Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica — Scienza e Ingegneria

Corso di Laurea Triennale in Informatica

Erasure Codes per Data Flooding Against Ransomware

Relatore:
Chiar.mo Prof.
Saverio Giallorenzo

Presentata da:
Daniele Grossi

Sessione Marzo

Anno Accademico 2024/2025

Indice

Elenco delle figure

Capitolo 1

Introduzione

I ransomware si sono affermati come una delle minacce più impattanti per sistemi e organizzazioni, siccome compromettono la disponibilità dei file (cifrandoli) e possono anche esfiltrare informazioni. La velocità con cui il malware riesce a scansionare e criptare il file system rende difficile intervenire in tempo per fermarlo. Un attacco potrebbe essere effettuato nell'ordine dei minuti, perciò oltre alle tecniche di prevenzione e individuazione è importante avere strumenti capaci di mitigare un attacco in corso, riducendone l'efficacia e aumentando la finestra temporale utile per prendere provvedimenti.

Per questo motivo nasce Ranflood [4], un software pensato come una soluzione di facile installazione, alla stregua di un antivirus, basato sul Data Flooding against Ransomware (DFaR). L'idea alla radice è quella di aumentare il costo operativo del ransomware durante l'attacco introducendo rumore e competizione. Il rumore è dato dalla generazione di file esca e la loro distribuzione nel file system che rende difficile al malware individuare in modo efficiente i file di valore per l'utente, questa attività prende il nome di flooding. Per quanto riguarda la competizione, si intende il fatto che Ranflood, eseguendo concorrentemente operazioni di I/O, le contende assieme al ransomware, rubandogli risorse (come accessi al disco e CPU) rallentandone il throughput operazionale (lettura dei file, cifratura, ecc). Ranflood supporta diverse strategie di generazione di file ad esempio casuali o basate su copie di file, e adotta un'architettura orientata al parallelismo, scomponendo le attività in task eseguiti in parallelo per massimizzare la pressione sulle risorse e migliorare la tolleranza a errori locali (file non accessibili, permessi...).

Una strategia recente implementata ai fini di generare flooding e renderlo utile anche nella fase di ripristino, è la generazione di frammenti di file tramite Shamir Secret Sharing (SSS) [10], dove un dato viene diviso in n frammenti e solo una parte di essi (k) è necessaria per ricostruirlo. In questo modo anche se alcuni di

questi frammenti vengono cifrati o distrutti, il dato è recuperabile. Questa tecnica permette anche segretezza, ovvero avendo un sottoinsieme di frammenti minore di una soglia k, non si ha alcuna informazione utile riguardo il file originale. Questa caratteristica dà la possibilità al flooding basato su SSS di poter essere utilizzato per contrastare l'esfiltrazione. Tuttavia questa tecnica produce un overhead elevato, siccome i frammenti possono avere la dimensione nell'ordine di grandezza del file originale, rendendo il costo computazionale proporzionale al numero di frammenti generati.

Questa tesi propone di studiare l'uso degli erasure codes come alternativa più efficiente a SSS per ottenere una logica di generazione dei frammenti più efficiente. Gli erasure codes [24], ampiamente usati in sistemi di storage affidabili, permettono di controllare il livello di ridondanza, producendo frammenti tipicamente proporzionali a $\frac{file\_size}{k}$ e riducendo l'overhead complessivo rispetto a SSS. Una maggiore efficienza comporta un trade-off, a differenza di SSS, gli erasure codes non forniscono segretezza, di conseguenza, in scenari in cui la minaccia include l'esfiltrazione dei dati, risulta opportuno combinare codifica con cifratura.

L'obiettivo del lavoro è dunque analizzare come una strategia basata su erasure codes possa essere integrata nell'architettura di Ranflood e come si caratterizzano le prestazioni sotto diversi linguaggi e librerie.

1.1 Struttura della Tesi

Abbiamo iniziato introducendo i contenuti della tesi, quindi Ranflood ed erasure codes (CAPITOLO (1)). Passeremo a un capitolo incentrato sul background (CAPITOLO (2)) e le conoscenze necessarie per approcciarsi alle sezioni successive, parlando di come funziona Ranflood (SEZIONE (2.1)) e le varie tecniche al suo interno (SOTTOSEZIONE (2.1.1)) (SOTTOSEZIONE (2.1.2)), soffermandoci su SSS (SOTTOSEZIONE (2.1.3)) e il collegamento con gli erasure codes (SOTTOSEZIONE (2.1.4)). Introdurremo i concetti di campi finiti (SOTTOSEZIONE (2.2.1)), la costruzione dei campi estesi tramite polinomi irriducibili (SOTTOSEZIONE (2.2.2)) e le operazioni aritmetiche rilevanti (SOTTOSEZIONE (2.2.3)), poiché costituiscono il contesto matematico in cui operano molti codici di correzione e ricostruzione. Verrà poi presentata una panoramica sugli erasure codes Reed Solomon (SEZIONE(2.3)), definendone parametri (SOTTOSEZIONE (2.3.1)) e proprietà fondamentali (SOTTOSEZIONE (2.3.2)) e discutendo i meccanismi di codifica/decodifica tipici per codici Reed-Solomon (SOTTOSEZIONI (2.3.3) (2.3.4)), includendo approcci basati su matrici di Vandermonde (SOTTOSEZIONE (2.3.3)) e Cauchy (SOTTOSEZIONE

(2.3.3)), oltre a una variante accelerata che usa tecniche di tipo FFT (SOTTOSEZIONE (2.3.3)) (SEZIONE (4.2.3)). Sulla base di tali concetti, è stata proposta una bozza di integrazione teorica in Ranflood (CAPITOLO (3)), mostrando come gli erasure codes possano essere impiegati per generare flooding "utile" e, allo stesso tempo, mantenere la possibilità di ripristino a soglia k su n frammenti (SEZIONE (3.2)), resta tuttavia centrale il trade-off legato alla confidenzialità (SEZIONE (3.1)), che negli erasure codes non è intrinseca e richiede eventuali meccanismi aggiuntivi (come la cifratura) in scenari di exfiltration. Infine l'ultimo capitolo trattasi di una valutazione empirica su test effettuati su vari linguaggi e librerie (CAPITOLO (4)). Partendo da una breve spiegazione per cui mantenendo k fisso e aumentando i pacchetti di parità si abbia una curva con andamento $f(x) = \frac{1}{x}$ (SEZIONE (4.1)), per poi passare ai test in cui si controlla per ogni m il valore k che ne massimizza il throughput all'aumentare delle dimensioni del dato da codificare per ogni linguaggio e librerie scelti (SEZIONE (4.2)).

Capitolo 2

Background

In questo capitolo elencheremo le conoscenze necessarie per avere una visione sul "Data Flooding against Ransomware" e gli erasure codes. Facendo un'introduzione al software Ranflood [4] [3], aritmetica dei campi finiti [8] [6] ed erasure codes [24]. In particolare ci focalizzeremo sulla variante Reed Solomon [9] [25] (scelti in quanto rappresentano una delle soluzioni MDS (SOTTOSEZIONE (2.3.2)) più diffuse, mature e supportate da implementazioni ottimizzate in diversi linguaggi), sulla quale seguiranno molteplici test in linguaggi differenti (CAPITOLO (4)) per studiarne il comportamento su piattaforme e implementazioni differenti. Nello specifico inizieremo introducendo Ranflood, per poi passare a parlare dei campi finiti (o campi di Galois), che sono l'impalcatura algebrica sulla quale avvengono operazioni di codifica e decodifica del dato, infine ci focalizzeremo sulle specifiche di funzionamento di Reed Solomon.

2.1 Ranflood

Ranflood è uno strumento drop-in (installabile come un normale software funzionante in ogni macchina) pensato per mitigare gli attacchi effettuati soprattutto da crypto-ransomware. L'obiettivo non è tanto "bloccare" l'attacco quanto più rallentare l'azione del malware e aumentarne il costo dell'attacco, guadagnando tempo utile per l'utente per applicare contromisure (ad esempio terminando i processi malevoli, isolare la macchina, ecc...). Per permettere ciò, la logica di Ranflood si basa su un metodo chiamato DFaR (Data Flooding against Ransomware) che combina tre concetti:

1. **Moving Target Defence**: mischiando file reali e file esca, si rende più difficile il lavoro al malware per selezionare solo i dati di valore, in modo tale che esso vada a cifrare anche file inutili.

2. **Resource Contention**: il ciclo di flooding e il ransomware si contendono risorse di I/O su disco e CPU rallentandone l'azione.

3. **Dynamic Honeypot**: invece di usare file esca statici essi vengono generati dinamicamente e in posizioni scelte a runtime.

2.1.1 Strategie di Flooding

Ranflood supporta tre strategie principali per contrastare i ransomware:

Random : genera rapidamente file esca di svariate dimensioni ed estensioni con contenuto casuale, mirando soprattutto a una contenzione di I/O e confusione.

On-The-Fly (copy based) : crea copie dei file reali dell'utente sul momento appoggiandosi a uno snapshot preliminare leggero per evitare di copiare file già cifrati.

Shadow (copy based) : mantiene snapshot/archivi più pesanti sacrificando spazio sul disco per aumentare la probabilità di ripristino dei dati.

2.1.2 Architettura Software

Ranflood utilizza un modello client-daemon, dove il daemon, o demone, è un processo che gira in background e fa il lavoro di flooding, snapshot, gestione della strategia attiva, quali directory target ecc. Il client serve per inviare comandi al daemon come per esempio start, stop e configurazioni. In questo modo, il processo demone resta attivo anche se la sessione utente termina, la mitigazione può essere attivata da un detector esterno e infine il piano di controllo non diventa un bottleneck mentre il software sta praticando flooding, saturando disco e CPU. Dentro al demone avviene un processo di modularizzazione che permette di rendere ogni strategia un modulo flessibile, in modo tale da rendere semplice l'aggiunta di nuove strategie senza riscrivere tutta l'applicazione. Una parte importante è come Ranflood suddivide il lavoro. Esso non esegue un'operazione alla volta, ma spezza il lavoro in piccoli task eseguiti in parallelo tramite un Task Manager. Ciò permette di massimizzare le operazioni I/O, saturando la banda del disco e ostacolando il throughput di cifratura del ransomware, sfruttando hardware che permette di usare più core e parallelismo, mettendo pressione sulle risorse. Infine, se un task fallisce (es. file non accessibile), fallisce solo quel task senza bloccare l'intera operazione.

2.1.3 SSS (Shamir Secret Sharing)

In Ranflood è stata inserita una tecnica copy-based basata su Shamir Secret Sharing [10] dove, invece che creare copie intere o file esca casuali, vengono frammentati file reali in più parti e sfrutta la possibilità di ripristinare il file originale avendo un sottoinsieme di tali frammenti, ciò permette di ottenere resilienza (ripristino del dato anche se si perdono dei frammenti) sia contro cryptoransomware che contro exfiltration ransomware.

SSS è un meccanismo che permette di trasformare un segreto in n frammenti tali che:

- con almeno k frammenti puoi ricostruire il file originale.
- con meno di k frammenti non si ricavano informazioni utili sul contenuto.

Quindi SSS oltre alla ricostruzione del dato mira anche alla segretezza. In Ranflood viene utilizzato per due obiettivi principali:

- generare molti shards per generare flooding e contrastare il ransomware con file esca.
- se un attaccante ruba dati (exfiltration), SSS offre il vantaggio che se l'attaccante ruba meno di k shards, non riesce a ricostruire il contenuto del file originale.

Per raggiungere questi due obiettivi SSS viene tarato con parametri diversi mantenendo la stessa tecnica.

SSS-Ransomware Si sceglie un k basso per massimizzare la probabilità di ripristino del dato anche se molti shards vengono cifrati. Si può utilizzare anche un n alto per aumentare il flooding. Quindi l'idea generale è "creo molti shards e me ne basta un numero molto basso per ricostruire il segreto".

SSS-Exfiltration si sceglie un k più elevato per alzare la soglia di shards che un attaccante dovrebbe rubare per ricostruire il dato originale. Qui l'idea quindi è "rendere difficile ricostruire i dati rubati senza raggiungere la soglia k".

Per poter ripristinare il dato serve riconoscere quali shards appartengono al file originale, sapere che parametri (k, n) sono stati generati e verificare l'integrità. Per questo, ogni shard contiene metadati come: valori di n e k, un identificatore di generazione (per distinguere split successivi), un riferimento al dato originale, hash del dato originale e/o dello shard.

2.1.4 SSS ed Erasure Codes

Nelle prossime sezioni verranno spiegate l'aritmetica e il funzionamento di erasure codes per aggiungere un'implementazione teorica simile a SSS. Gli erasure codes permettono di ottenere la proprietà di ricostruire il dato originale partendo da un sottoinsieme di frammenti, tuttavia, a differenza di SSS, non forniscono una garanzia di segretezza, i frammenti possono comunque trapelare informazioni parziali sul contenuto del file. Di conseguenza, nel contesto di Ranflood, gli erasure codes sono interessanti come meccanismo di flooding contro cryptoransomware e rigenerazione del file partendo da un sottoinsieme di frammenti, riducendo overhead rispetto a schemi di copia completa o SSS con n elevati. Con SSS la dimensione totale dei frammenti tende a crescere linearmente con n. Con gli erasure codes la dimensione di ogni shard è fissa a $\frac{\text{data\_size}}{k}$. Dopo le sezioni sulla matematica di erasure coding e sul campo in cui lavorano, si passerà a mostrare come integrarli come strumento all'interno di Ranflood e di come sia possibile aggiungere una possibile implementazione per contrastare l'esfiltrazione di dati tramite la possibilità di combinare la cifratura con gli erasure codes.

2.2 Aritmetica dei Campi Finiti

L'aritmetica dei campi finiti [8] [6], noti anche come campi di Galois, costituisce la base matematica e algebrica su cui si basano gli erasure codes, nel nostro caso specifico, per i codici Reed Solomon. La loro peculiarità è la fornitura di una struttura algebrica chiusa nella quale è possibile effettuare operazioni di somma, prodotto e inversione, garantendo che il risultato delle operazioni appartenga ancora al campo (proprietà di chiusura) e possiedano un inverso moltiplicativo.

2.2.1 Definizione Algebrica di Campo Finito

Un campo è una struttura algebrica dotata di due operazioni, ovvero somma e moltiplicazione, che soddisfano proprietà di chiusura, associatività, commutatività ed esistenza di un elemento neutro e dell'inverso. L'inverso esiste sempre per la somma mentre per la moltiplicazione esiste per tutti gli elementi eccetto lo 0. Quando il numero di elementi che compongono un campo è finito si parla di campo finito. Un teorema fondamentale dell'algebra afferma che un campo finito di elementi q esiste se e solo se: $q = p^w$, dove p è un numero primo e w un numero intero positivo. Inoltre, per ogni $q = p^w$ esiste un campo finito con q elementi ed è unico a meno di isomorfismo: questo risultato, legato alla teoria di Galois, è all'origine della

denominazione campi di Galois, e tali campi si indicano comunemente con $GF(q)$ oppure $\mathbb{F}_q$. Quando $w = 1$ si parla di campi primi nei quali le operazioni sono eseguite modulo p, la primalità di p garantisce che ogni elemento non nullo ha il proprio inverso, quando $w > 1$ si parla di campi di estensione. Nelle applicazioni digitali, si prediligono campi con $p = 2$, cioè $GF(2^w)$ siccome gli elementi verranno rappresentati come polinomi di grado $< w$ con coefficienti in $GF(2)$, che possono essere rappresentati come stringhe di bit. In questi campi, l'operazione di addizione è equivalente a uno XOR, annullando l'operazione di riporto e garantendo un'efficienza computazionale maggiore. I campi più utilizzati in ambito di erasure coding sono i $GF(2^8)$ (che si adattano perfettamente al byte) e $GF(2^{16})$.

2.2.2 Costruzione tramite Polinomio Irriducibile

Quando $w > 1$, il campo $GF(p^w)$ non può essere costruito semplicemente effettuando le operazioni modulo p^w, poiché la struttura risultante non sarebbe un campo (non tutti gli elementi non nulli ammetterebbero inverso moltiplicativo). È quindi necessario ricorrere a una costruzione più sofisticata basata sui polinomi [27].

Si consideri l'anello dei polinomi a coefficienti nel campo primo $\mathbb{F}_p$, indicato con $\mathbb{F}_p[x]$. Un campo di estensione $GF(p^w)$ viene ottenuto come quoziente:

$$GF(p^w) = \mathbb{F}_p[x]/(f(x))$$

dove $f(x)$ è un polinomio irriducibile di grado w su $\mathbb{F}_p$.

Un polinomio si dice *irriducibile* se non può essere fattorizzato come prodotto di due polinomi di grado minore con coefficienti nello stesso campo. Questa proprietà è essenziale: l'irriducibilità garantisce che $f(x)$ non possa essere scomposto in fattori più semplici assicurando così l'esistenza dell'inverso moltiplicativo essenziale per praticare divisioni nel momento della decodifica dei dati.

Gli elementi di $GF(p^w)$ sono quindi le classi di equivalenza dei polinomi di grado minore di w con coefficienti in $\mathbb{F}_p$. Ogni elemento del campo può essere rappresentato come:

$$a_{w-1}x^{w-1} + \cdots + a_1 x + a_0$$

con $a_i \in \mathbb{F}_p$.

Esempio: costruzione di $GF(2^3)$ Si consideri $\mathbb{F}_2 = \{0, 1\}$. Un polinomio irriducibile di grado 3 su $\mathbb{F}_2$ è:

$$f(x) = x^3 + x + 1$$

Il campo $GF(2^3)$ è quindi definito come:

$$\mathbb{F}_2[x]/(x^3 + x + 1)$$

Gli elementi del campo sono tutti i polinomi di grado inferiore a 3:

$$\{0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1\}$$

Poiché il polinomio scelto ha grado 3, il numero totale di elementi è $2^3 = 8$.
Nel campo così costruito vale la relazione fondamentale:

$$x^3 \equiv x + 1 \pmod{x^3 + x + 1}$$

che deriva direttamente dalla definizione del quoziente. Tale relazione consente di ridurre ogni polinomio di grado superiore o uguale a 3 a un polinomio di grado inferiore a 3, garantendo la chiusura delle operazioni all'interno del campo.

La scelta del polinomio irriducibile non è unica: esistono più polinomi irriducibili di grado w, ma tutti generano campi isomorfi tra loro, ovvero strutturalmente equivalenti dal punto di vista algebrico.

2.2.3 Operazioni nel campo $GF(2^w)$

Una volta costruito il campo $GF(2^w)$ tramite polinomio irriducibile di grado w, le moltiplicazioni e le addizioni assumono una forma particolarmente efficiente dal punto di vista computazionale, ciò ne ha portato un grosso utilizzo nei sistemi informatici.

- **Addizione** Nel campo $GF(2^w)$ gli elementi sono polinomi di grado $< w$ con coefficienti in $\mathbb{F}_2 = \{0, 1\}$. L'addizione si effettua coefficiente per coefficiente modulo 2. Dato che in modulo 2 vale $0 + 0 = 0$, $1 + 0 = 1$ e $1 + 1 = 0$, l'addizione equivale a uno XOR bit a bit.

  Esempio in $GF(2^3)$ $\quad a(x) = x^2 + x \quad b(x) = x^2 + 1$

  $a(x) + b(x) = (x^2 + x) + (x^2 + 1)$ i termini x^2 si annullano siccome $1 + 1 = 0$ avendo come risultato $x+1$. $a(x)$ in forma binaria è 110 e $b(x)$ 101 e lo XOR dà esattamente 011, che corrisponde a $x + 1$. L'assenza del riporto rende questa operazione estremamente efficiente a livello hardware e software.

- **Moltiplicazione** La moltiplicazione si compone in due fasi, ovvero la moltiplicazione polinomiale ordinaria e la riduzione usando come modulo il polinomio irriducibile scelto per costruire il campo. Riprendendo l'esempio di prima con polinomio irriducibile $f(x) = x^3+x+1$ e considerando la moltiplicazione tra x^2 e x che produce x^3, poiché nel campo in cui si lavora con $f(x)$ vale la relazione $x^3 \equiv x + 1$, di conseguenza $x \cdot x^2 = x + 1$

Un aspetto importante da ricordare riguarda il fatto che ogni elemento non nullo del campo ammette un inverso moltiplicativo. Esso può essere calcolato tramite l'algoritmo di Euclide oppure sfruttando la proprietà: $a^{-1} \equiv a^{2^w-2}$. L'esistenza dell'inverso è fondamentale per la fase di decodifica del dato siccome richiede l'inversione di matrici definite sul campo.

2.3 Codici Reed-Solomon

I codici Reed-Solomon (RS) [24] [9] [25] rappresentano una delle famiglie più importanti e diffuse di codici a correzione d'errore (Forward Error Correction, FEC) e appartengono alla classe dei codici a blocchi lineari, ciclici e non binari. Introdotti nel 1960 da Irving S. Reed e Gustave Solomon, questi codici sfruttano appieno l'aritmetica dei campi finiti $GF(p^w)$ precedentemente descritta, operando su "simboli" piuttosto che su singoli bit. Nel contesto informatico, si utilizza tipicamente il campo $GF(2^8)$, permettendo a ogni simbolo di corrispondere esattamente a un byte.

2.3.1 Definizione e Parametri Fondamentali

Un codice Reed-Solomon è un codice a blocchi sistematico, tipicamente denotato con $RS(n, k)$, in cui le variabili definiscono com'è strutturato il blocco da codificare:

- k rappresenta il numero di simboli di informazione originari (il *payload* del dato).
- n rappresenta il numero totale di simboli che compongono la parola di codice (*codeword*) al termine della codifica.
- $m = n - k$ indica il numero di simboli di parità (o ridondanza) aggiunti matematicamente.

Poiché i codici operano su un campo finito specifico, la lunghezza del blocco n è legata alla dimensione del campo stesso, seguendo la regola $n \leq q - 1$. Dove

$q = |GF(q)| = p^w$ Per $GF(2^8), q = 256$, la lunghezza massima del codeword è $n \leq 255$.

2.3.2 Proprietà MDS e Gestione delle Cancellazioni (*Erasures*)

La caratteristica fondamentale che rende i codici Reed-Solomon eccezionalmente adatti per i moderni sistemi di archiviazione distribuiti e per la mitigazione dei ransomware è la loro natura di codici **MDS** (*Maximum Distance Separable*). Un codice MDS garantisce che data la perdita di qualunque degli n pacchetti, avendo un sottoinsieme qualsiasi di almeno k elementi, il dato è ricostruibile.

Questa proprietà si traduce in capacità di ripristino ottimali, il cui comportamento varia a seconda del tipo di corruzione subita dal dato:

- **Errori (*Errors*):** Si verificano quando il valore di un simbolo viene alterato e la sua posizione all'interno del blocco non è nota al decodificatore. In questo caso, il codice RS può individuare e correggere fino a t simboli errati, dove $2t = m$.
- **Cancellazioni (*Erasures*):** Si verificano quando il dato è illeggibile o mancante, ma il sistema ne conosce a priori la posizione esatta (ad esempio, un settore del disco danneggiato o un file crittografato da un ransomware di cui si conosce l'identità). In scenari di erasure coding, il codice esprime la sua massima efficienza, potendo ricostruire esattamente il dato perdendo fino a m simboli.

In pratica, questa proprietà è estremamente potente: per recuperare l'intero blocco di dati originale, è sufficiente che il sistema abbia accesso a un sottoinsieme qualsiasi di k simboli integri, indipendentemente dal fatto che essi siano simboli di informazione originari o di parità.

2.3.3 Codifica

La fase di codifica di Reed Solomon consiste nel trasformare un insieme di k simboli di informazione in una codeword di dimensione n ottenendo $m = n-k$ simboli di parità calcolati algebricamente. A differenza di altre forme di ridondanza, i simboli aggiuntivi derivano da una struttura matematica precisa basata su polinomi e campi finiti. I simboli di dati $(d_0, d_1 \ldots, d_{k-1})$ vengono interpretati come i coefficienti di un polinomio di grado strettamente inferiore a k: $P(x) = d_0 + d_1 x + d_2 x^2 + \cdots + d_{k-1} x^{k-1}$. La codeword si ottiene valutando questo polinomio in n punti distinti appartenenti

al campo ($\alpha_1, \alpha_2, \ldots, \alpha_n \in GF(2^w)$) così da avere una codifica per ogni punto data da $c_i = P(\alpha_i)$ con $i = 1, \ldots, n$. Perciò il messaggio viene proiettato in più punti del piano rispetto a quelli che servirebbero per ricostruirlo. Questa sovradeterminazione è proprio ciò che rende possibile la ricostruzione del messaggio in presenza di perdite o corruzione di simboli.

Matrice di Vandermonde

La costruzione polinomiale può essere riscritta in forma matriciale detta di Vandermonde [28] definendo i dati con un vettore: $d = (d_0, d_1, \ldots, d_{k-1})^T$ il vettore di simboli codificati come $c = (c_1, c_2, \ldots, c_n)^T$ e la codifica come la moltiplicazione matrice vettore: $c = Gd$, dove G è una matrice di dimensione $n \times k$ in cui la $i-$esima riga è data da: $(1, \alpha_i, \alpha_i^2, \ldots, \alpha_i^{k-1})$ con α_i tutti distinti nel campo. Questa struttura garantisce che qualsiasi sottoinsieme di G formato da k righe sia linearmente indipendente, assicurando la proprietà MDS.

Matrice di Cauchy

Una variante di Vandermonde, spesso utilizzata siccome presenta vantaggi computazionali, è la matrice di Cauchy [26] [22] [16], dove gli elementi sono costruiti nella forma: $G_{ij} = \frac{1}{x_i - y_j}$ con x_i e y_j appartenenti a due insiemi $X = \{x_0, \ldots, x_{m-1}\}$ (uno per ogni riga di parità) e $Y = \{y_0, \ldots, y_{k-1}\}$ (uno per ogni colonna) con le proprietà:

1. tutti gli x_i distinti tra loro.
2. tutti gli y_i distinti tra loro.
3. $X \cap Y = \emptyset$.

L'ultima condizione serve per evitare divisioni per 0. La matrice di Cauchy ha la proprietà MDS. La matrice C di Cauchy ha dimensione $m \times k$ e la parità è calcolata come: $p_i = \sum_{j=0}^{k-1} C_{ij} \cdot d_j$.

Differenze tra Cauchy e Vandermonde

Nelle implementazioni con matrice di Vandermonde, le operazioni sono moltiplicazioni generiche in $GF(2^w)$, che sono più costose di semplici XOR. Le varianti Cauchy Reed Solomon trasformano la matrice generatrice in una bitmatrix su $GF(2)$ così da esprimere la generazione dei simboli di parità mediante combinazioni di operazioni XOR. Questa rappresentazione consente di ridurre il costo computazionale

della codifica, poiché le XOR sono in genere più efficienti delle moltiplicazioni nel campo esteso. La scelta di X e Y permette di ottimizzare il numero di 1 nella bitmatrix e di conseguenza ridurre il numero di operazioni.

Fast Fourier Transformation

Come visto in precedenza, il calcolo della codifica viene eseguito punto per punto, calcolando $P(\alpha_i)$ per ciascun punto. Tramite l'utilizzo della *Fast Fourier Transform* (FFT) [14] [15] su campi finiti, invece, è possibile accelerare in modo significativo questa fase. Invece di utilizzare un FFT classica si utilizza una additive FFT definita su campi binari estesi (come $GF(2^w)$). In questo contesto i punti di valutazione vengono organizzati come elementi appartenenti a somme di sottospazi vettoriali del campo. Il polinomio viene rappresentato in una base polinomiale costruita tramite i subspace polynomials.

Questa scelta consente di applicare una trasformata ricorsiva del tipo divide-et-impera capace di calcolare simultaneamente molteplici valutazioni del polinomio con complessità quasi lineare. Applicando questa struttura ai codici Reed-Solomon, si ottiene una codifica con complessità $O(n \log k)$. Per parametri ridotti gli approcci matriciali rimangono pienamente adeguati, al crescere delle dimensioni dei file, l'uso di trasformate veloci su campi finiti diventa essenziale per ridurne il costo computazionale.

2.3.4 Decodifica

La decodifica per i metodi matriciali funziona allo stesso modo sia per Vandermonde che per Cauchy, ovvero si prendono k blocchi integri (sottoinsieme qualsiasi di n) e si risolve il sistema lineare: $G'd = c$ cioè $d = (G')^{-1}c'$ dove G' è la sottomatrice di G di dimensione $k \times k$ e c' sono i simboli ricevuti.

Nel caso della additive FFT, invece, non viene eseguita tramite l'inversione della matrice, ma come problema di ricostruzione polinomiale su campi finiti. Si costruisce un error locator polynomial, ossia un polinomio che si annulla esattamente nelle posizioni corrispondenti ai simboli mancanti. A partire dai simboli ricevuti e da tale polinomio, viene formato un polinomio ausiliario su cui si applicano una trasformata inversa (IFFT), il calcolo della derivata formale e una successiva trasformata diretta (FFT). I valori mancanti vengono infine ricavati combinando il risultato ottenuto con la derivata dell'error locator polynomial. Questo approccio consente di ottenere una procedura di decodifica con complessità $O(n \log n)$, vantaggiosa con numero di shard e dimensione dei dati elevati.

Capitolo 3

Erasure Codes per Data Flooding against Ransomware

In questa sezione viene descritta l'architettura concettuale di erasure codes in Ranflood, con l'obiettivo di introdurre un'alternativa a SSS. L'idea è di mantenere la logica discussa in SSS, ovvero la generazione di frammenti di un dato e la possibilità di ripristino di esso partendo da un sottoinsieme di tali frammenti, riducendo overhead di storage e migliorando le prestazioni, grazie al fatto che, negli erasure codes, i frammenti hanno dimensione proporzionale a $\frac{file\_size}{k}$, a differenza di SSS in cui la dimensione del frammento ha approssimativamente la stessa dimensione del file. Questi vantaggi introducono però un trade-off rispetto a SSS, mentre quest'ultimo garantisce una proprietà di segretezza (cioè con meno di k frammenti non si ottengono informazioni utili sul contenuto), gli erasure codes sono nati principalmente per affidabilità e tolleranza alla perdita di shards. Anche utilizzando codifiche non sistematiche, un sottoinsieme di frammenti (minore di k) può trapelare informazioni parziali sul dato. Di conseguenza questo approccio, per essere utilizzato in scenari in cui la minaccia includa anche exfiltration, deve essere affiancato da un meccanismo di cifratura.

3.1 Obiettivi

Flooding : generare un numero elevato di frammenti consente di aumentare la pressione sulla contenzione delle operazioni I/O con il malware, spingendo il ransomware a sprecare tempo su artefatti che non corrispondono al file originale.

Ripristino del dato : grazie alla proprietà MDS degli erasure codes il dato originale è ripristinabile anche se una parte dei frammenti è stata cifrata, cancellata o resa inaccessibile. L'importante è che almeno k degli n frammenti totali siano integri.

Exfiltration : per contrastare questa tipologia di attacchi, una strategia potrebbe essere quella di combinare cifratura e codifica secondo lo schema Encrypt-then-encode, ovvero prima si cifra il file, ottenendo un ciphertext per poi codificarlo con erasure coding. Nella fase di ripristino del file le operazioni vengono invertite (Decode-then-decrypt).

3.2 Integrazione in Ranflood

Dal punto di vista architetturale, l'estensione si inserirebbe senza modificare la struttura di Ranflood client-daemon e il task manager. Cambia semplicemente il tipo di lavoro generato dalla strategia. Al posto di task copy based o SSS, vengono aggiunti:

- **Task di codifica**: lettura del file, (cifratura opzionale) suddivisione in blocchi e generazione dei n shards codificati con scrittura concorrente dei frammenti su disco.
- **Task di ripristino**: raccolta e verifica dei frammenti, decodifica del file originale (e decifratura nel caso fosse stata applicata in precedenza)

In questo modo, si mantengono i vantaggi del modello di Ranflood quali: parallelismo, robustezza a errori locali e massima pressione sulle risorse I/O.

3.3 Informazioni all'Interno degli Shards

Nella fase di ripristino sono necessarie alcune informazioni che devono essere aggiunte ad ogni shard nella fase di codifica del dato. Ogni shard deve avere:

- **File_id**: utile per capire quali shards appartengono allo stesso file.

- **Generation_id**: distingue split dello stesso file diversi nel tempo, in altre parole, evita di mischiare shards di split differenti.
- **Shard_index**: l'indice dello shard utile al decoder per sapere quali simboli ha.
- **k, n**: i valori di codifica, per sapere la soglia e il totale di shards.
- **Data_size**: dimensione del file originale, per tagliare eventuali padding in fase di ricostruzione.
- **Shard_hash**: utile per differenziare gli shards corrotti da quelli utilizzabili.
- **original_file_hash**: per verificare la correttezza finale con il file originale, una volta ricostruito il file corrotto.

In caso sia stata effettuata un cifratura del file, all'interno degli shards è utile salvare anche l'algoritmo utilizzato, il nonce (valore unico usato per quella determinata cifratura) e infine la chiave di decifratura cifrata con una master key salvata in un luogo sicuro.

Capitolo 4

Valutazione Empirica

Dopo aver visto come utilizzare gli erasure codes all'interno di Ranflood passiamo alla valutazione empirica di diverse implementazioni in: Rust, Go, Python. Gli esperimenti sono stati effettuati su una macchina con montato un processore Intel(R) Core(TM) i3-10110U CPU @ 2.10GHz x86_64, 8GB di RAM DDR4 2667 MT/s (1333 MHz) con sistema operativo Linux. Lo scopo di questi test è analizzare le prestazioni computazionali delle diverse implementazioni di erasure coding al fine di:

- visualizzare l'andamento del throughput generale fissando k (data shards) e variando m (parity shards)

- trovare, al variare di m , il k che massimizza il throughput.

Per queste prove, un vincolo teorico è dato dal campo di Galois utilizzato. La maggior parte delle librerie analizzate opera su GF(2^8) (o GF(256)) poiché questa dimensione permette di manipolare i dati direttamente a livello di singoli byte, garantendo un'elevata efficienza computazionale. L'uso di questo campo, però, implica che il numero totale di shards generati ($n = k + m$) non può eccedere la dimensione del campo stesso. Pertanto ogni test è configurato in modo che la somma di k e m sia sempre minore o uguale a 255. Un ulteriore vincolo aggiunto è quello che k sia $\leq m$. Nonostante le librerie di erasure coding permettano configurazioni con shards di parità inferiori agli shards di dati per permettere che lo spazio di archiviazione sia ottimizzato, nel nostro contesto questa logica viene invertita per permettere di rendere efficace il flooding, affinché possano essere creati shards esca per il ransomware, costretto a sprecare cicli di calcolo e accessi al disco per cifrare file inutili, garantendo così all'utente il tempo necessario per intervenire e interrompere l'attacco. Nel caso il ransomware abbia criptato alcuni shards k e altri m, se gli shards

totali sono comunque $\geq k$ il file è recuperabile grazie alla proprietà MDS dei codici Reed-Solomon.

4.1 Andamento del Throughput Fissando k

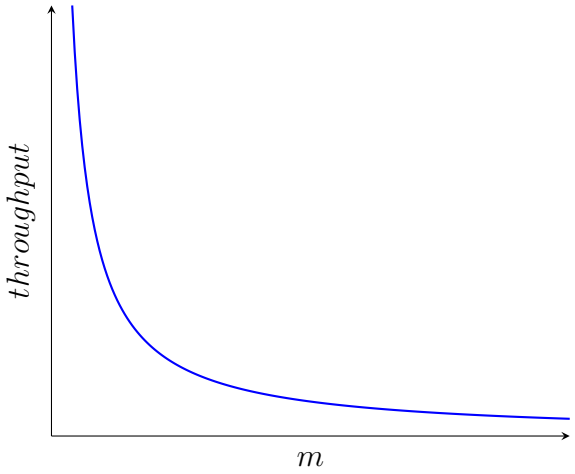


Figura 4.1: Rappresentazione del decadimento del throughput all'aumentare di m

Nei codici Reed-Solomon mantenendo k fisso il throughput decade proporzionalmente inverso a m. Questo andamento è dovuto alla complessità algoritmica di codifica $O(k * m)$ di Reed-Solomon siccome all'aumentare di m il numero di operazioni (moltiplicazioni e XOR nei campi di Galois) cresce linearmente il che è conseguenza di una matrice generatrice di dimensione maggiore all'aumentare di m che verrà moltiplicata per i k dati per generare la ridondanza richiesta. Perciò, il tempo, a parità di risorse hardware, aumenta circa in modo lineare e il throughput decresce approssimativamente secondo la funzione $f(m) = \frac{1}{m}$.

4.2 Analisi del Valore Ottimale di k al Variare di m

In questa sezione si studia come il parametro k (numero di data shard) influenzi le prestazioni di codifica al variare di m (numero di parity shard), con l'obiettivo di determinare, per ogni valore di m, il valore $k^*(m)$ che massimizza il throughput di encoding per ogni implementazione.

4.2.1 Impostazione sperimentale

Come anticipato in precedenza i test sono stati effettuati esplorando tutte le coppie ammissibili di k e m tali che la loro somma sia inferiore a 256 e $k \leq m$. Per

ogni coppia (k, m) è stato misurato il tempo di codifica t ripetendo l'operazione per $R = 30$ rounds. Il tempo misurato include la preparazione del buffer (shardizzazione dei dati), inizializzazione dell'encoder, eventuale allocazione degli shards di parità e l'esecuzione della codifica. Dai tempi raccolti si è calcolato il tempo medio $\bar{t}(k, m)$ e il throughput definito come $T(k, m) = \frac{data\_size}{\bar{t}(k,m)}$. Successivamente per ogni valore di m è stato selezionato il k che massimizza il throughput per poi rappresentare graficamente il risultato per ogni size testata (1KiB, 10KiB, 50KiB, 100KiB, 500KiB, 1MiB, 2MiB, 4MiB).

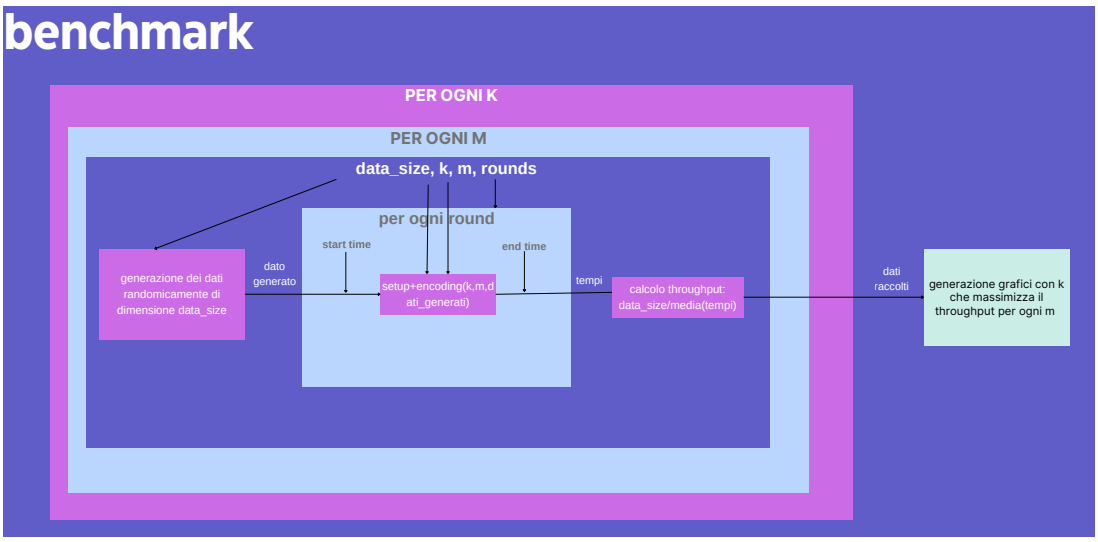


Figura 4.2: Schema dello svolgimento dei test

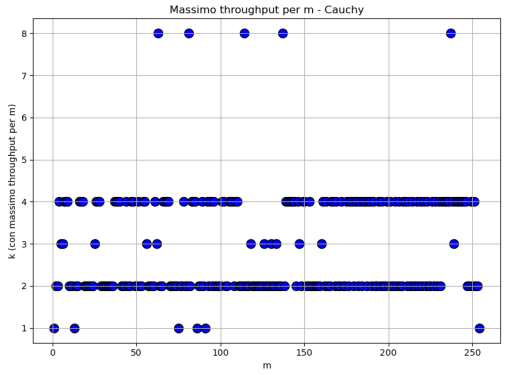
4.2.2 Go

In questa sezione tratteremo l'implementazione su Go utilizzando la libreria *klauspost/reedsolomon* [13], che rappresenta una delle implementazioni più performanti di erasure coding Reed-Solomon in Go. Originariamente era semplicemente una traduzione della libreria *JavaReedSolomon* di *Backblaze* [2], per poi essere arricchita con molteplici ottimizzazioni che la rendono ideale per sistemi di archiviazione ad alte prestazioni. Le ottimizzazioni in questione sono:

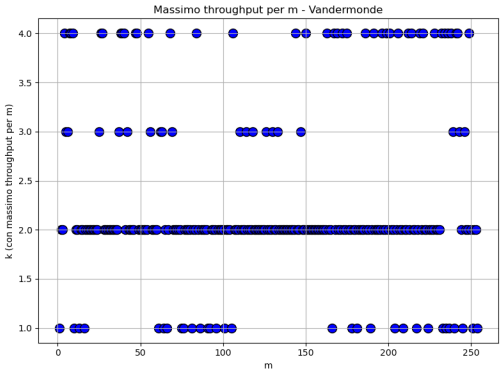
- Accelerazioni hardware: utilizza SIMD (Single Instruction Multiple Data) (SSE3, AVX2 e AVX-512 per x86/64 e Neon per ARM) le quali sono scritte direttamente in assembly per velocizzare i calcoli nel campo di Galois.
- Assenza di CGO: nonostante l'assembly la libreria rimane Go puro senza dipendenze esterne C garantendo una facile portabilità e compilazione.

- Parallelismo: Gestisce automaticamente l'uso delle goroutines per distribuire il carico su più core ottimizzando i tempi di esecuzione.

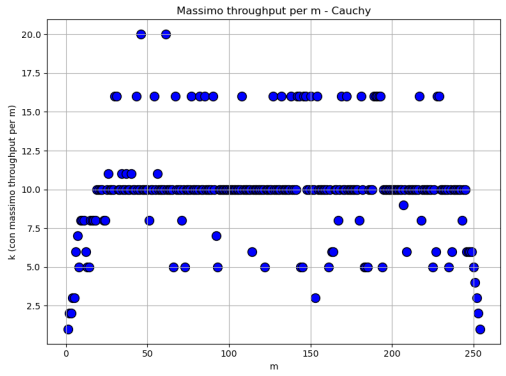
La libreria *klauspost/reedsolomon* permette di scegliere tra implementazione con matrice di Vandermonde o di Cauchy (che analizzeremo nei grafici sottostanti).



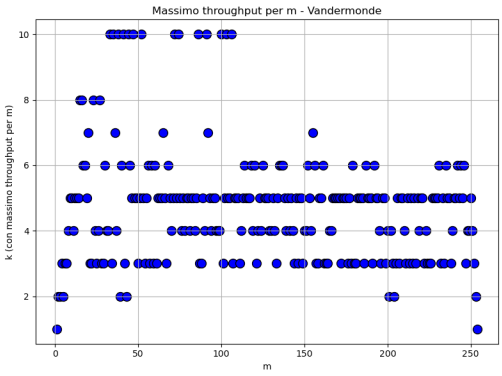
Cauchy data_size = 1KiB



Vandermonde data_size = 1KiB

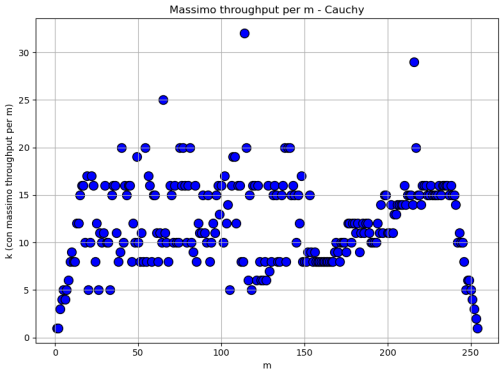


Cauchy data_size = 10KiB

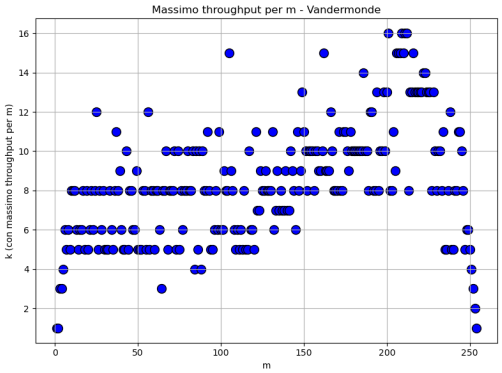


Vandermonde data_size = 10KiB

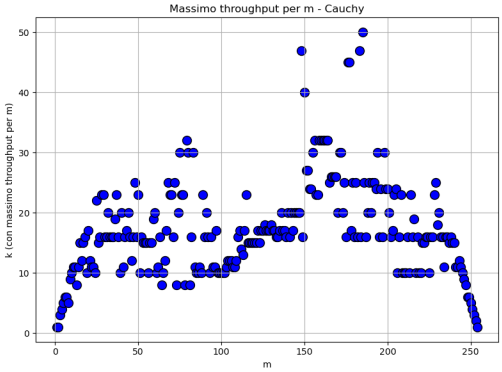
Figura 4.3: Plot per k che massimizza il throughput per ogni m dove $data\_size$ = 1KiB, 10KiB



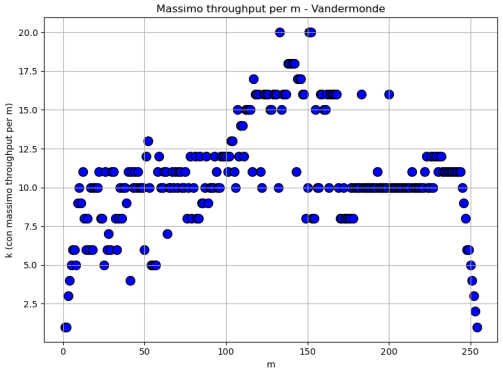
Cauchy data_size = 50KiB



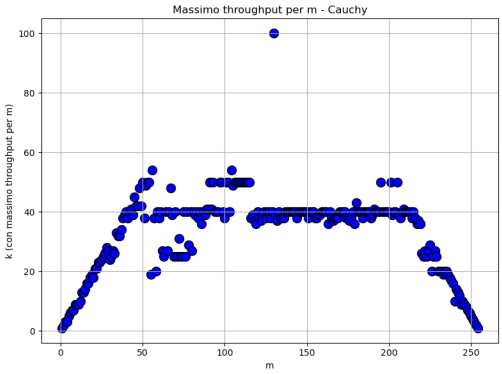
Vandermonde data_size = 50KiB



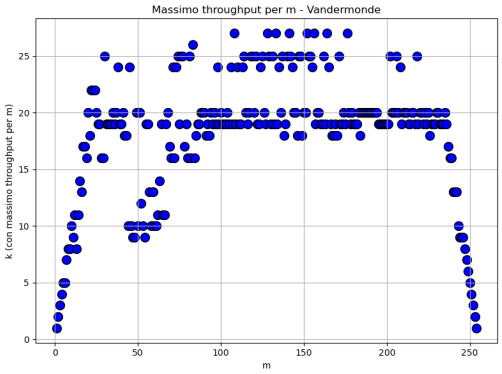
Cauchy data_size = 100KiB



Vandermonde data_size = 100KiB

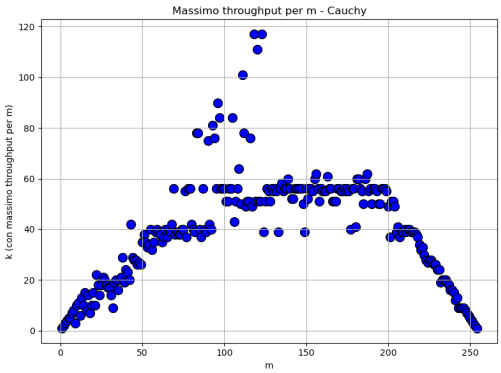


Cauchy data_size = 500KiB

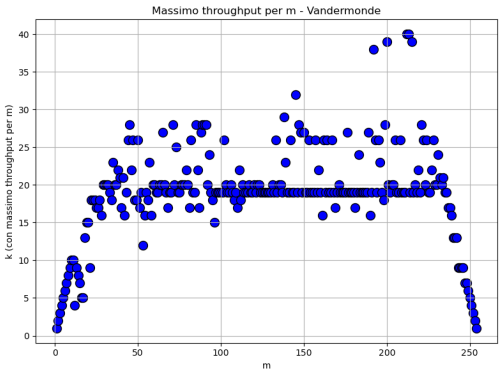


Vandermonde data_size = 500KiB

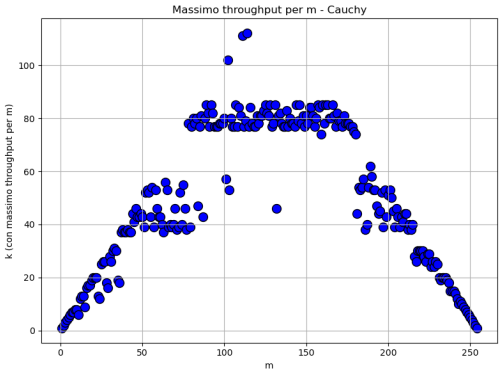
Figura 4.4: Plot per k che massimizza il throughput per ogni m dove $data\_size$ = 50KiB, 100KiB, 500KiB



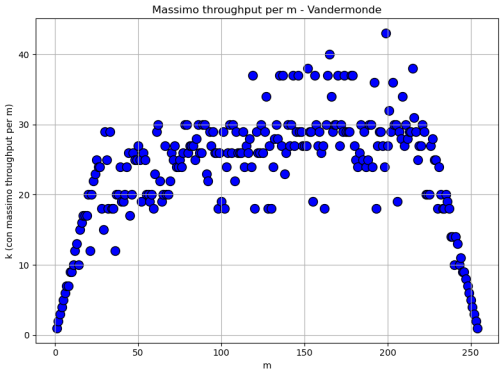
Cauchy data_size = 1MiB



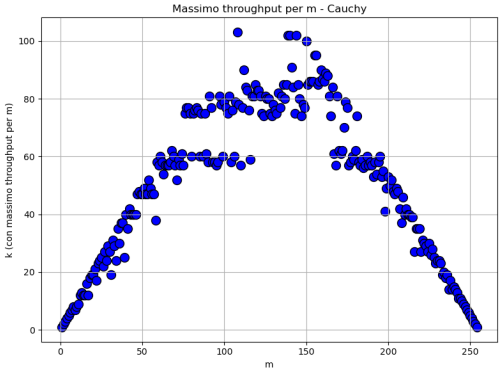
Vandermonde data_size = 1MiB



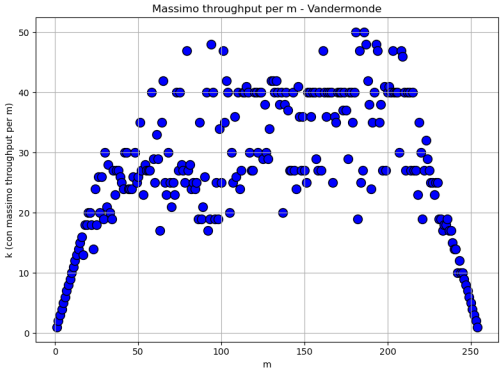
Cauchy data_size = 2MiB



Vandermonde data_size = 2MiB



Cauchy data_size = 4MiB



Vandermonde data_size = 4MiB

Figura 4.5: Plot per k che massimizza il throughput per ogni m dove $data\_size$ = 1MiB, 2MiB, 4MiB

Osservazioni sui Grafici

I grafici mostrano come la dimensione del dato da codificare influenzi significativamente la scelta di k che massimizza il throughput.

Per $data\_size$ piccoli, il k che massimizza il throughput rimane su valori bassi, poiché i costi fissi e per shard (allocazioni, preparazione dei buffer e l'inizializzazione dell'encoder) incidono sul tempo totale: perciò aumentare k riduce la dimensione degli shards (shard_size = $\frac{data\_size}{k}$) e può rendere meno efficiente per esempio le ottimizzazioni SIMD, oltre ad aumentare l'overhead di gestione.

Per dimensioni del dato pari a 1KiB, 10KiB, 50KiB (Figures 4.3) (Figures 4.4) si osservano quindi valori bassi di k.

All'aumentare della dimensione (a partire da 100KiB (Figures 4.4)(Figures 4.5)) si nota un cambiamento: i grafici assumono una forma più regolare, nel caso di Cauchy l'andamento si avvicina a una forma a campana, mentre in Vandermonde i k tendono a concentrarsi in una fascia relativamente stabile di valori. Entrambe le implementazioni presentano la presenza di gradini in alcune zone. In questo caso la fase di encoding tende a dominare il tempo totale mentre i costi fissi vengono progressivamente ammortizzati. I gradini osservati possono essere ricondotti a soglie implementative interne alla libreria, legate alla scelta dei kernel SIMD, al numero di shard e alla dimensione degli shard, per alcuni intervalli i k si stabilizzano e risultano più favorevoli rispetto ai valori adiacenti.

Per quanto riguarda la differenza tra matrici di Cauchy e Vandermonde, a parità di data_size i grafici mostrano che con la matrice di Cauchy i valori ottimali di k risultano più elevati rispetto a Vandermonde (Figure 4.5e)(Figure 4.5f).

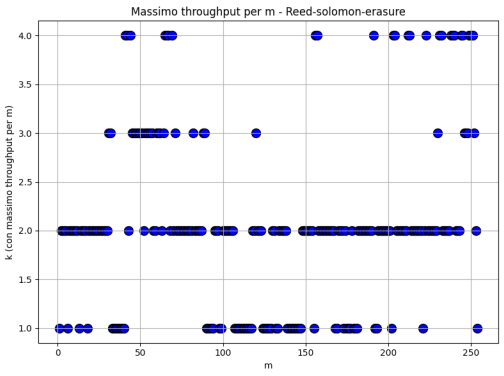
Entrambe le matrici sono MDS su GF(2^8), pertanto le differenze prestazionali osservate sono frutto di aspetti implementativi come il costo di inizializzazione e precomputazione dell'encoder, generazione dei coefficienti e interazione con kernel SIMD e Lookup Tables. Siccome il benchmark include tutti i passaggi della codifica, i risultati suggeriscono che, nell'implementazione considerata, la variante Cauchy risulti mediamente più favorevole dal punto di vista prestazionale rispetto a Vandermonde [22] [16] raggiungendo throughput medi più elevati (nell'ordine di decine di MiB/s in funzione di k, m, $data\_size$).

4.2.3 Rust

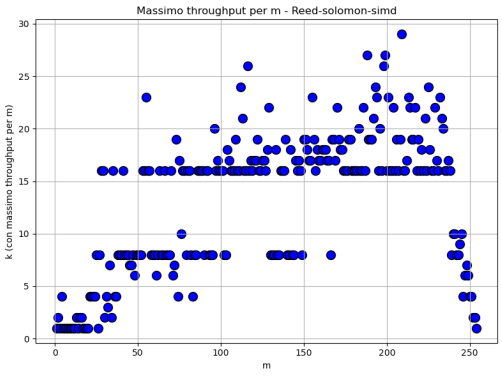
In questa sezione tratteremo l'implementazione su Rust utilizzando le librerie *reed-solomon-erasure* [23] e *reed-solomon-simd* [1] le quali presentano alcune differenze sostanziali.

- *reed-solomon-erasure* è un'implementazione pura in Rust che segue la struttura della libreria *JavaReedSolomon* di *Backblaze* per le versioni antecedenti la 2.0.0, dalle versioni successive si segue l'approccio di *klauspost/reedsolomon* con aggiunta la possibilità di attivare accelerazioni SIMD scritte in C (se specificato) tramite funzionalità simd-accel la quale è ottimizzata per processori (x86-64) Intel di generazione Haswell e successive e Arm 64 bit. Per architetture diverse è possibile forzare la compilazione del codice C su un target specifico. Questa libreria lavora su GF(2^8) e se si lavora con un maggior numero di shards è possibile passare a un campo GF(2^{16}).

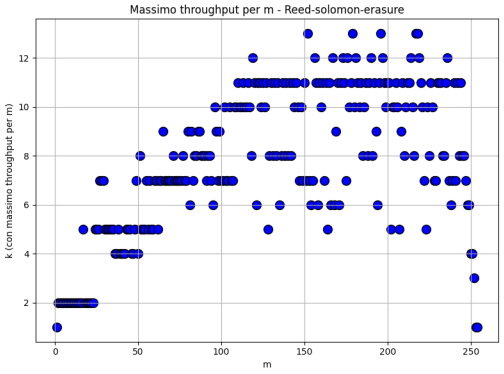
- *reed-solomon-simd* questa libreria si basa sull'algoritmo Leopard-RS [5], quindi utilizza algoritmi basati sulla trasformata di Fourier veloce (FFT), lavora su un campo GF(2^{16}), ha accelerazioni SIMD per estensioni AVX2 e SSSE3 per processori x86-64 e Neon per architetture Arm 64, in caso non siano disponibili le accelerazioni hardware è previsto un fallback automatico su codice Rust puro.



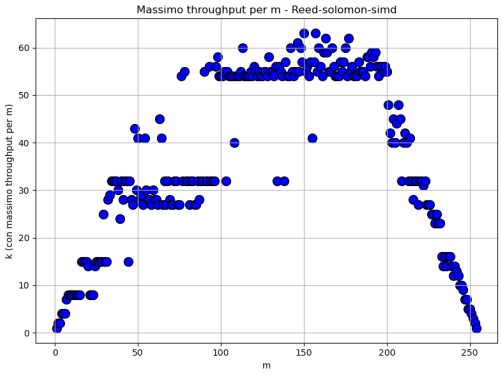
reed-solomon-erasure data_size = 1KiB



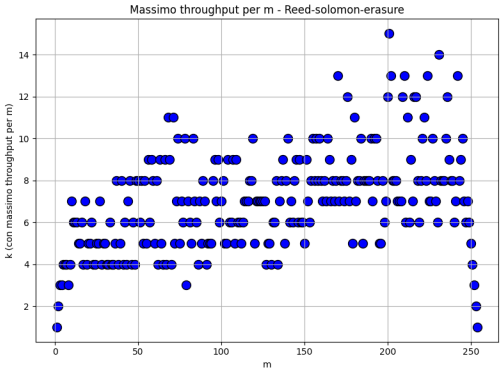
reed-solomon-simd data_size = 1KiB



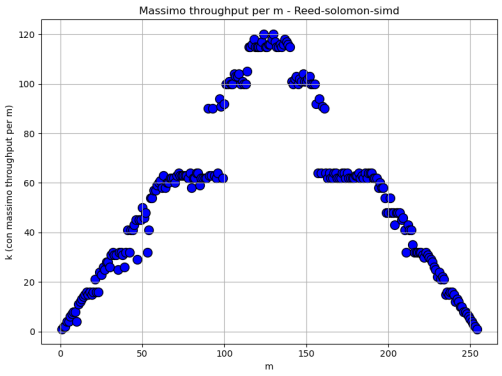
reed-solomon-erasure data_size = 10KiB



reed-solomon-simd data_size = 10KiB

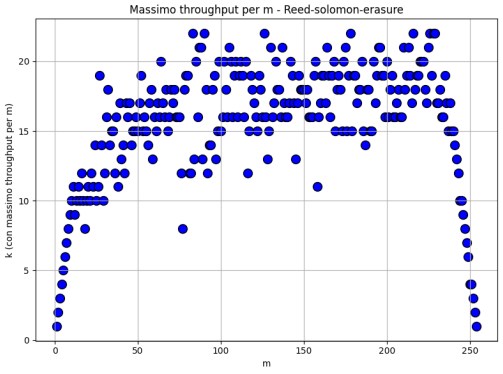


reed-solomon-erasure data_size = 50KiB

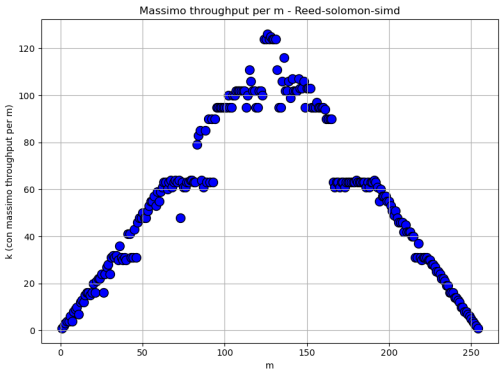


reed-solomon-simd data_size = 50KiB

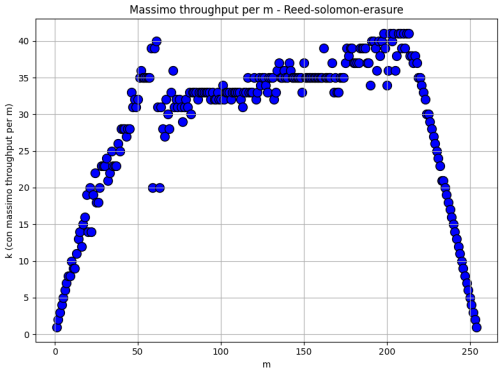
Figura 4.6: Plot per k che massimizza il throughput per ogni m dove data_size = 1KiB, 10KiB, 50KiB



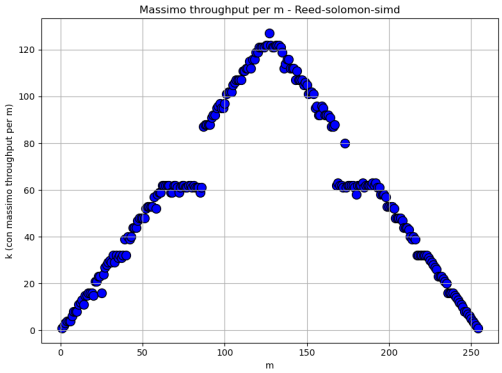
reed-solomon-erasure data_size = 100KiB



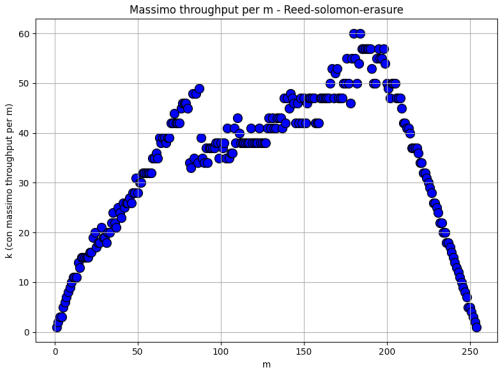
reed-solomon-simd data_size = 100KiB



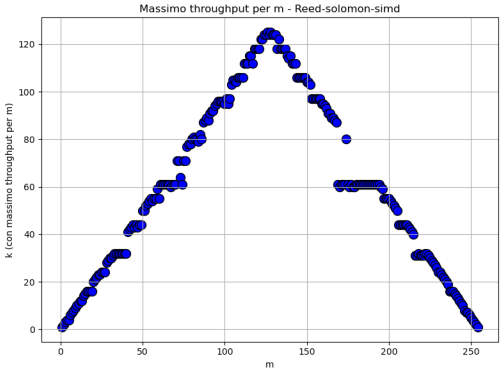
reed-solomon-erasure data_size = 500KiB



reed-solomon-simd data_size = 500KiB

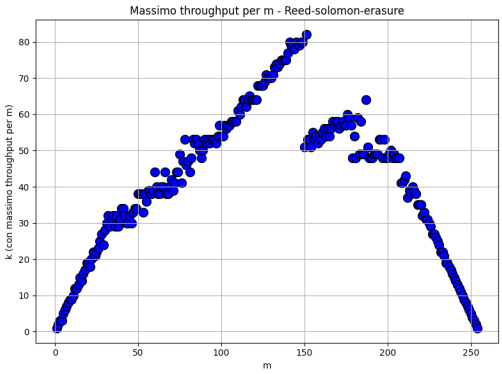


reed-solomon-erasure data_size = 1MiB

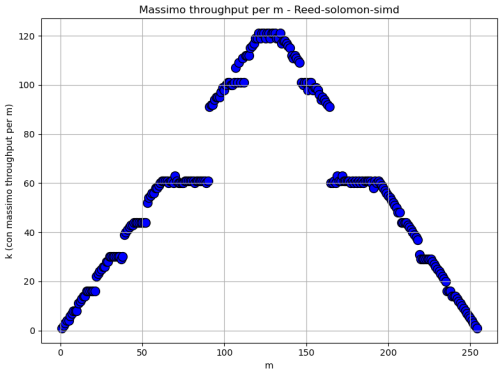


reed-solomon-simd data_size = 1MiB

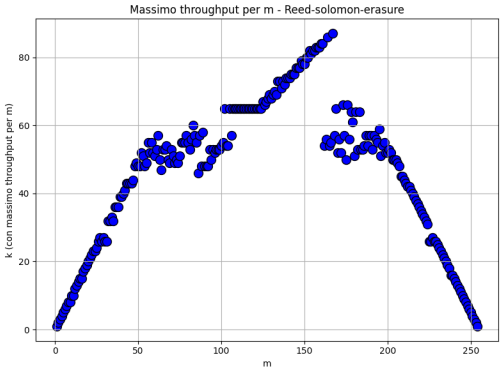
Figura 4.7: Plot per k che massimizza il throughput per ogni m dove data_size = 100KiB, 500KiB, 1MiB



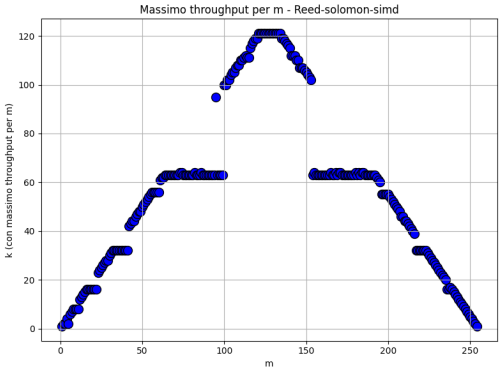
reed-solomon-erasure data_size = 2MiB



reed-solomon-simd data_size = 2MiB



reed-solomon-erasure data_size = 4MiB



reed-solomon-simd data_size = 4MiB

Figura 4.8: Plot per k che massimizza il throughput per ogni m dove data_size = 2MiB, 4MiB

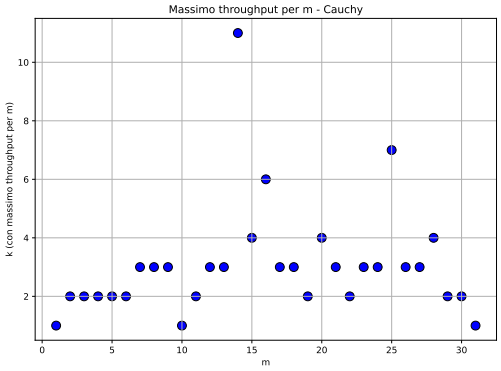
Osservazioni sui Grafici

Per quanto riguarda l'implementazione con *reed-solomon-erasure* con matrice di Vandermonde il comportamento è simile alla libreria *klauspost/reedsolomon* di Go con matrice di Vandermonde, nonostante abbia la forma a campana più accentuata che si inizia già a intravedere avendo una *data_size* di 100KiB (Figure 4.7a). Quindi all'aumentare della dimensione del dato da codificare si nota un minore rumore dei dati e assumono un comportamento un poco più deterministico rispetto a Go. Inoltre, l'assenza di un garbage collector permette una gestione diretta e continua delle risorse, può contribuire a evitare possibili interruzioni e overhead di un runtime garbage collected che renderebbe, come in Go, i dati più rumorosi. Tuttavia tale comportamento non è attribuibile soltanto a questo fattore ma anche a differenze implementative, modalità di gestione dei buffer ed encoder. Per *reed-solomon-simd* la situazione è differente. A differenza delle altre librerie, benché si ottenga un grafico a campana aumentando la dimensione del dato, questo ha una struttura a gradini, visibile specialmente da 100KiB (Figure 4.7b) in poi, poiché il k ottimale non varia continuamente ma si concentra su valori specifici. Ciò è frutto di un'implementazione che sfrutta aggressivamente le operazioni SIMD con algoritmi basati su FFT. In particolare, nell'implementazione Leopard il numero di frammenti di parità viene ricondotto alla successiva potenza di due, come anche il valore di n tale che $n \geq k+m$. Di conseguenza, per intervalli di valori di m la struttura rimane invariata e lo stesso valore di k continua a risultare ottimale. Quando invece viene superata una di queste soglie, il costo cambia e il massimo del throughput si sposta bruscamente, producendo l'effetto a gradini.

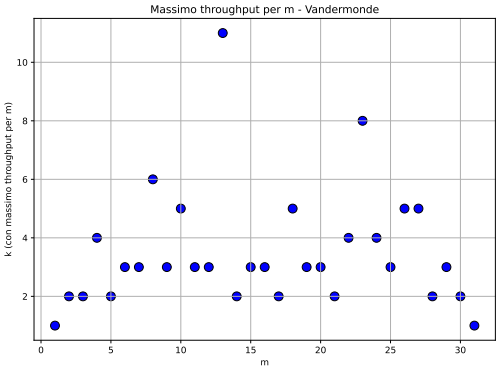
I throughput delle due librerie risultano differenti, premiando l'implementazione di *reed-solomon-simd* caratterizzata da una complessità asintotica più favorevole rispetto a *reed-solomon-erasure*.

4.2.4 Python

In questa sezione tratteremo l'implementazione su Python utilizzando la libreria *pyeclib* [7] [20] la quale è un frontend della libreria *liberasurecode* [19] che fa da ponte per permettere a *pyeclib* di comunicare con backend specifici scritti in C, come *jerasure* [21] e *Intel-Isa-L* [11]. I test sono stati effettuati su *Intel-Isa-L* con varianti Cauchy e Vandermonde e su *jerasure* con le medesime varianti. *Intel-Isa-L* rappresenta uno dei backend più performanti su CPU Intel/AMD moderne con architetture x86-64. Tutti i backend citati lavorano su GF(2^8) con limitazioni pratiche molto restrittive. *Intel-Isa-L* con matrice di Vandermonde e Cauchy e *jerasure* con matrice di Vandermonde hanno un limite di $k + m \leq 32$ [18], mentre *jerasure* con matrice di Cauchy $k + m \leq 16$ [17]. Ciò comporterebbe (per contrastare un ransomware) la possibilità di integrare delle copie degli shards per generare ulteriore traffico dati.

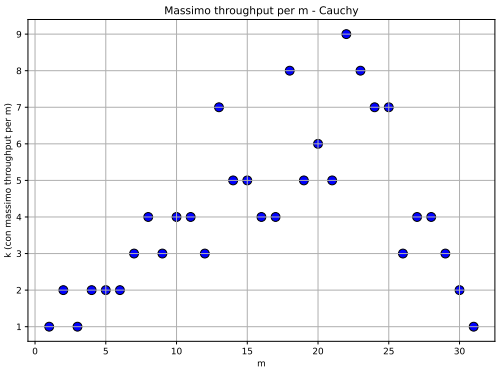


Intel-Isa-l Cauchy data_size = 1KiB

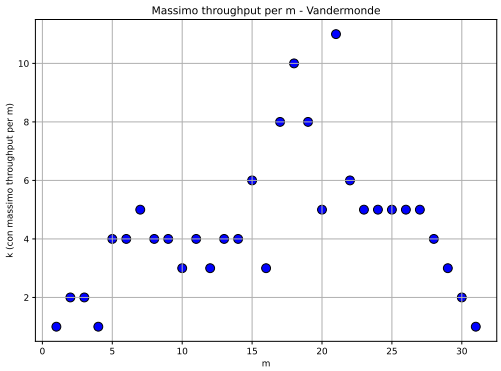


Intel-Isa-L Vandermonde data_size = 1KiB

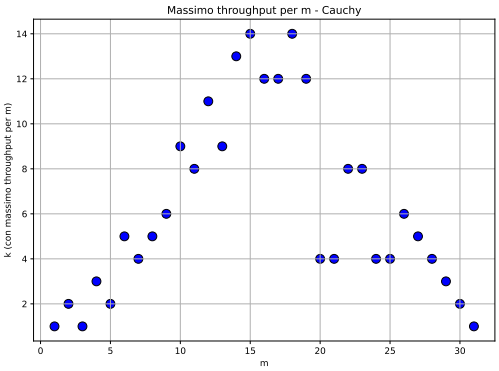
Figura 4.9: Plot per k che massimizza il throughput per ogni m dove data_size = 1KiB



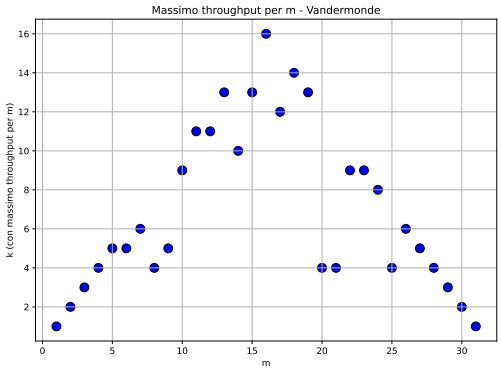
Intel-Isa-l Cauchy data_size = 10KiB



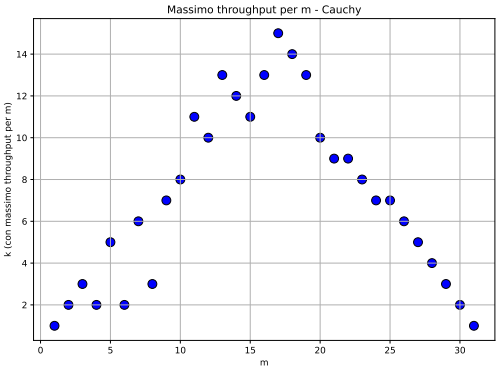
Intel-Isa-L Vandermonde data_size = 10KiB



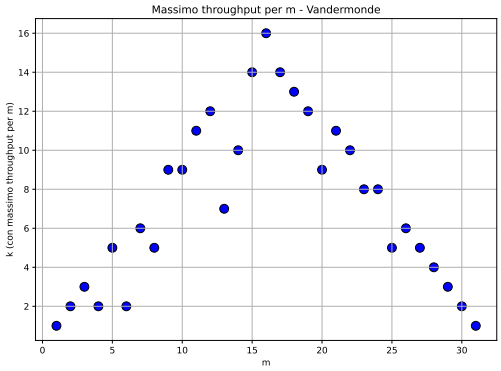
Intel-Isa-l Cauchy data_size = 50KiB



Intel-Isa-L Vandermonde data_size = 50KiB

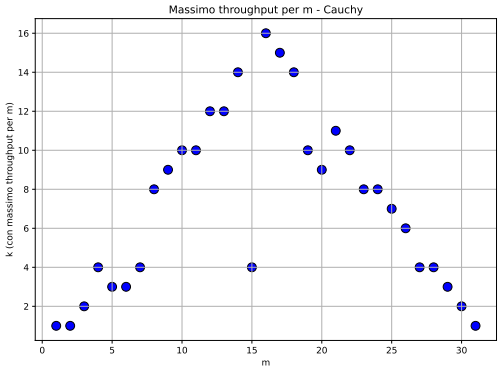


Intel-Isa-l Cauchy data_size = 100KiB

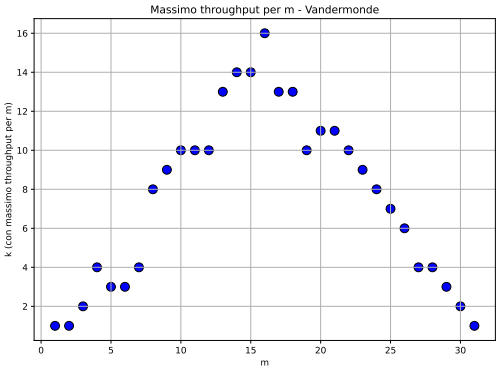


Intel-Isa-L Vandermonde data_size = 100KiB

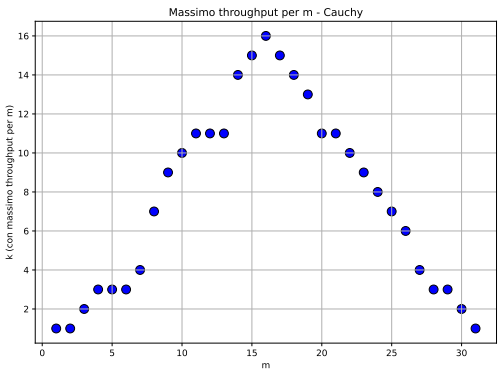
Figura 4.10: Plot per k che massimizza il throughput per ogni m dove data_size = 10KiB, 50KiB, 100KiB



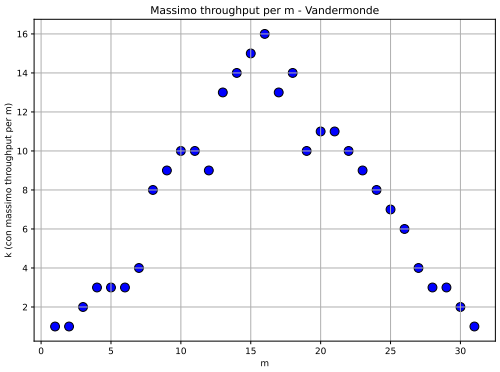
Intel-Isa-l Cauchy data_size = 500KiB



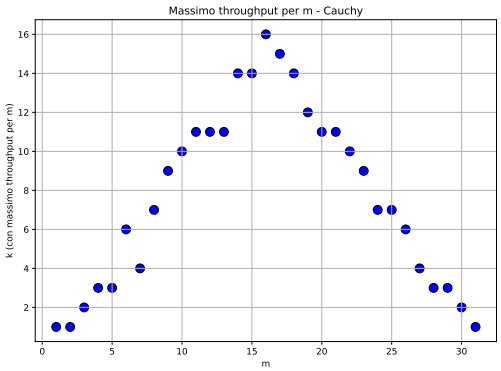
Intel-Isa-L Vandermonde data_size = 500KiB



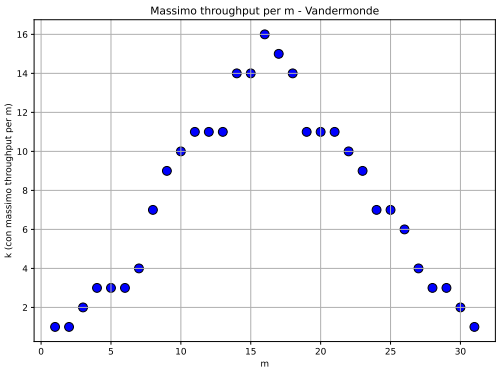
Intel-Isa-l Cauchy data_size = 1MiB



Intel-Isa-L Vandermonde data_size = 1MiB

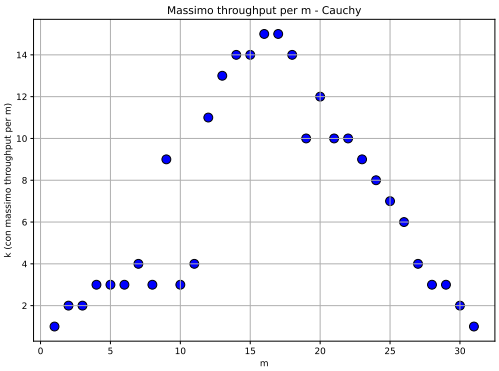


Intel-Isa-l Cauchy data_size = 2MiB

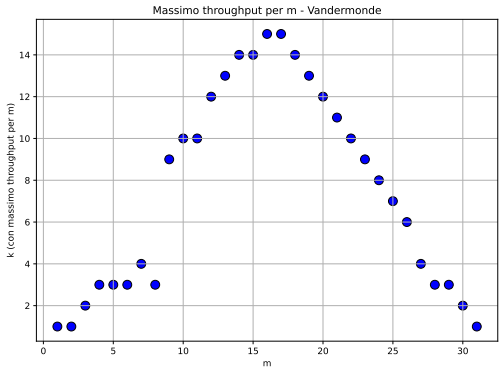


Intel-Isa-L Vandermonde data_size = 2MiB

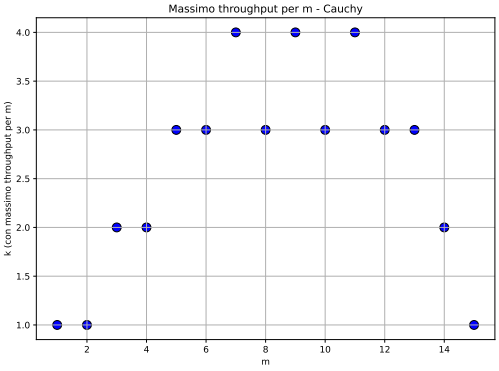
Figura 4.11: Plot per k che massimizza il throughput per ogni m dove data_size = 500KiB, 1MiB, 2MiB



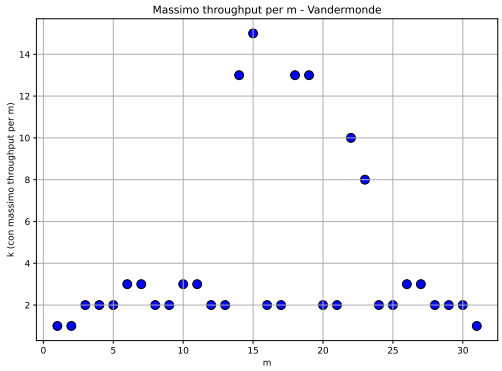
Intel-Isa-l Cauchy data_size = 4MiB



Intel-Isa-L Vandermonde data_size = 4MiB



jerasure Cauchy data_size = 4MiB



jerasure Vandermonde data_size = 4MiB

Figura 4.12: Plot per k che massimizza il throughput per ogni m dove data_size = 4MiB

Osservazioni sui Grafici

Sono stati rappresentati due grafici per la libreria *jerasure* per mostrare il comportamento differente da Isa-L al variare della matrice. Possiamo notare come l'andamento dei punti sia molto simile alle precedenti implementazioni, per via delle limitazioni date dalla libreria il dataset di test risulta minore rispetto le altre librerie ma l'andamento rimane simile. La libreria *pyeclib* permette ottimizzazioni SIMD tramite *Intel-Isa-L*. Nel caso il dispositivo non sia supportato si può configurare l'encoder con *jerasure* ricordando i limiti citati in precedenza. A differenza di *jerasure* (Figure 4.12c) (Figure 4.12d), usare una matrice rispetto a un'altra non ha praticamente alcuna differenza (sia a livello grafico che a livello di throughput) (Figure 4.12a) (Figure 4.12b). Siccome Isa-L, oltre ad avere limitazioni drastiche per k e m (che facilitano l'ottimizzazione delle operazioni nella cpu), utilizza un approccio in cui ogni coefficiente della matrice (indipendentemente da Cauchy o Vandermonde) viene trasformato in una tabella di lookup pre-calcolata e, durante la fase di codifica, il processore esegue operazioni di shuffle vettoriale che impiegano lo stesso numero di cicli di clock a prescindere dai coefficienti dati [12]. Questo contribuisce a spiegare anche il valore simile di throughput per i due metodi. Al contrario, *jerasure*, non avendo queste ottimizzazioni, ricade in un andamento riconducibile ai test effettuati su *klauspost/reedsolomon* di Go (Figures 4.5) e *reed-solomon-erasure* di Rust (Figures 4.8).

4.2.5 Considerazioni Finali sui Grafici

Possiamo notare dai grafici come ci sia un pattern ricorrente in tutte e tre le implementazioni. Ovvero la presenza di una struttura complessiva riconducibile a una campana sempre più visibile all'aumentare della grandezza del dato. Questa forma è in parte indotta dai vincoli $k \leq m$ e $k + m \leq 255$ che impongono che il massimo k teorico per ogni m tende ad avvicinarsi a $k_{max}(m) = min(m, 255 - m)$. Per dati sufficientemente grandi, il massimo throughput tende ad essere raggiunto vicino a tale valore siccome, a m fissato, aumentando k diminuisce la dimensione di ogni shard e quindi la quantità di parità da produrre ha dimensione e costi minore (proporzionale a $m \cdot shard\_size$). Qui il costo di encoding domina il tempo totale, quindi conviene spingere i k verso valori elevati compatibili con i vincoli. Per $data\_size$ piccoli, i valori di k tendono a stabilizzarsi su numeri bassi, siccome i costi per shard incidono maggiormente rispetto all'encoding. Incrementare k sì riduce la dimensione degli shard, ma aumenta anche l'overhead di gestione e può rendere meno efficaci alcune ottimizzazioni a basso livello. Di conseguenza il k migliore si stabilizzerà su soglie più basse e compaiono gradini o plateau. Per quanto riguarda l'implementazione con FFT, abbiamo un andamento più regolare dato dai vincoli precedentemente imposti con, a differenza delle scorse implementazioni, la comparsa di alcuni gradini a ridosso di alcuni valori di k, questo comportamento è coerente con la struttura dell'algoritmo, l'encoder FFT lavora su blocchi la cui lunghezza viene arrotondata ad una potenza di due, così da applicare la trasformata in modo più efficiente. Infine, nel confronto tra matrici di Vandermonde e di Cauchy emerge l'importanza del costo di setup dell'encoder. In particolare, l'approccio basato su Vandermonde, nelle implementazioni considerate, presenta generalmente un costo di generazione/precomputazione maggiore. La costruzione di Cauchy, avendo un overhead di setup più contenuto, segue all'incirca l'andamento "atteso" imposto dai vincoli e dal trade-off tra shard size e lavoro computazionale. Questa differenza però, per soluzioni ottimizzate a prescindere dalla matrice generatrice, viene annullata come in Python vedendo come la differenza nei grafici con implementazioni con matrici differenti sia quasi nulla.

4.2.6 Considerazioni sul Throughput

In questa sottosezione trarremo considerazioni sul throughput e sul throughput assoluto $T_{abs} = T \cdot \frac{k+m}{k}$ (definito come la quantità complessiva di dati generati in output per unità di tempo) in modo tale da avere un metodo di comparazione coerente con SSS, dal momento che in quel caso l'analisi è stata condotta tenendo

conto del volume totale di dati generati durante la frammentazione. SSS presenta, nel caso si generi flooding contro un crypto ransomware, un throughput pari a 187.1 MB/s con valori $n = 200$ e $k = 2$, mentre per contrastare l'esfiltrazione usa parametri $n = 150$ e $k = 6$ ottenendo un throughput pari a 53.76 MB/s.

Go

Dal punto di vista dei parametri, l'incremento di m porta generalmente a una riduzione del throughput, in quanto aumenta la ridondanza da produrre. Tuttavia considerando il throughput assoluto l'aumento di ridondanza si traduce in una maggiore quantità di dati prodotti al secondo. I risultati del throughput tendono ad aumentare all'aumentare della dimensione del file fino a stabilizzarsi su $data\_size$ elevati, siccome per dimensioni piccole il throughput è penalizzato dal costo di setup dell'encoder. Per quanto riguarda la differenza tra Cauchy e Vandermonde emerge che queste due varianti per quanto riguarda il throughput hanno un andamento piuttosto simile, con però un vantaggio da parte di Cauchy (soprattutto in file di piccole dimensioni), dovuto come descritto in precedenza dalla creazione della matrice prima dell'encoding effettivo. È importante osservare che i risultati ottenuti riflettono sia l'efficienza dell'algoritmo di codifica sia la parallelizzazione introdotta dalla libreria tramite goroutines che distribuisce il lavoro su più core. A confronto con SSS si ha un throughput assoluto più elevato, sia nel caso di SSS-Ransomware (dalle 8 alle 24 volte in base alla dimensione del file) che SSS-Exfiltration (dalle 3 alle 70 volte in base alla dimensione del file) (SOTTOSEZIONE (2.1.3)), usando i parametri k, n usati nel paper [10].

Rust

Il comportamento della libreria *reed-solomon-erasure* è riconducibile all'implementazione di Go, all'aumentare della dimensione del file il throughput aumenta fino a stabilizzarsi per dimensioni di file da 500KiB in poi. A differenza di Go però qui non si ha una parallelizzazione del lavoro perciò non si può fare un confronto vero e proprio. Per questo motivo i valori ottenuti sono generalmente più contenuti ma mantengono comunque un throughput assoluto migliore rispetto a SSS (fino a 2x nel caso SSS-Ransomware e da 2x a 7x nel caso SSS-exfiltration). Per quanto riguarda reed-solomon-simd il throughput tende a crescere fino a $data\_size$ 500KiB per poi scendere gradualmente per le dimensioni maggiori successive. In confronto alla libreria *reed-solomon-erasure* il throughput assoluto è tre volte maggiore, invece rispetto a SSS si ha nel caso di SSS-Ransomware un aumento da 2 a 10 volte in base alla dimensione del file e per SSS-Exfiltration da 6 a 35 volte.

Python

Per quanto riguarda la libreria *pyeclib* con backend *Intel-Isa-L* si ha un andamento del throughput molto simile a *reed-solomon-erasure* e *klauspost/reedsolomon* ovvero aumentando la dimensione del file il throughput aumenta fino a stabilizzarsi a dimensioni del file più elevate. In questo caso tra Cauchy e Vandermonde producono risultati molto simili anche in termini di throughput, in accordo con quanto osservato nei grafici. Ciò suggerisce che il backend ottimizzato domina il comportamento prestazionale più della matrice generatrice utilizzata. Un confronto con SSS risulta meno significativo siccome la libreria è limitata a un massimo di 32 frammenti, un'idea nel caso si usi questa libreria per generare flooding sarebbe quella di generare delle copie dei frammenti di parità in modo tale da raggiungere un numero di frammenti consono al DFaR.

Per *jerasure* la situazione è differente, Cauchy risulta mediamente più veloce rispetto a Vandermonde. A differenza del backend *Intel-Isa-L*, *jerasure*, è quindi più sensibile alla matrice generatrice utilizzata e i risultati mostrano un comportamento meno uniforme. Ciò è coerente con il fatto che le differenze implementative tra Cauchy e Vandermonde rimangano maggiormente visibili senza essere migliorate da un backend fortemente ottimizzato come invece avviene con *Intel-Isa-L*.

Capitolo 5

Conclusioni

All'interno di questa tesi abbiamo visto come sia possibile estendere Ranflood tramite l'impiego degli erasure codes, introducendo una strategia simile a SSS ma caratterizzata da un overhead di storage più controllabile e, in generale, da migliori prestazioni operative. Siamo partiti facendo un'introduzione di quello che accadrà all'interno della tesi per poi passare a un capitolo di background dove si è parlato approfonditamente di Ranflood, aritmetica dei campi finiti, SSS ed erasure codes. Passando a un capitolo di una bozza di implementazione per Ranflood elencandone gli obiettivi, dirigendoci poi verso una fase di test al variare della dimensione del file sulle varie librerie proposte in precedenza.

5.1 Sintesi dei Risultati

I risultati ottenuti mostrano come gli erasure codes siano un'estensione concreta e promettente per Ranflood. Rispetto a SSS consentono di mantenere una ricostruzione del dato a soglia con un overhead di storage più mitigato, permettendo di bilanciare in modo più flessibile ridondanza, quantità di dati generati e throughput complessivo. Dal punto di vista sperimentale (CAPITOLO 4), i test hanno evidenziato che i backend analizzati raggiungono throughput assoluti generalmente maggiori rispetto a SSS, rendendo gli erasure codes un meccanismo utile al flooding nel contesto DFaR oltre ad essere interessanti come meccanismo di recovery.

L'analisi ha mostrato come il comportamento prestazionale dipenda significativamente sia dalla dimensione del file sia dalla libreria utilizzata. Per file piccoli, i costi di setup e di gestione degli shards incidono in modo dominante, mentre per file medi e grandi questi costi vengono progressivamente ammortizzati e la fase di encoding tende a prevalere. In questo scenario, le implementazioni classiche basate su Reed-Solomon mostrano un andamento riconducibile a una forma a cam-

pana, influenzata dai vincoli sui parametri e dal compromesso tra dimensione degli shards, numero totale di frammenti e costo computazionale. Si è notato che backend fortemente ottimizzati, come Isa-L, tendono a ridurre praticamente del tutto le differenze tra matrici generatrici, mentre implementazioni meno ottimizzate, mantengono una maggiore sensibilità tra Cauchy e Vandermonde. Nel complesso, i risultati suggeriscono quindi che un'estensione basata su erasure codes per Ranflood sia una soluzione tecnicamente valida, capace di offrire un compromesso più favorevole tra throughput, ridondanza e occupazione di spazio sul disco rispetto a SSS, pur richiedendo meccanismi aggiuntivi nel caso sia necessaria confidenzialità dei dati.

5.2 Lavori Futuri

All'interno di questa tesi è stata esplorata una metodologia per contrastare i ransomware tramite erasure codes, proponendo una possibile integrazione teorica in Ranflood. In futuro si prevede di estendere questo lavoro nelle seguenti direzioni:

Implementazione completa in Ranflood Realizzare l'estensione effettiva in Ranflood integrandola nel suo modello a task e validandone la stabilità in scenari reali. Un'idea potrebbe essere introdurre un livello di astrazione che permetta di selezionare il "backend" di codifica/decodifica: ad esempio sviluppare un *wrapper* Java (JNI/FFM) verso librerie native ottimizzate, così da ridurre l'overhead e sfruttare implementazioni Reed-Solomon accelerate senza modificare la logica applicativa di Ranflood.

Valutazioni sperimentali su ransomware reali Eseguire test su campioni di ransomware e dataset realistici, misurando il rallentamento della cifratura, overhead I/O, consumo del disco e restore del dato originale dopo perdita o cifratura di frammenti.

Cifratura aggiungere una metodologia di cifratura e codifica dei dati per riuscire a contrastare anche gli exfiltration ransomware e valutarne l'efficienza.

Scelta dinamica del backend e di k e m Progettare un meccanismo che scelga dinamicamente il backend ottimale in base alle componenti hardware della macchina su cui è installato Ranflood per permettere tutte le accelerazioni e ottimizzazioni disponibili. Un'ulteriore specifica potrebbe essere la scelta dei valori di codifica dei dati in base allo spazio disponibile all'interno del disco, alla dimensione del file, alla tipologia di file (per esempio, un file più importante riceverà una ridondanza maggiore rispetto a uno meno importante oppure, nel caso il disco sia saturo, riducendo la ridondanza).

Bibliografia

[1] AndersTrier. Anderstrier/reed-solomon-simd: Reed-solomon simd library (rust). https://github.com/AndersTrier/reed-solomon-simd.

[2] Backblaze. Backblaze/javareedsolomon: Java reed-solomon library. https://github.com/Backblaze/JavaReedSolomon. Site/blog: https://www.backblaze.com/blog/reed-solomon/.

[3] Davide Berardi, Saverio Giallorenzo, Andrea Melis, Simone Melloni, Loris Onori, and Marco Prandini. Data flooding against ransomware: Concepts and implementations. *Computers & Security*, 131:103295, 2023.

[4] Davide Berardi, Saverio Giallorenzo, Andrea Melis, Simone Melloni, and Marco Prandini. Ranflood: A mitigation tool based on the principles of data flooding against ransomware. *SoftwareX*, 25:101605, 2024.

[5] catid. Leopard-rs: Mds reed-solomon erasure correction codes for large data in c. https://github.com/catid/leopard/blob/master/README.md.

[6] Keith Conrad. Finite fields. https://kconrad.math.uconn.edu/blurbs/galoistheory/finitefields.pdf.

[7] PyECLib contributors. Pyeclib: A simple python interface for implementing erasure codes. https://pypi.org/project/pyeclib/. Description: https://pypi.org/project/pyeclib/#description.

[8] Wikipedia contributors. Campo finito. https://it.wikipedia.org/wiki/Campo_finito. See also: https://en.wikipedia.org/wiki/GF(2).

[9] Wikipedia contributors. Codice reed–solomon. https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction.

[10] Daniele D'Ugo, Saverio Giallorenzo, and Simone Melloni. Contrasting crypto and exfiltration ransomware with shamir's secret sharing data flooding. In *24th IEEE International Conference on Trust, Security and Privacy in Computing*

and Communications, TrustCom 2025, Guiyang, China, November 14-17, 2025, pages 3143–3151. IEEE, 2025.

[11] Intel. intel/isa-l: Intelligent storage acceleration library (isa-l). `https://github.com/intel/isa-l`.

[12] Intel. Isa-l: Gf(2^8) operations and precomputed tables (headers). `https://github.com/intel/isa-l/blob/master/include/gf_vect_mul.h`. See also: `https://github.com/intel/isa-l/blob/master/include/erasure_code.h`.

[13] klauspost. klauspost/reedsolomon: Reed-solomon library (go). `https://github.com/klauspost/reedsolomon`. Docs: `https://pkg.go.dev/github.com/klauspost/reedsolomon`.

[14] Sian-Jheng Lin and Wei-Ho Chung. An efficient (n, k) information dispersal algorithm for high code rate system over fermat fields. *IEEE Communications Letters*, 16(12):2036–2039, 2012.

[15] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to reed-solomon erasure codes. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 316–325, 2014.

[16] M. Makovenko, M. Cheng, and C. Tian. Revisiting the optimization of cauchy reed-solomon coding matrix for fault-tolerant data storage. *IEEE*. `https://ieeexplore.ieee.org/document/9528940`.

[17] OpenStack. Jerasure backend: Cauchy reed-solomon limit (jerasure_rs_cauchy.c). `https://github.com/openstack/liberasurecode/blob/master/src/backends/jerasure/jerasure_rs_cauchy.c`.

[18] OpenStack. liberasurecode: max fragments (erasurecode.h). `https://github.com/openstack/liberasurecode/blob/master/include/erasurecode/erasurecode.h`.

[19] OpenStack. liberasurecode readme. `https://github.com/openstack/liberasurecode/blob/master/README.md`.

[20] OpenStack. pyeclib: A simple python interface for implementing erasure codes. `https://github.com/openstack/pyeclib`.

[21] James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications. Technical Report CS-08-627, University of Tennessee, Department of Electrical Engineering and Computer Science, 2008. Version 1.2.

[22] James S. Plank and Lihao Xu. Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications. *IEEE Transactions on Parallel and Distributed Systems*. https://ieeexplore.ieee.org/document/1659489.

[23] rust rse. rust-rse/reed-solomon-erasure: Reed-solomon erasure coding library (rust). https://github.com/rust-rse/reed-solomon-erasure. Docs: https://docs.rs/reed-solomon-erasure/latest/reed_solomon_erasure/.

[24] Zhirong Shen, Yuhui Cai, Keyun Cheng, Patrick P. C. Lee, Xiaolu Li, Yuchong Hu, and Jiwu Shu. A survey of the past, present, and future of erasure coding for storage systems. https://keyuncheng.github.io/files/publications/tos24ecsurvey.pdf.

[25] León van de Pavert. Reed-solomon encoding and decoding. https://www.theseus.fi/bitstream/handle/10024/32913/Reed-Solomon%20Encoding%20and%20Decoding.pdf.

[26] Wikipedia contributors. Cauchy matrix. https://en.wikipedia.org/wiki/Cauchy_matrix.

[27] Wikipedia contributors. Polinomio irriducibile. https://it.wikipedia.org/wiki/Polinomio_irriducibile.

[28] Wikipedia contributors. Vandermonde matrix. https://en.wikipedia.org/wiki/Vandermonde_matrix.