan di banyak sistem (Linux, iOS, Android)

Iterasi 1000 vs 100:

Proyek ini menggunakan 1000 iterasi untuk enkripsi key derivation di awal, tapi menggunakan 100 iterasi untuk per-chunk encryption (untuk performa). Idealnya:

- **Key derivation (1x):** 10.000 - 100.000 iterasi
- **Per-chunk (1000x):** 100 - 1.000 iterasi (trade-off speed vs security)

6.3 Coverage Calculation

Coverage adalah persentase file yang terenkripsi:

```
Coverage = (Total Encrypted Bytes / File Size) × 100%
```

Contoh:

```
File size: 5.242.886 bytes
Encrypted: 1.211.904 bytes
Coverage: 23.12%
```

Artinya, 23.12% dari file terenkripsi, sisanya (76.88%) masih plaintext.

Target Coverage:

- **File Kecil (< 1MB):** 100% (full encryption)
- **File Sedang (1-10MB):** 20-40%
- **File Besar (> 10MB):** 10-30%

Semakin besar file, semakin kecil coverage (untuk performa).

🐛 BAB 7: CATATAN BUG, FIX, DAN IMPROVEMENT

7.1 Bug Kritis yang Ditemukan

BUG #1: Salsa20 Instance Reuse

Deskripsi: Salsa20 adalah stream cipher dengan internal state (counter). Saat instance yang sama dipakai untuk enkripsi multiple chunks, counter tidak direset, menyebabkan keystream berbeda untuk setiap chunk.

Dampak:

- Enkripsi chunk ke-2 menggunakan keystream posisi 512+, bukan 0
- Dekripsi menggunakan instance baru (keystream posisi 0)
- Keystream tidak match → dekripsi gagal

Contoh:

```
// SALAH (BUG)
Algo.Salsa20 salsa = new Algo.Salsa20();
for (int i = 0; i < 10; i++) {
    encrypted[i] = salsa.Encrypt(chunks[i], key, iv); // State tidak reset!
}
```

Fix:

```
// BENAR
for (int i = 0; i < 10; i++) {
    using (Algo.Salsa20 salsa = new Algo.Salsa20()) {
        encrypted[i] = salsa.Encrypt(chunks[i], key, iv); // Instance baru
setiap kali
    }
}
```

Lesson Learned: Stream cipher dengan stateful counter harus menggunakan instance baru atau reset manual untuk setiap encryption session.

BUG #2: Loop Enkripsi Melompat Melewati Batas File

Deskripsi: Loop enkripsi menambah posisi dengan `bytesToRead` lalu `jump`, tanpa cek apakah posisi baru melebihi fileSize.

Kode Bermasalah:

```
while (currentPos < fileSize) {
    currentPos += bytesToRead;
    currentPos += rng.Next(minJump, maxJump); // Bisa melebihi fileSize!
}
```

Dampak:

- Untuk file 5MB dengan jump 10MB, loop berhenti setelah chunk pertama
- Coverage sangat rendah (1-5% padahal target 20-30%)

Fix:

```
while (currentPos < fileSize) {
    // ... enkripsi chunk
    currentPos += bytesToRead;

    if (currentPos >= fileSize) break;

    int jump = rng.Next(minJump, maxJump);
    if (currentPos + jump >= fileSize) break; // Cek sebelum loncat

    currentPos += jump;
}
```

BUG #3: GetAdaptiveParams Menggunakan Nama File dengan .EKSTENSION

Deskripsi: Saat dekripsi, `GetAdaptiveParams(file, ...)` dipanggil dengan file = "dummy_teks.txt.LOCKED". Fungsi ini extract extension dengan `Path.GetExtension()` yang menghasilkan ".LOCKED", bukan ".txt".

Dampak:

- Enkripsi menggunakan parameter untuk ".txt" (chunk 512, jump 2KB)
- Dekripsi menggunakan parameter "default" (chunk 128, jump berbeda)
- Parameter tidak match → dekripsi gagal

Fix:

```
string originalName = file.Replace("." + extension, "");
string tempPath = Path.Combine(Path.GetDirectoryName(file), originalName);
GetAdaptiveParams(tempPath, fileSize, ...); // Gunakan nama asli
```

BUG #4: Footer Tidak Terhapus dengan Benar

Deskripsi: Saat dekripsi, footer (IV + marker) dipotong dengan `fs.SetLength()`, tapi FileStream masih terbuka dalam mode ReadWrite. MMF yang dibuka setelahnya masih membaca data lama dari memory buffer.

Dampak:

- File terlihat sudah dipotong (size berkurang)
- Tapi MMF masih membaca data dengan footer
- Marker tidak hilang dari memory

Fix:

```
// Baca IV dulu
using (FileStream fs = new FileStream(file, FileMode.Open, FileAccess.Read))
{
    fs.Seek(-footerSize, SeekOrigin.End);
    fs.Read(iv, 0, IV_SIZE);
    // ... validasi marker
}

// TUTUP file, baru potong
using (FileStream fs = new FileStream(file, FileMode.Open, FileAccess.Write))
{
    fs.SetLength(originalLength);
    fs.Flush();
}

// Tunggu OS release file handle
Thread.Sleep(100);

// Baru buka MMF untuk dekripsi
```

BUG #5: Key Derivation di Dalam Loop

Deskripsi: `Rfc2898DeriveBytes` dipanggil di dalam function `encryptSalsa()` yang dipanggil berkali-kali dalam loop.

Dampak:

- PBKDF2 dengan 1000 iterasi sangat lambat
- Untuk 100 file, derive key dilakukan 100x
- Waktu enkripsi 10x lebih lambat

Fix:

```
// Derive key SEKALI di luar loop
Rfc2898DeriveBytes RFCDB = new Rfc2898DeriveBytes(password, salt, 1000);
byte[] key = RFCDB.GetBytes(32);
RFCDB.Dispose();

// Gunakan key yang sama untuk semua chunk
for (...) {
    encrypted = salsa.Encrypt(chunk, key, iv);
}
```

7.2 Improvements yang Dilakukan

IMPROVEMENT #1: Adaptive Parameters Berdasarkan Jenis File

Before: Semua file menggunakan parameter yang sama (chunk 512, jump 2KB).

After: Parameter disesuaikan dengan karakteristik file:

- Text file: Chunk kecil (512), jump rapat (2-8KB)
- Video: Chunk besar (256KB), jump jauh (1-5MB)
- Database: Chunk sesuai page size (4KB)

Benefit:

- Enkripsi lebih efektif (targetkan bagian kritis)
- Performa lebih baik (chunk besar untuk file besar)

IMPROVEMENT #2: Tambah Flush() di MMF Write

Deskripsi: Setelah `accessor.WriteArray()`, tambahkan `accessor.Flush()` untuk memastikan data langsung tertulis ke disk, bukan hanya ke buffer.

Benefit:

- Mencegah data loss jika program crash
- Memastikan enkripsi benar-benar tersimpan

**IMPROVEMENT #3:

Validasi bytesToRead > 0**

Deskripsi: Sebelum enkripsi chunk, cek apakah `bytesToRead > 0`. Jika 0 atau negatif, skip chunk tersebut.

Benefit:

- Mencegah crash saat enkripsi chunk kosong
- Handle edge case (file size pas kelipatan chunk)

IMPROVEMENT #4: Log Debug untuk Troubleshooting

Deskripsi: Tambahkan log untuk:

- ChunkSize, MinJump, MaxJump
- Total chunks encrypted
- Coverage percentage
- Entropy score

Benefit:

- Mudah debug jika ada masalah
- Verifikasi parameter bekerja sesuai ekspektasi
- Transparansi untuk user

IMPROVEMENT #5: Delay Setelah Close MMF

Deskripsi: Tambahkan `Thread.Sleep(100-200ms)` setelah close MMF dan sebelum operasi file lain (rename, delete).

Benefit:

- Memberikan waktu bagi OS untuk release file handle
- Mencegah error "file is being used by another process"
- Meningkatkan stabilitas di berbagai OS

7.3 Rangkuman Teknis

Total Bugs Fixed: 5 major bugs

Total Improvements: 5 enhancements

Waktu Development: ~10 tahun troubleshooting dan fixing

Lines of Code:

- Program.cs: ~200 lines
- Algo.cs: ~600 lines
- Total: ~800 lines

Performa:

- Enkripsi 5MB: ~0.5 detik
- Coverage: 20-30% (target tercapai)
- Entropy: 5.28 (dalam range 5.0-6.5)

Kesuksesan:

- Enkripsi: 100% berhasil
- Dekripsi: 100% berhasil (file identik dengan original)
- Intermittent pattern: Konsisten dan reproducible

🎓 KESIMPULAN

Proyek ini berhasil mengimplementasikan **Intermittent Encryption** dengan:

Teknologi Utama:

- Memory-Mapped Files untuk efisiensi memory
- Salsa20 stream cipher untuk kecepatan
- PBKDF2 untuk key derivation yang aman
- LCG untuk deterministik random jumps

Fitur Unggulan:

- Adaptive parameters berdasarkan jenis file
- Entropy scoring untuk verifikasi kualitas enkripsi
- Coverage calculation untuk transparansi
- Reproducible encryption pattern untuk dekripsi

Lesson Learned:

- Stream cipher dengan state harus di-handle dengan hati-hati
- File I/O dengan MMF butuh proper synchronization
- Parameter tuning penting untuk balance speed vs security
- Testing dengan berbagai ukuran file crucial untuk menemukan edge cases

Potensi Pengembangan:

- Multi-threading untuk enkripsi parallel
- Compression sebelum enkripsi untuk efisiensi lebih
- GUI untuk kemudahan user
- Support untuk file > 100GB dengan streaming

📜 HALAMAN KREDIT

👥 TIM PENGEMBANGAN

Arie Kolmteistov - *Lead Developer & Documentation Specialist*

Periode: 2016 - Sekarang

Kontribusi:

- Implementasi awal algoritma Salsa20 (2016-2021)
- Desain arsitektur core encryption engine
- Dokumentasi teknis dan user manual
- Testing dan quality assurance

Viktor Hartmann (Codename) - *Research & Logic Architect*

Periode: 2022 - 2025

Kontribusi:

- Konsep dan desain intermittent encryption pattern
- Penelitian adaptive parameters berdasarkan tipe file
- Algoritma deterministic random jump dengan LCG
- Optimasi entropy scoring dan coverage calculation
- Analisis performa dan benchmarking

Sebastian Reichert (Codename) - *Code Optimizer & Technical Writer*

Periode: 2024 - 2025

Kontribusi:

- Debugging dan fixing critical bugs (Salsa20 state management, MMF synchronization)
- Performance improvements (key derivation optimization, adaptive chunking)
- Memory-Mapped Files implementation dan troubleshooting
- Technical documentation dan whitepaper writing
- Code review dan best practices enforcement

🛠️ TEKNOLOGI & FRAMEWORK

Bahasa Pemrograman: C# (.NET Core 6.0+)

Algoritma Kriptografi:

- Salsa20 Stream Cipher (Daniel J. Bernstein, 2005)
- PBKDF2 (RFC 2898)
- SHA-256 Hash Function

Libraries:

- System.IO.MemoryMappedFiles
- System.Security.Cryptography
- System.Diagnostics

Development Tools:

- Visual Studio Code
- .NET SDK
- Git Version Control

📓 REFERENSI & INSPIRASI

Academic Papers:

- Bernstein, D.J. (2005). "The Salsa20 Family of Stream Ciphers"
- NIST Special Publication 800-132: "Recommendation for Password-Based Key Derivation"

Ransomware Research:

- Analysis of LockBit 3.0 intermittent encryption (2022)
- BlackMatter partial encryption techniques (2021)

Open Source Contributions:

- Salsa20 C# Implementation (adapted from Chaos.NaCl)

🏆 PENCAPAIAN

Timeline Pengembangan:

2016-2018: Fase Eksplorasi

- Implementasi Salsa20 dari spesifikasi
- Testing enkripsi/dekripsi basic
- Dokumentasi awal

2019-2021: Fase Konsolidasi

- Code refactoring dan optimization
- Penambahan RSA key encryption
- MD5 hashing untuk integrity check

2022-2023: Fase Inovasi

- Research intermittent encryption
- Desain adaptive parameters
- Proof of concept dengan file besar

2024-2025: Fase Finalisasi

- Memory-Mapped Files implementation
- Bug fixes dan stability improvements
- Dokumentasi lengkap dan presentasi

Metrics:

- Total Lines of Code: ~800
- Bugs Fixed: 5 critical, 12 minor
- Test Files Processed: 1000+
- Largest File Encrypted: 50GB

🙏 ACKNOWLEDGMENTS

Terima kasih kepada:

- **Daniel J. Bernstein** untuk algoritma Salsa20 yang elegant dan cepat
- **Komunitas .NET** untuk dokumentasi MMF yang lengkap
- **Security Researchers** yang membuka wawasan tentang ransomware techniques
- **Beta Testers** yang membantu menemukan edge cases

Special Thanks:

- Teman-teman yang mendukung selama development
- Komunitas Toxic~~Stack~~Overflow untuk solusi technical issues
- Open source contributors yang menginspirasi project ini

📞 KONTAK

Project Repository: [Private/Internal Use]

Untuk pertanyaan & saran:

- Email: [encryption-engine@tuta.io]
- GitHub: [https://github.com/kolmteistov]

Lisensi: Educational/Research Use Only

"public static void."
