Relazione di Progetto

Crittografia Simmetrica e Asimmetrica

Daniele Guiducci
Sicurezza dell'informazione

27 giugno 2025

Indice

Capitolo 1

Introduzione

L'attività progettuale si basa su due categorie principali di crittografia: quella simmetrica e quella asimmetrica. Sono stati sviluppati in Java diversi wrapper e classi di supporto per algoritmi come AES (in modalità ECB, CBC e GCM), ChaCha20 per la parte simmetrica, e RSA per la parte asimmetrica.

Ogni algoritmo è stato accompagnato da test pratici volti a evidenziare eventuali criticità legate all'uso improprio dei parametri (ad esempio, IV statici, chiavi troppo corte, ecc.).

Parallelamente all'implementazione, è stata svolta un'analisi di queste vulnerabilità, dimostrando con esempi pratici come alcune scelte errate possano compromettere gravemente la sicurezza del sistema. Ho simulato scenari di cifratura ripetitiva, uso di IV costanti, e la debolezza delle chiavi RSA a 512 bit.

Gli obiettivi principale del progetto sono due : da un lato sviluppare competenze pratiche nell'uso delle primitive crittografiche in Java (basandosi anche su quello che abbiamo visto nell'esercitazioni), dall'altro comprendere e documentare le problematiche più comuni in cui si può incorrere durante l'implementazione di sistemi sicuri. Chiaramente gli esempi riportati sono a fini didattici e non rispecchiano possibili attacchi che un attaccante può fare attivamente in un canale.

Capitolo 2

Crittografia Simmetrica

Nel progetto, l'implementazione della cifratura simmetrica è stata strutturata in modo modulare per garantire riusabilità e flessibilità. A tal fine, sono state progettate due classi fondamentali: CipherWrapper e SecureRandomWrapper. Con questa suddivisione l'implementazione è facilmente estendibile a nuove modalità di cifratura.

La classe CipherWrapper funge da componente base per la gestione degli algoritmi simmetrici come AES e ChaCha20. Essa incapsula l'oggetto Cipher fornito dalla Java Cryptography Architecture (JCA) e centralizza la generazione e l'utilizzo della chiave simmetrica, che viene creata dinamicamente con un KeyGenerator. La progettazione della classe permette di gestire diverse modalità operative del cifrario (come ECB, CBC, GCM) semplicemente specificando la trasformazione desiderata all'atto della creazione. Inoltre, CipherWrapper supporta metodi sovraccarichi per includere parametri specifici come IV o AAD (Authenticated Additional Data), rendendola adatta anche per algoritmi AEAD (Authenticated Encryption with Associated Data), come AES-GCM o ChaCha20-Poly1305.

La classe SecureRandomWrapper, invece, fornisce un'interfaccia sicura e controllata alla generazione casuale di numeri e byte, elemento essenziale nella crittografia moderna. Utilizzando il costruttore basato sull'algoritmo "SHA1PRNG", essa consente la riproducibilità (opzionale) dei risultati tramite seed esplicito, ma anche la generazione robusta di IV casuali tramite il metodo generateIV. L'astrazione di SecureRandomWrapper è stata utile per isolare l'aspetto randomico dalla logica crittografica, facilitando la manutenzione e l'analisi dei componenti del sistema.

2.1 Analisi sintetica dei metodi

1. `CipherWrapper(String transformation, String keyGenAlgo, SecureRandomWrapper srw)`:
   Inizializza il cifratore con l'algoritmo specificato e genera una chiave segreta con `computeSecretKey`.

2. `computeSecretKey(String keyGenAlgo, SecureRandom sr)`:
   Crea una chiave segreta con l'algoritmo dato e un generatore di numeri casuali sicuro.

3. `encrypt(String plaintext)`:
   Cifra un testo in modalità ECB (non usa parametri aggiuntivi).

4. `encrypt(String plaintext, AlgorithmParameterSpec spec)`:
   Cifra un testo in modalità che richiede parametri aggiuntivi (es. IV per CBC).

5. `encrypt(String plaintext, String additionalData, AlgorithmParameterSpec spec)`:
   Cifra un testo con dati aggiuntivi (usato per AEAD come AES-GCM).

6. `decrypt(byte[] ciphertext)`:
   Decifra un testo cifrato in modalità ECB.

7. `decrypt(byte[] ciphertext, AlgorithmParameterSpec spec)`:
   Decifra un testo cifrato con parametri specifici (IV).

8. `decrypt(byte[] ciphertext, String additionalData, AlgorithmParameterSpec spec)`:
   Decifra un testo cifrato con dati aggiuntivi (per AEAD).

2.2 Algoritmi implementati

Per estendere la classe generica `CipherWrapper`, sono state create quattro classi specializzate che rappresentano diverse modalità operative dell'algoritmo AES (o ChaCha20). Ogni classe incapsula una modalità ben precisa, fornendo metodi di cifratura e decifratura su misura e migliorando la chiarezza del codice applicativo.

- **AESECBCipherWrapper**: Fornisce una variante dell'AES in modalità ECB, utile per mostrare le vulnerabilità legate a questa modalità. La semplicità di questa classe riflette anche la semplicità (e pericolosità) della modalità ECB, che non prevede IV e produce output ripetibili per input identici. Implementa un cifrario a blocchi e il limite principale è il suo determinismo, cioè che a blocchi uguali di testo in chiaro corrispondono blocchi uguali di testo cifrato.

- **AESCBCWrapper**: Implementa l'AES in modalità CBC con padding PKCS5. La classe consente sia la generazione automatica di un IV casuale tramite `SecureRandomWrapper`, sia l'iniezione di un IV fisso per scopi dimostrativi o di test. L'uso esplicito dell'`IvParameterSpec` garantisce il controllo sulla componente di inizializzazione, fondamentale per la sicurezza del CBC. Anche questa modalità di cifratura implementa dei cifrari a blocchi, CBC nasce per superare il determinismo di ECB, ogni blocco di testo in chiaro è in XOR con il cifrato del blocco precedente, in modo tale da rendere non deterministica l'uscita. Uno dei principali limiti è la propagazione dell'errore, se viene alterato il testo cifrato (aggiunta/cancellazione/modifica di un bit) l'errorre viene propagato in tutta la catena

- **AESGCMCipherWrapper**: Gestisce la modalità AES-GCM, un algoritmo AEAD che unisce cifratura e autenticazione. Include la possibilità di passare AAD (Additional Authenticated Data) e specifica un `GCMParameterSpec` contenente IV e lunghezza del MAC, offrendo una protezione avanzata contro gli attacchi di tipo modificazione/manomissione del messaggio. E' basato su un cifrario a blocchi (AES), ma grazie alla modalità CTR si comporta come un cifrario a flusso sincrono, con l'aggiunta di autenticazione tramite MAC. La modalità CTR ha uno schema di funzionamento base che mette in XOR il testo in chiaro con l'uscita cifrata di un contatore.

- **ChaCha20CipherWrapper**: Implementa l'algoritmo ChaCha20, alternativa moderna all'AES per ambienti software-oriented. Utilizza il `ChaCha20ParameterSpec`, che include sia il nonce (IV) sia un contatore iniziale. Sebbene nel costruttore venga indicato `"AES"` come algoritmo per la generazione della chiave (una semplificazione), la classe utilizza internamente il cifrario ChaCha20, supportando cifratura sicura ed efficiente anche su dispositivi con prestazioni limitate. E' un cifrario a flusso sincrono, dove il keystream è indipendente dal testo cifrato (al contrario di quelli a flusso autosincronizzante che hanno retroazione); un possibile limite di questa tipologia di cifrari è la robustezza contro

attacchi attivi sul canale: nel caso di aggiunta/cancellazione di un bit del testo cifrato c'è una perdita di sincronismo fra i due cifrari e quindi devono essere fatti ripartire dallo stesso punto per avere una comunicazione consistente, nel caso invece di modifica di un bit l'errore è limitato al bit d'interesse.

2.3 Risulati e vulnerabilità analizzate

2.3.1 AES in modalità ECB

```
Plaintext:  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Ciphertext: C629489560126CDFF59F752871B9AE64
            C629489560126CDFF59F752871B9AE64
            9927B2C7327CA240DF5B5CFB6A40473C

Decrypted:  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
[VULNERABILITA'] In ECB blocchi di testo in chiaro uguali
                 corrispondono a blocchi di testo cifrato identici.
```

Commento: AES in modalità ECB (Electronic Codebook) non offre alcun grado di randomizzazione: blocchi di testo in chiaro uguali generano blocchi cifrati identici. Questo permette di individuare pattern nel messaggio cifrato, rendendolo vulnerabile all'analisi. ECB non è adatto per dati strutturati o ripetitivi. In generale al giorno d'oggi non è utilizzato

2.3.2 AES in modalità CBC con IV fisso

```
Cifrato #1: 9IOmm8vvkvvg3PCJNBoA2x2iLCxdvqeRX+bj93/+av8=
Cifrato #2: 9IOmm8vvkvvg3PCJNBoA2x2iLCxdvqeRX+bj93/+av8=

Uguali ? => true
```

```
[VULNERABILITA'] IV fisso => messaggi identici =>
                 ciphertext identici.
```

Commento: CBC (Cipher Block Chaining) supera il limite deterministico di ECB solo se l'IV (Initialization Vector) è diverso per ogni cifratura. Utilizzando un IV fisso, i messaggi identici generano ciphertext identici, violando il principio di *indistinguibilità semantica*. L'IV deve essere sempre *casuale e non riutilizzato*.

2.3.3 AES in modalità CBC con IV casuale

```
Cifrato #1: plw8RrpPk3THz2Iq/SDMuteT/CpLgwtxY+7icN5gxFk=
Cifrato #2: CqJMUipdnkDJKBYcn022K1o2jNUJ65eo3n2HJHTUFFg=

Uguali?      false
[SICURO] IV diverso => ciphertext diverso anche
         se il plaintext e' lo stesso.
```

Commento: Qui la randomizzazione dell'IV garantisce che lo stesso messaggio produca ciphertext differenti, rendendo la cifratura sicura e resistente all'analisi. Questo è il comportamento corretto atteso da CBC.

2.3.4 AES in modalità GCM (AEAD)

```
Ciphertext: /RL2JwMRhvzLf2WPkNxRjnpWd2iQeK//YH29DT4uB6cKeVPe
Decrypted:  Messaggio importante

[SICURO] AAD corretto => decrittazione riuscita.
[OK] AAD modificato => decrittazione fallita => integrita' garantita.
```

Commento: AES-GCM (Galois/Counter Mode) è una modalità **AEAD** (Authenticated Encryption with Associated Data) che garantisce sia la riservatezza che l'integrità. L'uso corretto di dati autenticati (AAD) consente di rilevare modifiche ai dati durante la decifrazione.

2.3.5 ChaCha20 con nonce/counter riusati

```
Ciphertext #1: UzMm+LKVjFk=
Ciphertext #2: JhscfowREBE=

Uguali?          false
[ROBUSTEZZA] Nonce/counter riusati => cifrati comunque diversi.
```

Commento: Anche se in questo test i ciphertext risultano diversi (probabilmente per via della diversa chiave interna), il riutilizzo di nonce e counter in un cifrario a flusso come ChaCha20 è pericoloso: può esporre informazioni sul plaintext tramite attacchi al keystream. È fondamentale usare nonce univoci per ogni messaggio.

2.3.6 ChaCha20 con chiave/nonce/counter riusati

```
Ciphertext #1: ooyH1ST/REo=
Ciphertext #2: ooyH1ST/REo=

Uguali?        true
[VULNERABILITA'] Riutilizzo di chiave/nonce/counter =>
                 riuso keystream => forte vulnerabilit
```

Commento: In questo scenario critico, riutilizzando esattamente gli stessi parametri (chiave, nonce e counter), il keystream generato è lo stesso. Questo porta a ciphertext identici, compromettendo totalmente la sicurezza del messaggio. In un cifrario a flusso sincrono come ChaCha20, **la riusabilità dei parametri è una delle peggiori vulnerabilità possibili**.

2.3.7 Considerazioni

Questi esperimenti dimostrano come:

- la **modalità ECB** sia insicura per la maggior parte degli usi pratici;
- l'uso **corretto dell'IV** (casuale e non ripetuto) sia fondamentale in modalità come CBC;
- le modalità **AEAD** come AES-GCM o ChaCha20-Poly1305 forniscano integrità oltre che riservatezza;
- i **cifrari a flusso sincrono** richiedano estrema attenzione nell'uso di nonce e contatori.

Capitolo 3

Crittografia Asimmetrica

3.1 RSA: cifratura e firma digitale

Per la parte di cifratura asimmetrica è stato adottato l'algoritmo RSA, un sistema a chiave pubblica che si basa su problemi matematici computazionalmente difficili, come la fattorizzazione di grandi numeri primi. L'implementazione è centrata sulla classe `RSACipherWrapper`, che incapsula le funzionalità principali per la generazione delle chiavi, cifratura e decifratura.

La classe genera una coppia di chiavi (`KeyPair`) tramite l'algoritmo `RSA` con una lunghezza configurabile (es. 2048 o 4096 bit), bilanciando sicurezza e prestazioni. La cifratura viene eseguita con la chiave pubblica, rendendo il messaggio leggibile solo al possessore della chiave privata, mentre la decifratura utilizza la chiave privata per ricostruire il messaggio originale.

L'algoritmo di cifratura è inizializzato con la trasformazione `RSA/ECB/PKCS1Padding`, che, pur essendo una scelta comune nelle librerie Java, presenta vulnerabilità note in quanto l'uso di ECB e padding deterministici può esporre il sistema ad *attacchi di tipo padding oracle* o a *pattern analysis*, specialmente se utilizzato direttamente per cifrare dati lunghi.

Inoltre, se estesa, la classe può essere impiegata anche per firmare digitalmente un messaggio: in questo caso si applica la cifratura con la chiave privata, e la verifica viene effettuata con la chiave pubblica, garantendo **autenticità** e **integrità** del contenuto.

3.2 Risultati e Vulnerabilità analizzate

3.2.1 Generazione delle chiavi (RSA 512)

```
====== Chiavi generate (RSA 512) ======
Chiave pubblica:  MFwwDQYJ.....
Chiave privata:   MIIBVgIB.....
```

In questo output, viene generata una coppia di chiavi RSA a 512 bit. Tali chiavi, però, sono considerate crittograficamente deboli al giorno d'oggi e non devono essere usate in ambienti reali perchè non garantiscono un livello di sicurezza tale da evitare attacchi di brute force. In ambito pratico e professionale infatti si raccomanda di usare almeno chiavi RSA da 2048 bit, che garantiscono un livello di sicurezza accettabile.

3.2.2 Cifratura / Decifratura

```
====== Cifratura / Decifratura ======
Messaggio cifrato (Base64): djBLqVv1e....
Messaggio decifrato:        Questo e' un messaggio segreto
```

Il messaggio originale è stato cifrato con la chiave pubblica e correttamente decifrato con la chiave privata, dimostrando il corretto funzionamento dell'algoritmo.

3.2.3 Firma Digitale

```
====== Firma ======
Firma digitale (Base64): ECrT3rTD....
Verifica firma: VALIDA
```

La firma del messaggio è verificata correttamente. Questo meccanismo garantisce l'autenticità del mittente e l'integrità del messaggio ricevuto. RSA implementa uno schema di firma con recupero. Questo significa che la firma contiene l'intero messaggio m, il messaggio viene frammentato e ad ogni blocco viene applicato RSA (naturalmente dim. blocco $<$ modulo). La firma con

recupero (come abbiamo visto a lezione) in origine non garantisce l'inalterabilità del documento per questo non dovrebbe essere usato come firma con validità legale, è comunque possibile adottare firma con recupero con validità legale se vengono utilizzate delle funzioni di ridondanza che arricchiscono il messaggio originale con informazioni univoche che ci permettono di dare un significato univoco alla concatenazione.

3.2.4 Vulnerabilità: RSA-512

```
======= [VULNERABILITA': RSA-512] =======
- Le chiavi RSA-512 sono deboli e possono essere fattorizzate
    pubblicamente.
- Ad esempio, il modulo n puo' essere inserito su
    https://factordb.com per recuperarne i fattori primi (p, q).
- Questo permette a un attaccante di ricostruire la chiave privata.
```

L'uso di RSA-512 espone il sistema a gravi vulnerabilità: la chiave privata può essere ricostruita conoscendo il modulo pubblico. Questo è possibile tramite database pubblici come `factordb.com`, dove è sufficiente incollare il modulo n per ottenerne i fattori.

3.2.5 Nota finale:

la sicurezza della crittografia RSA dipende fortemente dalla dimensione del modulo. È fortemente raccomandato usare chiavi di almeno **2048 bit** per applicazioni pratiche.

Capitolo 4

Conclusioni

L'utilizzo di cifrari simmetrici o asimmetrici dipende dalla tipologia di requisiti che devono essere garantiti. In generale i cifrari simmetrici sono più efficienti in termini di prestazioni in quanto non dipendono da esponenziazioni modulari, quindi sono più veloci rispetto a quelli asimmetrici.
Il problema dei cifrari simmetrici in contesti distribuiti è legato alla distribuzione della chiave e quindi della scalabilità.

Per quanto riguarda quelli asimmetrici vengono utilizzati soprattuto quando si vuole garantire l'autenticità del messaggio in quanto il possessore che firma un messaggio attribuisce ad esso la paternità in quanto è l'unico possessore della chiave privata.

In entrambi i casi, simmetrici e asimmetrici, la robustezza dipende da come vengono utilizzati i parametri degli algoritmi (IV, dimensioni chiavi, nonce) e spesso vengono combinati entrambi i cifrari per andare a sfruttare al meglio i vantaggi di entrambi, come nei cifrari ibridi.