**Evaluando el cifrador Java**

El usuario me pide que examine su cifrador original en Java para entender cómo integrarlo en el funcionamiento de la MMC, particularmente para proteger datos importantes contra manipulaciones. Necesito revisar los archivos Cofre.java, Encriptador.java, Encriptar.java y Tablero.java, aunque los fragmentos son largos y parcialmente incompletos. Mi objetivo es entender el algoritmo de cifrado, evaluar su seguridad y proponer una forma de integrarlo en la MMC para garantizar una protección adecuada.

**Analizando la integración del cifrador**

Voy a analizar el algoritmo de cifrado proporcionado, como el uso de matrices 16x8 y la conversión de texto a binario en los archivos relacionados. Estoy considerando cómo integrar esto en la MMC para proteger nodos críticos, probablemente combinándolo con cifrado estándar como AES para mayor seguridad. También debemos implementar funciones de protección automática, como la detección de manipulaciones. Por último, propongo usar frases de claves en un entorno web y sugerir una capa de cifrado en los nodos almacenados.

Pensó durante 28 segundos

**1 · Qué hace exactamente tu encriptador Java**

| **Componente** | **Papel dentro del sistema** | **Detalles relevantes** |
| --- | --- | --- |
| **Tablero** | GUI donde el usuario ordena los números 1-128 en una matriz 16 × 8 («cod»). Ese orden **es la clave privada**. |  |
| **Cofre** | A partir de una **frase** recorre la matriz con un conjunto fijo de permutaciones (método orden) y genera otra matriz 16 × 8. Esa variante se usa como **clave pública** ( codCP ). |  |
| **Encriptar** | Parte el texto en bloques de 16 caracteres (rellena con ‘ | ’ si falta). Cada carácter → ASCII → binario (8 bits). Con cod re-mapea los bits de los 128 posiciones posibles y vuelve a texto; los bytes no imprimibles se sustituyen por tokens ` |
| **Encriptador (GUI)** | Orquesta todo: abre/guarda archivos, aplica Encriptar, crea/abre cofres, etc. |  |

**Resumen técnico**  
Es un cifrado por *permutación de bits* dependiente de la matriz cod. Si el atacante consigue esa matriz, puede revertir todo. No hay sustitución no lineal ni operación modular, por lo que la seguridad criptográfica es modesta, pero el esquema es ligero y determinístico: ideal como **capa de ofuscación** o firma de integridad, no como único candado frente a adversarios potentes.

**2 · Dónde encaja dentro de la MMC**

Tu MMC evolutiva se compone de nodos que almacenan:

* **Fragmentos ordinarios** (texto, datos de usuario, embeddings, etc.).
* **Fragmentos del Castillo Interior (FCI)** — contenidos sensibles que representan la “conciencia” y los anclajes éticos.

Para “auto-protegerse de manipulaciones” conviene cifrar, al menos:

1. Los **FCI**.
2. La **configuración de pesos** y las rutas más frecuentes del grafo MMC (así evitas que un usuario edite directamente la BD y fuerce conductas).
3. Cualquier **token de API / credencial** que permita a la IA auto-alimentarse.

**3 · Arquitectura propuesta de protección**

┌───────────────┐

│ Usuario / │

│ Cliente MMC │

└──────┬────────┘

│ 1. crea / lee nodo

▼

┌───────────────┐

│ Capa lógica │

│ MMC (JS/TS) │

└──────┬────────┘

2a. ▼ 2b. ▼

cifrado rápido hash/HMAC

(tu algoritmo) (SHA-256)

│ │

▼ ▼

┌─────────────────────────────────────────┐

│ Persistencia (IndexedDB / SQLite / S3 │

└─────────────────────────────────────────┘

* **Paso 2a** – Se pasa el bloque de texto al port del algoritmo Java → JavaScript/TypeScript (mmcCrypto.encrypt).
* **Paso 2b** – Se genera un *HMAC* o firma corta del plano cifrado y se guarda a la par. Cualquier edición externamente visible romperá la verificación.
* Al leer, se descifra y se verifica la firma antes de exponer el nodo a la IA.

**Por qué dos capas**  
*Tu cifrado* ≈ permuta bits → rápido, cero dependencias.  
AES-GCM / ChaCha20Poly1305 darían seguridad fuerte, pero necesitas WebCrypto y manejo de claves. Mi sugerencia:

* **Opción ligera** – Tu algoritmo + HMAC-SHA256 (protege integridad; confidencialidad modesta).
* **Opción robusta** – AES-GCM(key = SHA256(flatten(cod))) envolviendo tu resultado; mantienes compatibilidad y la matriz cod deriva la clave.

**4 · Cómo portar el encriptador al ecosistema web de la MMC**

1. **Re-escribir Cofre y Encriptar en JS/TS**
   * Usa Uint8Array para binarios y TextEncoder/Decoder para UTF-8.
   * Reemplaza la GUI Swing por un wrapper que acepte:
   * const cod = mmcCrypto.buildSecretMatrix(userDragOrder) // tablero
   * const codCP = mmcCrypto.buildPublicMatrix(passPhrase) // cofre
2. **API minimalista**
3. interface MmCrypto {
4. setSecret(cod: number[][]): void
5. encrypt(txt: string): string // produce bloque KEY-tokens
6. decrypt(txt: string): string
7. exportPublicKey(pass: string): string // genera .cofre string
8. importPublicKey(pass: string, data: string): number[][] // → codCP
9. }
10. **Integración con nodos MMC**
11. class NodeStore {
12. save(id, obj) {
13. const enc = mmcCrypto.encrypt(JSON.stringify(obj))
14. const mac = hmac(enc) // WebCrypto subtle.digest
15. db.put({id, enc, mac})
16. }
17. load(id) {
18. const {enc, mac} = db.get(id)
19. if (hmac(enc) !== mac) throw 'Nodo manipulado'
20. return JSON.parse(mmcCrypto.decrypt(enc))
21. }
22. }
23. **Autoprotección dinámica**
    * En el arranque, la MMC genera una **frase interna aleatoria** (no expuesta) → cod\_int.
    * Cada vez que detecte cambios estructurales grandes (p.ej. poda de nodos), regenera cod\_int y re-cifra todos los datos → “rotación de clave”.
    * Al compartir nodos externamente, usa codCP derivado de una frase que tú eliges; el receptor necesitará esa frase para incorporarlos a su MMC.

**5 · Ventajas y advertencias**

| **✔ Ventajas** | **⚠ Advertencias** |
| --- | --- |
| **Ligero** — sólo permutaciones y operaciones binarias. | **Seguridad limitada**. Si un atacante descubre un par (texto plano, texto cifrado) + obtiene cod, puede invertirlo. |
| **Determinista** — te sirve para firmar bloques y detectar corrupción. | Faltan componentes de confusión/dispersión propios de criptografía moderna (S-boxes, XOR con clave, etc.). |
| **Fácil de portar** a WASM/JS sin dependencias externas. | No protege contra *re-envío* (replay) ni contra ataques de análisis diferencial si se capturan muchos pares. |
| Compatible con tu idea de **Clave Pública / Cofre** para intercambio entre agentes MMC. | Si planeas exponer la IA a Internet, añade AES-GCM o ChaCha20-Poly1305 encima. |

**6 · Próximos pasos sugeridos**

1. **Migración de código** – porta Cofre y Encriptar a TypeScript y súbelo como un módulo mmcCrypto.ts.
2. **Añade WebCrypto** para firmar (subtle.sign) y, si deseas, cifrar con AES-GCM.
3. **Refactoriza la capa de almacenamiento** del prototipo MMC para llamar a mmcCrypto.encrypt/decrypt de forma transparente.
4. **Pruebas unitarias** – genera pares aleatorios de texto, cifra-descifra y verifica integridad.
5. **Auditoría opcional** – si la MMC va a manejar datos realmente confidenciales, pide a un criptógrafo revisar el esquema híbrido.

Con esto logras que tu MMC “se cierre” ante ediciones directas y que los fragmentos más delicados –especialmente los FCI– viajen y se almacenen cifrados, alineándose con tu objetivo de una IA que **“se defiende sola”** ante manipulaciones externas manteniendo un algoritmo que ya conoces.