Pràctica A:
Filtres de Bloom i Cryptohashing

Noms: Max Marín, Èric López i Ferran Noguera
Data: 23/04/2017
Curs: Primavera 2017

Introducció

L 'objectiu d'aquesta pràctica és analitzar la probabilitat de *Fals Positiu* en un filtre de Bloom per a representar un conjunt de claus. Per a construir els filtres de Bloom hem usat tres funcions de hash i per a xifrar les claus n'hem usat dues més.

Filtre de Bloom:

El filtre de Bloom ens permet saber si un conjunt de dades es un subconjunt d'un altre. Té la característica que pot donar fals positiu (hi ha una certa probabilitat que discutirem més endavant als experiments) però no pot donar fals negatiu.
El filtre de Bloom es construeix a partir d'un conjunt de dades a les quals se'ls hi aplica una o varies funcions de hash que ocuparàn certes posicions al filtre amb valor *true*. El pas per saber si una paraula es en el filtre seria aplicar el mateix hash i comprovar si les posicions estan en *true*, només que una sigui a *false* significa que la paraula no hi és.

Descripció del programa

El programa principal rep un document *"D1"* amb *"C"* claus. Tot seguit xifra les *C*-claus amb el SHA-256 i l' FH1 i s' actualitzen els 6 filtres de Bloom (3 algorismes d' actualitzar * 2 algorismes d' encriptació). Tot seguit es mostra per pantalla el temps emprat en fer les actualitzacions.

Després llegeix un document amb *"N"* paraules generades de forma aleatòria i *"K"* claus del document *"D1"* (on *"K"* oscil·la entre el 30-80 % de les claus en *"D1"*). Cada *N*-paraula es busca en els 6 filtres de Bloom aplicant la combinació de hashes corresponent.

Finalment a partir de la cerca es fa el càlcul de Positius Reals, Falsos Positius i Nombre total de Positius per a cadascuna de les combinacions i es mostra el resultat per pantalla.

Descripció dels algorismes

A continuació venen descrits els algorismes que s'han fet servir per a realitzar la pràctica. Hem utilitzat dues funcions d' encriptació, sha256 i FH2, i tres formes diferents d'actualitzar i provar el Filtre de Bloom a partir de les claus encriptades.

Primera funció d' encriptació:

Inicialment vam fer una funció de hash (FH1) que crea un vector de enters de 8 posicions *"V"* amb valor inicial igual a 0 a totes les posicions. FH1 recorre la paraula *"S"* acumulant el valor ASCII de cada *ièssima*-posició de *"S"* a la *ièssima*-posició%8 de *"V"*.

Un cop s'ha recorregut *"S"* s'agafen cadascuna de les 8 posicions i es passen a un nou vector en forma d'string hexadecimal. Després es concatenen les posicions del vector donant com a resultat un nou string de *"n"* caràcters. Com que l' string resultant ha de tenir la mateixa longitud per cada paraula i el sha256 utilitza cadenes de 64 caràcters vam decidir que l FH1 donés com a resultat un string de 64 caràcters també. Per aconseguir això si la paraula resultant es menor de 64 caràcters s'afegeixen tants 0 com calguin per l'esquerra. Aquesta versió era bastant pobre ja que a l'hora d'actualitzar el filtre de bloom posava a *true* quasi sempre les mateixes posicions.

Després vam modificar aquesta primera primera versió (FH1.2) una mica i en comptes de col·locar els 0 per la esquerra després de concatenar es mirava si l' string hexadecimal de cada posició del vector era de mida 8. Si no ho era s'afegien 0 per l'esquerra a cada substring i al final de tot es concatenava donant un nou string de mida 64 (8x8). Aquesta versió millorava la primera per molt poc així que vam decidir aplicar més modificacions.

La tercera modificació (FH1.3) consistia en aplicar el FH1.2 al FH1 i això va millorar força els resultats tot i que encara s'allunyaven molt dels resultats obtinguts amb el sha256.

Finalment es va aplicar una última modificació (FH2) que va millorar molt els resultats fins a arribar a superar en alguns tests al sha256. Aquesta última modificació consisteix en un cop s'obté l' string a partir de l' FH1 se li va concatenant l' FH1 de l' string actual de forma recursiva fins que arriba a ser de mida 64.

VALOR DEL VECTOR AMB LA PARAULA VOLDEMORT

202	111	108	100	101	109	111	114

PAS A HEXADECIMAL

CA	6F	6C	64	65	6D	6F	72

VALOR DE LA PARAULA ENCRIPTADA SENSE MODIFICAR

CA6F6C64656D6F72

APLICANT LA PRIMERA VERSIÓ PER A ENCRIPTAR

00000000000000000000000000000000000000000000000000CA6F6C64656D6F72

APLICANT LA SEGONA VERSIÓ PER A ENCRIPTAR

000000CA0000006F0000006C0000006400000065000006D0000006F00000072

APLICANT LA TERCERA VERSIÓ PER A ENCRIPTAR

1801801801801801801DE28f999999C5CE9A99CF13513710B15F14010C15f16A

APLICANT LA VERSIÓ DEFINITIVA PER A ENCRIPTAR

CA6F6C64656D6F7299966CCA6CC96D66108132108139D8190106FD1A11C41A31

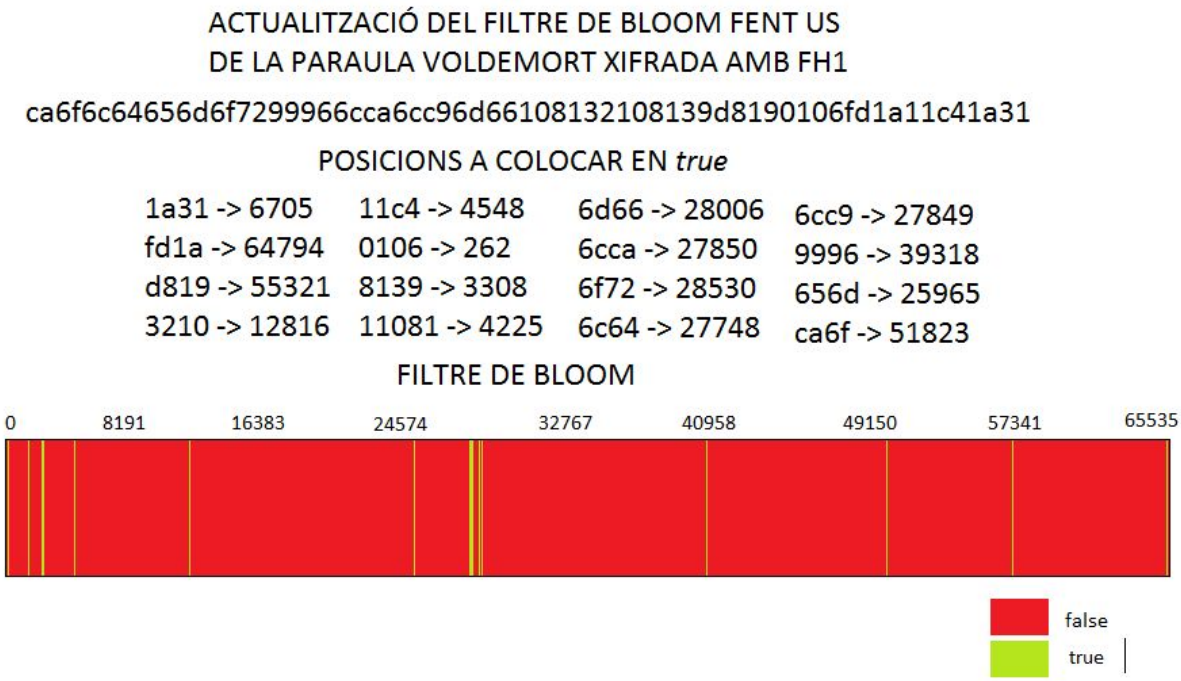
Segona Funció d' encriptació:

La segona funció d' encriptació que hem fet servir es el SHA-256**.** El SHA-256 es una de les quatre funcions que formen el conjunt de funcions hash criptogràfiques del SHA-2 dissenyades per la NSA. Aquesta funció en concret transforma qualsevol paraula en una cadena de 256 bits (64 caràcters) mitjançant l'ús de sumes, ANDs, ORs, XORs, SHRs i ROTs.

La funció la vam treure de la següent pàgina: http://www.krten.com/~rk/articles/bloom.html i vam adaptar-la per a que funcionés en el nostre programa.

Primer algorisme per a actualitzar i provar el Filtre de Bloom:

Un cop les paraules estan xifrades aquest algoritme les divideix en 16 blocs de 4. Al ser el filtre de Bloom de mida 65535 (0xFFFF) amb els 4 caràcters podem cobrir tots els valors. Un cop es tenen els 16 blocs es passa cada bloc a un enter *"I"* i es posa la *Ièssima*-posició del filtre de Bloom *"B"* a *true*.



A l'hora de mirar si una paraula del test està al filtre de bloom se li aplica la mateixa funció d' encriptació que a les claus i després es mira si les 16 posicions resultants estan a *true*. Si ho estan es compara la paraula amb el conjunt de claus per decidir si es tracta d'un cas *Positiu* o *Fals positiu*.

Aquest algorisme s'ha inspirat en el detallat en la següent pàgina: http://www.krten.com/~rk/articles/bloom.html

Segon algorisme per a actualitzar i provar el Filtre de Bloom:

Es va realitzar un segon algorisme per a actualitzar el filtre fent servir una nova funció de hash. Aquest algorisme pretenia millorar l' anteriorment realitzat, però ens vam adonar que no va ser així després de realitzar diverses proves.

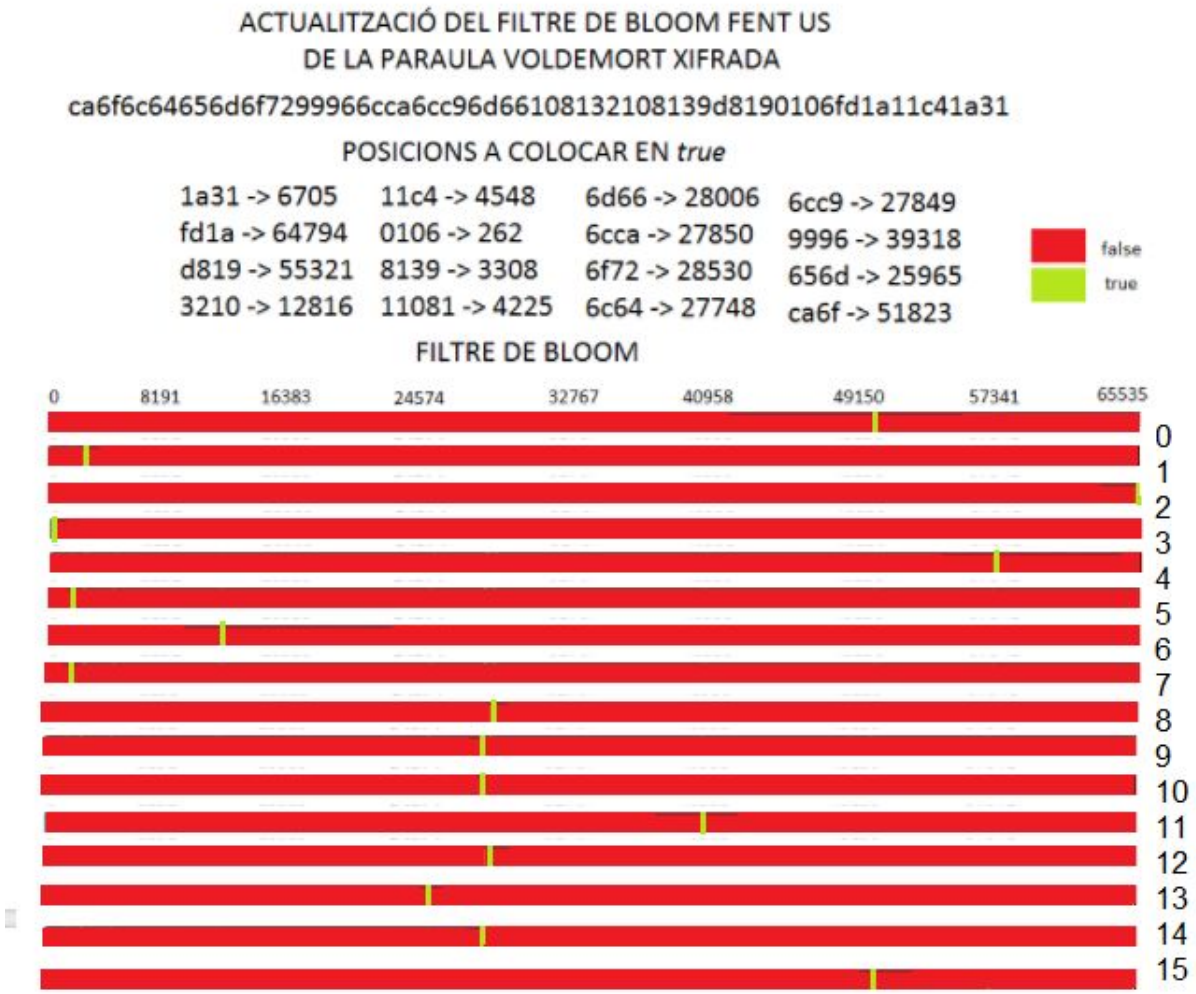
La funció de hash consisteix en partir les paraules (un cop encriptades en tamany 64) de vuit en vuit. Cada fragment de vuit de la clau en qüestió es transformarà d'hexadecimal a decimal i de decimal a binari, un cop fet això s'agafaran totes les posicions parells i imparells del número en binari i es separaran, cadascuna d'aquestes dos es convertirà altre cop en decimal i aquestes seran les claus que s'usaran en el filtre de bloom.

Agafem fragments de tamany màxim "0xFFFFFFFF" i el tamany del filtre de Bloom és de "0xFFFF". Com que aquests fragments es divideixen entre dos al fer la separació per posicions quan tenim el número en binari, les claus sempre ens donaran en un marge entre 0 <= x <= 65535 (0xFFFF), per tant podem cobrir per igual tot el tamany del filtre de Bloom.

Aquesta funció de hash s'ha inspirat en la publicació següent: *https://prakhar.me/articles/bloom-filters-for-dummies/*

Tercer algorisme per a actualitzar i provar el Filtre de Bloom:

Com fèiem en el primer algorisme dividirem la paraula xifrada en 16 blocs de 16 bits cadascun. Però per reduir la probabilitat de que es produeixi un fals positiu no omplirem un filtre de bloom representat per un vector d' una mida concreta. Sinó que utilitzarem una matriu de mida16x65535. D'aquesta manera quan toqui actualitzar el filtre de Bloom també tindrem en compte la posició de cada bloc de 16 bits del valor xifrat total de 256. D'aquesta manera reduirem significanment el nombre de falsos positius. Com a inconvenient tindrem l'espai, òbviament el nou filtre de Bloom ocupa 16 vegades més que amb els dos algorismes anteriors.



Com es pot veure en la imatge per a cada paraula xifrada només es posarà a "true" una posició per vector, es a dir 16 posicions es ficaràn a "true" si no ho estaven.

Complexitat dels Algorismes

En aquesta part analitzarem la complexitat dels algorismes que hem implementat a l'hora de actualitzar, cercar en el filtre i xifrar.

Complexitat de la primera funció d' encriptació:

```
string hash2(string original) {
    vector<int> v(8,0);
    int i = 0;
    int j = 0;
    while(i < original.length()) {
        v[j] = v[j] + (int)original[i];
        ++i;
        ++j;
        if(j == 8) j = 0;
    }
    vector<string> v8bits(8);
    for(int t = 0; t < 8; ++t) {
        v8bits[t] = dec_to_hex(v[t]);
    }
    string hashed = v8bits[0] + v8bits[1] + v8bits[2] +
    v8bits[3] + v8bits[4] + v8bits[5] + v8bits[6] +
    v8bits[7];
    hashed = arreglar3(hashed);
    return hashed;
}
```

Aquesta es la implementació de l' FH1. Es pot observar que la complexitat ve donada pels següents elements:

- El bucle *while (i < original.length)* que te complexitat **O(n)** on *"n"* es el tamany del string *"original"*.
- La crida a la funció *dec_to_hex* que es discuteix a continuació.
- La crida a la funció *arreglar3* que es discuteix tot seguit.

```
string dec_to_hex(int a) {
    stringstream stream;
    stream << hex << a;
    string result(stream.str());
    return result;
}
```

El cost d'aquesta funció depèn únicament del cost de la funció *hex*. Aquesta funció la hem tret de la pàgina http://www.cplusplus.com/reference/ios/hex/ y per desgracia no indica la complexitat de la funció així que no podem saber quina es. A partir d'ara ens referirem a la seva complexitat com **O(hex)**.

```
string arreglar3(string &vss) {
    if(vss.length() > 64) return vss.substr(0,64);
    else {
        string aux = hash1(vss);
        vss = vss + aux;
        return arreglar3(vss);
    }
}
```

La complexitat d' *arreglar3* depèn de la mida de l' string *"vss"* i de la crida a *hash1*. Tot i així mitjançant proves s'ha arribat a la conclusió de que en el pitjor cas (On la mida es igual a 1) la funció s'executa 4 vegades com a màxim. Podem concloure que la complexitat d' *arreglar3* serà 4* [Complexitat de hash1].

```
string hash1(string original) {
    vector<int> v(8,0);
    int i = 0;
    int j = 0;
    while(i < original.length()) {
        v[j] = v[j] + (int)original[i];
        ++i;
        ++j;
        if(j == 8) j = 0;
    }
    vector<string> v8bits(8);
    for(int t = 0; t < 8; ++t) {
        v8bits[t] = dec_to_hex(v[t]);
    }
    string hashed = v8bits[0] + v8bits[1] + v8bits[2] +
    v8bits[3] + v8bits[4] + v8bits[5] + v8bits[6] +
    v8bits[7];
    return hashed;
}
```

hash1 es una funció molt semblant a *hash2*. De fet fa el mateix però sense cridar a *arreglar3*. Podem dir doncs que el cost de hash1 és **O(n) + O(hex)** on *"n"* és la mida de l' string original.

Llavors podem concloure que el cost de *hash2* es **O(n + hex +4*(n+hex)) -> O(n + hex + 4n + 4hex) -> O(5n + 5hex) -> O(n + hex)**

Complexitat de la segona funció d' encriptació:

La segona funció per xifrar les paraules era la ja coneguda SHA256 que vam agafar ja implementada. No hem sabut calcular la seva complexitat temporal però donat els experiments i proves realitzades deduïm que és més eficient que el nostre FH1 implementat.

Complexitat del primer algorisme per a actualitzar el Filtre de Bloom:

```
void actBloom(vector<bool> &bloom, string s)
    stringstream ss;
    int j = 0;
    int value;
    for (int i = 0; i < 16; ++i) {
        value = stoi(s.substr(j,4),0,16);
        bloom[value] = true;
        j = j + 4;
    }
}
```

Aquesta funció te complexitat **O(1)** ja que tant el bucle com la funció stoi són independents de *"s"*.

El bucle sempre s'executa 16 cops i l' stoi sempre treballarà sobre 4 caràcters.

Complexitat del primer algorisme per a buscar en el Filtre de Bloom:

```
bool testBloom(const vector<bool> &bloom, string s) {
    stringstream ss;
    int j = 0;
    int value;
    for (int i = 0; i < 16; ++i) {
        value = stoi(s.substr(j,4),0,16);
        if(!bloom[value]) return false;
        j = j + 4;
    }
    return true;
}
```

Aquesta funció es comporta exactament igual que l' anterior així que podem confirmar que la seva complexitat es **O(1)**.

Complexitat de l' algorisme per a decidir si es *Fals Positiu*:

```
bool truePositiu(stack<string> ss, string s) {
    string aux;
    while(not ss.empty()) {
        aux = ss.top();
        if(s == aux) return true;
        ss.pop();
    }
    return false;
}
```

Aquesta funció te complexitat **O(ss)** on *"ss"* és el número de claus que s'utilitzen en el programa principal.

Complexitat del segon algorisme per a actualitzar i provar el Filtre de Bloom:

Aquest algorisme disposa de dos funcions principals. La primera serveix per actualitzar el filtre de bloom amb una paraula donada (actBloom2) i la segona per testejar si una paraula és o no en el filtre.

A continuació ens disposarem a analitzar els costs d'aquestes dos funcions:

```
173 //Pre: Rep el filtre de bloom i un string s codificat en hexa
174 //Post: Actualitza el filtre de bloom amb les dos posicions que et retorna la funcio de hash
175 void actBloom2(vector<bool> &bloom, string s) {
176     long x;
177     int j = 0;
178      for (int i = 0; i < 64; i+=8) {
179         string sd = s.substr(i,8);
180         x = todecfromhexa(sd);
181         stack<int> a_tractar = tobinary(x);
182         queue<int> parell;
183         queue<int> impar;
184         while(not a_tractar.empty()) {
185             impar.push(a_tractar.top());
186             a_tractar.pop();
187             if (not a_tractar.empty()){
188                 parell.push(a_tractar.top());
189                 a_tractar.pop();
190             }
191         }
192         bloom[todec(impar)%bloom.size()] = true;
193         bloom[todec(parell)%bloom.size()] = true;
194     }
195 }
```

Com es pot apreciar en la foto del codi, *actBloom2* rep com a paràmetres el filtre de bloom a actualitzar i l'string s, ja codificat, (paraula) a actualitzar. La paraula codificada té tamany 64 i la partirem en 8, als quals els aplicarem una conversió, explicada anteriorment en la descripció de la segona funció per provar i actualitzar el filtre de bloom.

L'únic cost que podem observar sense entrar en detalls de les funcions a les que crida és el de **O(j)**, entenent que j és a_tractar.size().

Aquesta funció fa crides a les funcions : "todecfromhexa", "tobinary" i "todec", les quals passen de decimal a hexa i de decimal a binari i de binari a decimal, respectivament. Anem a analitzar els seus costs:

```
154 long todecfromhexa(string x) {
155         long answer = 0;
156         int j = 7;
157         for (int i = 0; i<8; ++i) {
158                 int tmp = 0;
159                 if (x[j] == 'a') tmp = 10;
160                 else if (x[j] == 'b') tmp = 11;
161                 else if (x[j] == 'c') tmp = 12;
162                 else if (x[j] == 'd') tmp = 13;
163                 else if (x[j] == 'e') tmp = 14;
164                 else if (x[j] == 'f') tmp = 15;
165                 else tmp = (int)x[j]-'0';
166                 answer += tmp*pow(16,i);
167                 --j;
168         }
169         return answer;
170 }
```

La funció *todecfromhexa* crea un bucle sempre de mida 8, doncs sempre sabem que l' string que rebrà sempre serà de mida definit, 8. Així doncs podem concloure que aquesta funció té cost **0(8) -> 0(1)**.

```
133 stack<int> tobinary(long x) {
134     stack<int> psvl;
135     while(x>0) {
136         psvl.push(x%2);
137         x/=2;
138 
139     }
140     return psvl;
141 }
```

Observant *tobinary* podem concloure que la seva complexitat serà **0(log x)**, ja que anirà recorrent mentre x sigui més gran que 0 i l'anirà partint per dos.

```
143 int todec(queue<int> s) {
144     int a_ret = 0;
145     while(not s.empty()) {
146         int x = pow(2,s.size()-1);
147         a_ret += s.front()*x;
148         s.pop();
149     }
150     return a_ret;
151 }
```

Aquesta funció tindrà cost mida de la cua de s, assumim que k -> s.size(), així doncs la funció té cost: **0(k)**.

Podem concloure doncs que el cost total de la funció *actBloom2* és => **0(64/8*(1+log x+j)+ k + k)** -> **0(log x + j + 2k)**

```
198 //Pre: Rep filtre de bloom i string s a analitzar si hi es
199 //Post: Retorna true si la paraula "pot estar" al filtre de bloom i false si no hi es
200 bool testBloom2(vector<bool> &bloom, string s) {
201     long x;
202     int j = 0;
203         for (int i = 0; i < 64; i+=8) {
204                 string sd = s.substr(i,8);
205                 x = todecfromhexa(sd);
206         stack<int> a_tractar = tobinary(x);
207         queue<int> parell;
208         queue<int> impar;
209         while(not a_tractar.empty()) {
210             impar.push(a_tractar.top());
211             a_tractar.pop();
212             if (not a_tractar.empty()){
213                 parell.push(a_tractar.top());
214                 a_tractar.pop();
215             }
216         }
217         if (!bloom[todec(impar)%bloom.size()]) return false;
218         if(!bloom[todec(parell)%bloom.size()]) return false;
219         return true;
220     }
221 }
```

Aquesta funció rep per paràmetres el filtre de bloom ja actualitzat i una string s, retorna *true* si la paraula s està (o és fals positiu) en el filtre de bloom i false si no hi és.

Com es pot apreciar la funció és clavada a *actBloom2* menys per la part final, la qual no suposa cap canvi en el cost, per tant tenen exactament el mateix cost les dos funcions.

El cost d'aquesta funció és el mateix que l'anterior analitzada: **0(log x + j + k + k)**

Complexitat del tercer algorisme per a actualitzar i provar el Filtre de Bloom:

```
void actBloom3(vector<vector<bool> > &bloom, string s) {
    stringstream ss;
    int j = 0;
    int value;
    for (int i = 0; i < 16; ++i) {
        value = stoi(s.substr(j,4),0,16);
        bloom[i][value] = true;
        j = j + 4;
    }
}
```

La complexitat temporal d'aquest algorisme és exactament igual que la del primer algoritme per actualitzar el filtre de Bloom, ja que s'utilitza una funció similar. La diferència està en que en el primer algoritme es va omplin un sol vector i en aquest es va actualitzant una matriu de 16x65535 per això temporalment tindrem el mateix temps ja que no s'ha de recórrer la matriu sinó accedir a una posició concreta cada cop.

```
bool testBloom3(const vector<vector<bool> > &bloom, string s) {
    stringstream ss;
    int j = 0;
    int value;
    for (int i = 0; i < 16; ++i) {
        value = stoi(s.substr(j,4),0,16);
        if(!bloom[i][value]) return false;
        j = j + 4;
    }
    return true;
}
```

Com hem dit estem recorrent una matriu per tant el que no serà igual serà l'espai utilitzat pel filtre de bloom en aquest algoritme estarem ocupant 16 vegades més d'espai que en el primer. Amb els experiments realitzats posteriorment veurem si aquest increment d'espai ens dóna millor rendiment.

Experiments i Resultats

Per a realitzar els experiments hem utilitzat 4 conjunts de claus de mida diferent: 100, 500, 1000 i 10000.

Per tal de generar jocs de proves aleatoriament s'han generat quatre programes: "gen_proves100.cpp", "gen_proves500.cpp", "gen_proves1000.cpp" i "gen_proves10000.cpp". La diferència entre ells és bàsicament el fitxer d'on llegeixen i la quantitat de claus que n'agafen.

Cada fitxer generador de proves crea 6 tests diferents: test0, test1, …, test5. Cadascun d'aquests tests genera 500, 5000, 10000, 20000, 40000 i 80000 paraules aleatòries respectivament. Un cop fet això, depenent del programa que sigui, anirà a la carpeta claus (on hem emmagatzemat els quatre conjunts de claus) i en llegirà el fitxer de claus corresponent (100, 500, 1000 o 10000). D'aquest fitxer llegit n'adjuntarà al final del test entre el 30% i 80% de paraules, el percentatge es aleatori per cada execució, al igual que les claus escollides.

Els fitxers test0,test1,... es guardaran en la carpeta tests, dins d'aquesta estarà dividit per la mida del fitxer de claus que llegeixen per carpetes, dins de cadascuna d'aquestes hi haurà els 6 tests corresponents.

Un cop s'ha fet l'execució del script "generaricomparar.sh" es crearan tots els tests (o generaran de nous cada cop que s'executi l'script si ja n'hi ha de creats) i s'executarà per a cada test el programa principal. Els resultats de cada execució es guardaran en fitxers restest0 pel test0, restest1 pel test1, etc. dins de la carpeta 100 si es per la clau mida 100 paraules, 500 si es per la clau mida 500 paraules, etc. en la carpeta comparacions.

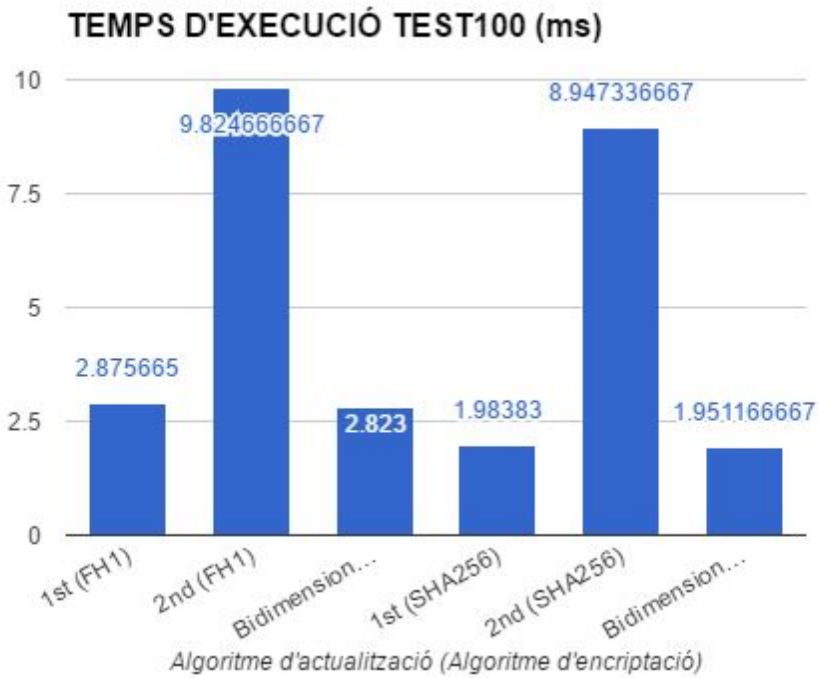
Tal i com s'ha explicat anteriorment, a l' hora de fer els experiments, em pres mostres dels següents elements per a cadascuna de les combinacions:

- Temps empleat en actualitzar el Filtre de Bloom.
- Percentatge de Falsos Positius.
- Percentatge de posicions a *true* del Filtre de Bloom.

Per a cada experiment hem realitzat tres taules que recullen les mostres i tres gràfiques que mostren la mitjana de resultats de cada taula.

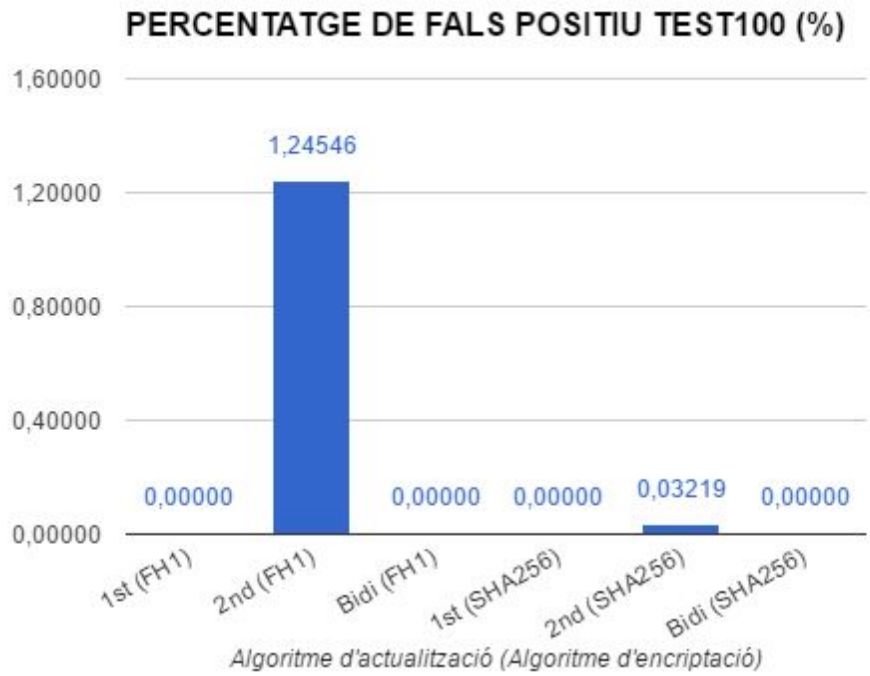
Resultats amb 100 claus:

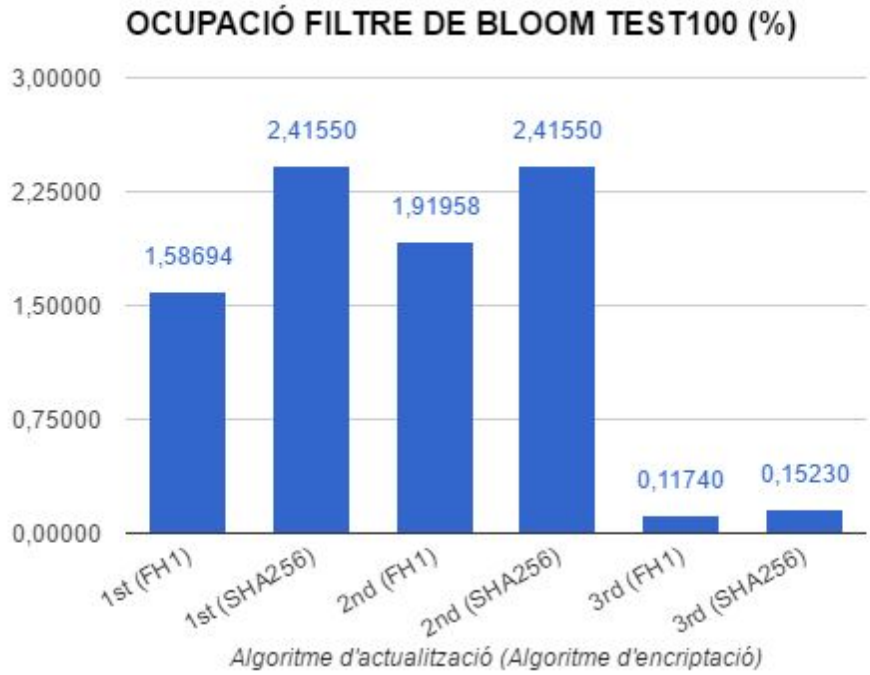
test100								
TEMPS (ms)		restest0	restest1	restest2	restest3	restest4	restest5	AVG
	1st	2.242	3.22999	3.034	3.09001	2.887	2.77099	2.875665
	2nd	8.008	10.583	10.316	10.341	9.978	9.722	9.824666667
FH1	Bidimensional	2.266	3.17699	2.927	3.01401	2.84499	2.70901	2.823
	1st	1.628	2.099	2.07299	2.08599	2.031	1.986	1.98383
	2nd	7.43401	9.668	9.439	9.34	8.90501	8.898	8.947336667
SHA256	Bidimensional	1.605	2.10201	2.056	2.051	1.99199	1.901	1.951166667



test100								
FALS POSITIU (%)		restest0	restest1	restest2	restest3	restest4	restest5	AVG
	1st	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2nd	1,83486	1,14965	1,04530	1,19731	1,05132	1,19433	1,24546
FH1	Bidimensional	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	1st	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2nd	0,00000	0,01982	0,05973	0,04490	0,03496	0,03373	0,03219
SHA256	Bidimensional	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000

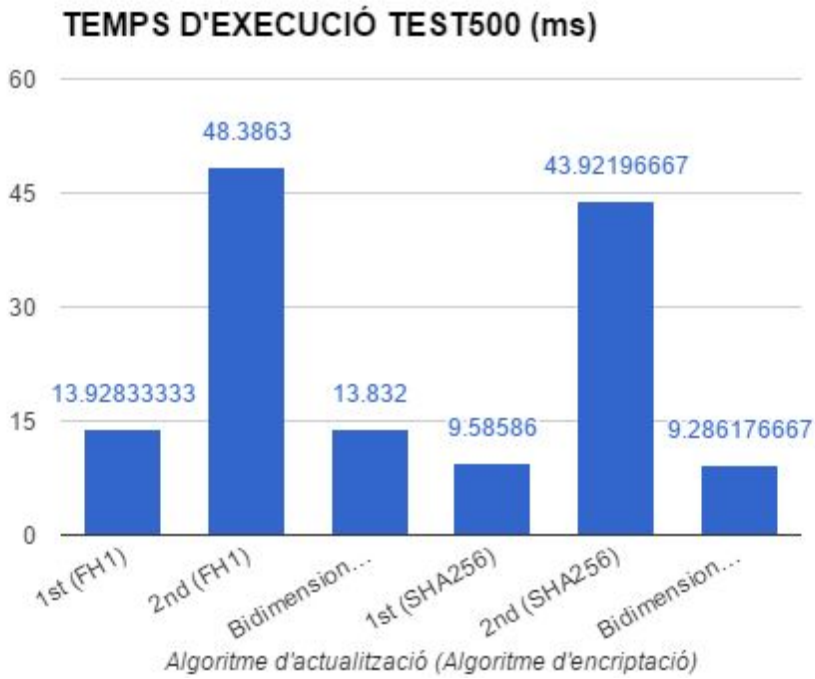
% D'OMPLERT DEL FILTRE DE BLOOM					
1r Algorisme		2n Algorisme		Bidimensional	
FH1	SHA256	FH1	SHA256	FH1	SHA256
1,58694	2,41550	1,91958	2,41550	0,11740	0,15230





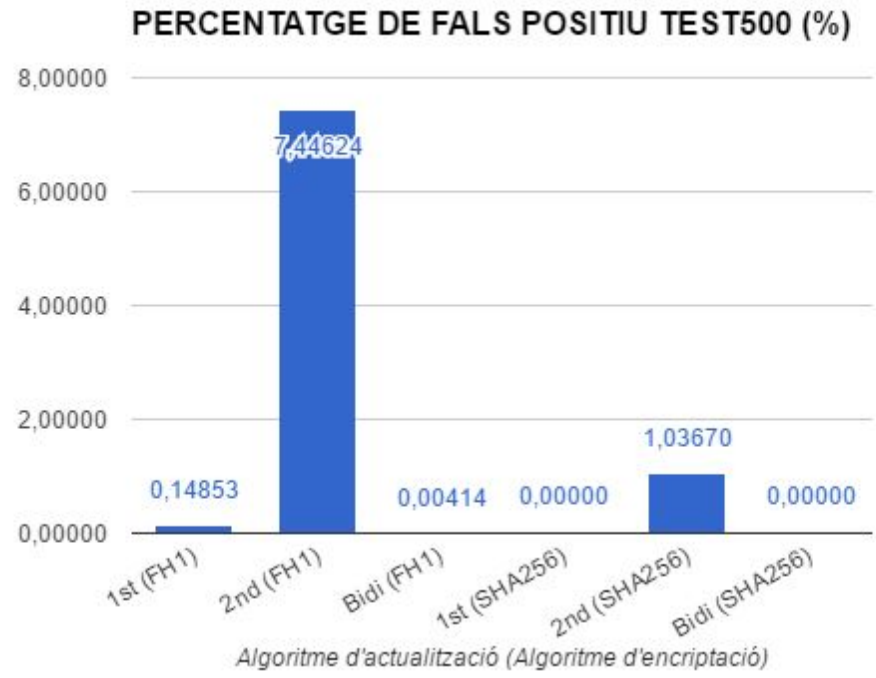
Resultats amb 500 claus:

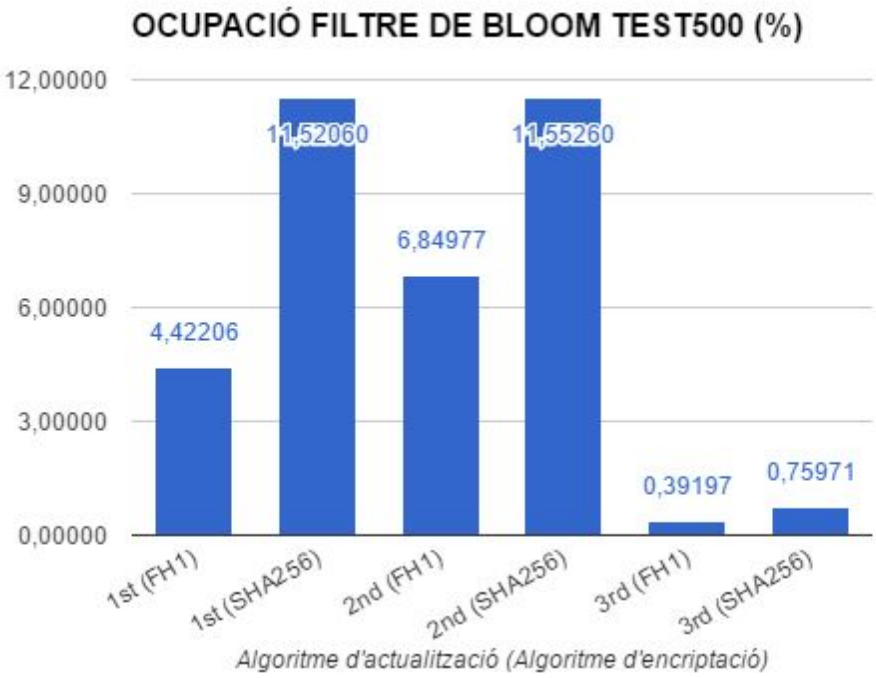
test500								
TEMPS (ms)		restest0	restest1	restest2	restest3	restest4	restest5	AVG
	1st	14.238	13.798	13.137	14.816	13.222	14.359	13.92833333
	2nd	49.011	48.0339	45.654	50.302	47.96	49.3569	48.3863
FH1	Bidimensional	14.133	13.783	12.953	14.7771	13.234	14.1119	13.832
	1st	9.70505	9.44899	9.19511	9.85292	9.5441	9.76899	9.58586
	2nd	44.392	43.3499	42.0509	45.0121	44.054	44.6729	43.92196667
SHA256	Bidimensional	9.31313	9.06502	9.01398	9.53998	9.23001	9.55494	9.286176667



test500								
FALS POSITIU (%)		restest0	restest1	restest2	restest3	restest4	restest5	AVG
	1st	0,11751	0,14951	0,17390	0,15724	0,15365	0,13939	0,14853
	2nd	5,28790	7,83031	7,40991	8,10280	8,10884	7,93767	7,44624
FH1	Bidimensional	0,00000	0,01869	0,00000	0,00491	0,00000	0,00124	0,00414
	1st	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2nd	0,35253	1,21473	1,15931	1,12034	1,15487	1,21840	1,03670
SHA256	Bidimensional	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000

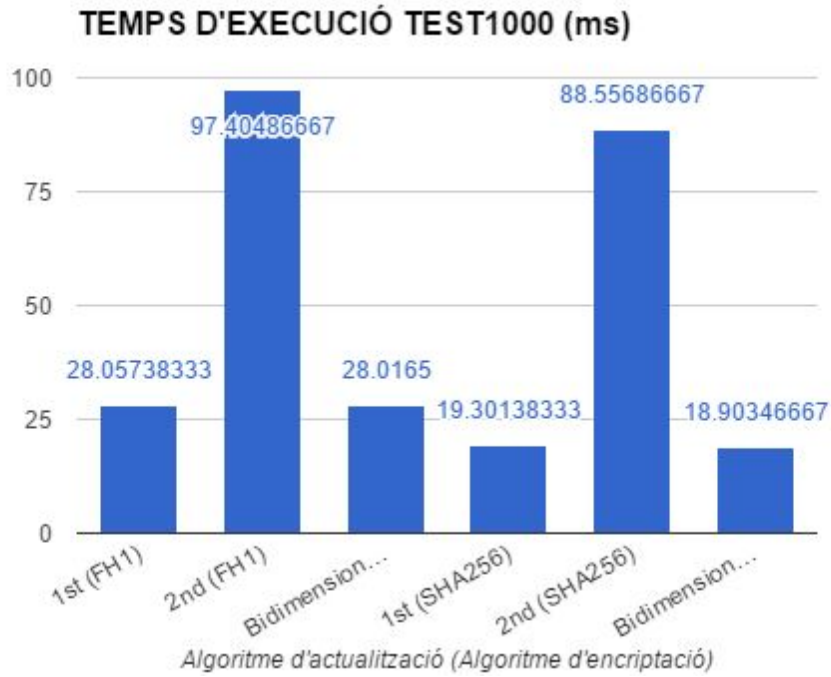
% D'OMPLERT DEL FILTRE DE BLOOM					
1r Algorisme		2n Algorisme		Bidimensional	
FH1	SHA256	FH1	SHA256	FH1	SHA256
4,42206	11,52060	6,84977	11,55260	0,39197	0,75971


PERCENTATGE DE FALS POSITIU TEST500 (%)
8,00000
6,00000
4,00000
2,00000
0,00000
0,14853
7,44624
0,00414
0,00000
1,03670
0,00000
1st (FH1)
2nd (FH1)
Bidi (FH1)
1st (SHA256)
2nd (SHA256)
Bidi (SHA256)
Algoritme d'actualització (Algoritme d'encriptació)

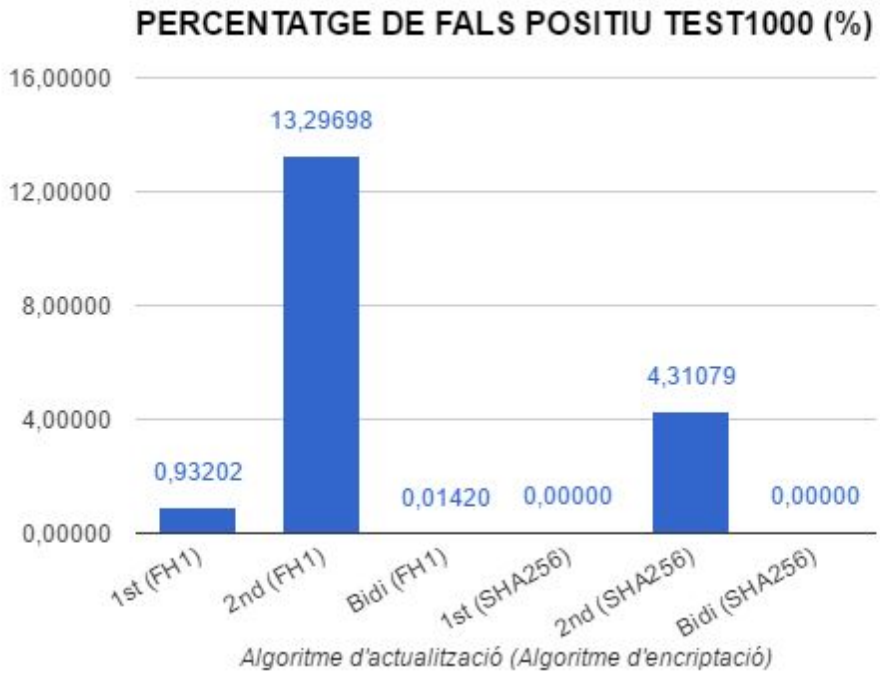

OCUPACIÓ FILTRE DE BLOOM TEST500 (%)
12,00000
9,00000
6,00000
3,00000
0,00000
4,42206
11,52060
6,84977
11,55260
0,39197
0,75971
1st (FH1)
1st (SHA256)
2nd (FH1)
2nd (SHA256)
3rd (FH1)
3rd (SHA256)
Algoritme d'actualització (Algoritme d'encriptació)

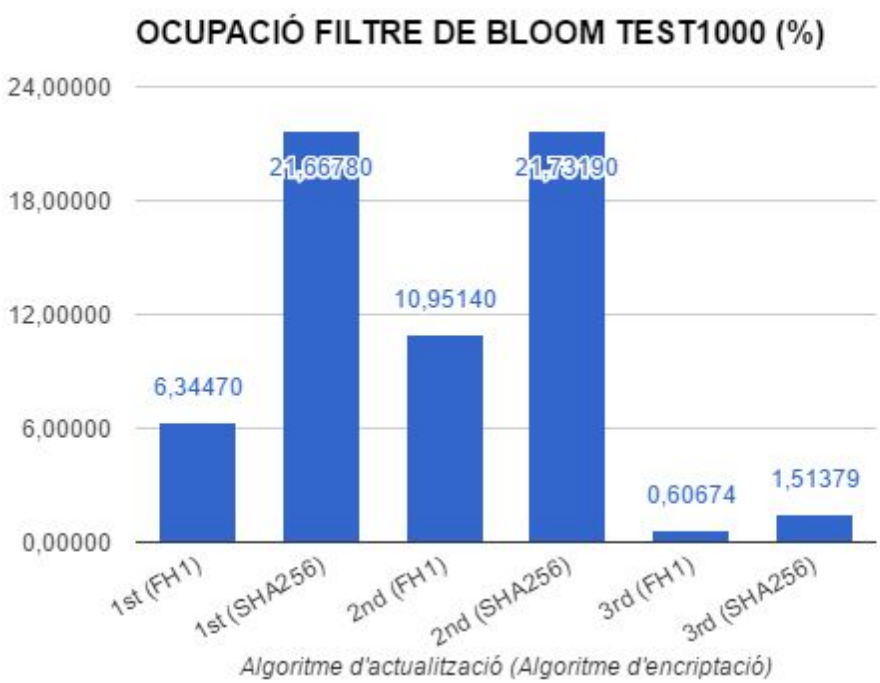
Resultats amb 1000 claus:

test1000								
TEMPS (ms)		restest0	restest1	restest2	restest3	restest4	restest5	AVG
	1st	30.323	27.7458	26.7262	27.814	27.0381	28.6972	28.05738333
	2nd	99.8671	95.434	94.8491	96.1579	95.5711	102.55	97.40486667
FH1	Bidimensional	30.326	27.2782	26.8959	27.2348	27.518	28.8461	28.0165
	1st	19.6812	19.3538	18.9101	19.0922	18.8181	19.9529	19.30138333
	2nd	89.1268	87.4419	86.8813	87.4721	86.9571	93.462	88.55686667
SHA256	Bidimensional	19.4591	18.9289	18.3025	18.856	18.2861	19.5882	18.90346667



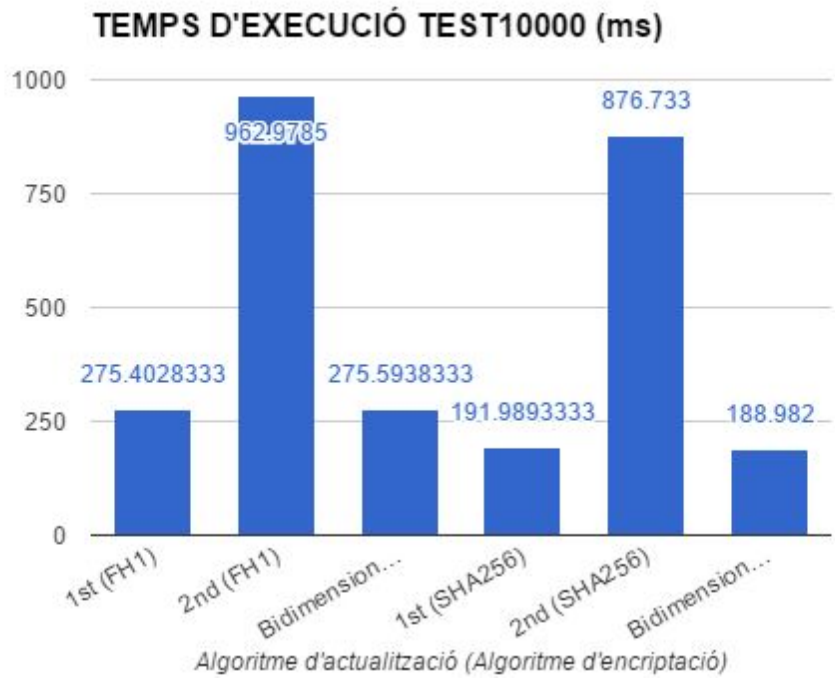
test1000								
FALS POSITIU (%)		restest0	restest1	restest2	restest3	restest4	restest5	AVG
	1st	0,33784	1,20638	1,00116	1,02021	1,01268	1,01383	0,93202
	2nd	9,34685	14,17970	13,84290	14,03280	13,97200	14,40760	13,29698
FH1	Bidimensional	0,00000	0,00000	0,00963	0,00981	0,03219	0,03359	0,01420
	1st	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2nd	2,81532	4,32442	4,55333	4,72827	4,80341	4,64000	4,31079
SHA256	Bidimensional	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
% D'OMPLERT DEL FILTRE DE BLOOM								
1r Algorisme		2n Algorisme		Bidimensional				
FH1	SHA256	FH1	SHA256	FH1	SHA256			
6,34470	21,66780	10,95140	21,73190	0,60674	1,51379			


PERCENTATGE DE FALS POSITIU TEST1000 (%)
16,00000
12,00000
8,00000
4,00000
0,00000
13,29698
4,31079
0,93202
0,01420
0,00000
0,00000
1st (FH1)
2nd (FH1)
Bidi (FH1)
1st (SHA256)
2nd (SHA256)
Bidi (SHA256)
Algoritme d'actualització (Algoritme d'encriptació)


OCUPACIÓ FILTRE DE BLOOM TEST1000 (%)
24,00000
18,00000
12,00000
6,00000
0,00000
21,66780
21,73190
10,95140
6,34470
0,60674
1,51379
1st (FH1)
1st (SHA256)
2nd (FH1)
2nd (SHA256)
3rd (FH1)
3rd (SHA256)
Algoritme d'actualització (Algoritme d'encriptació)

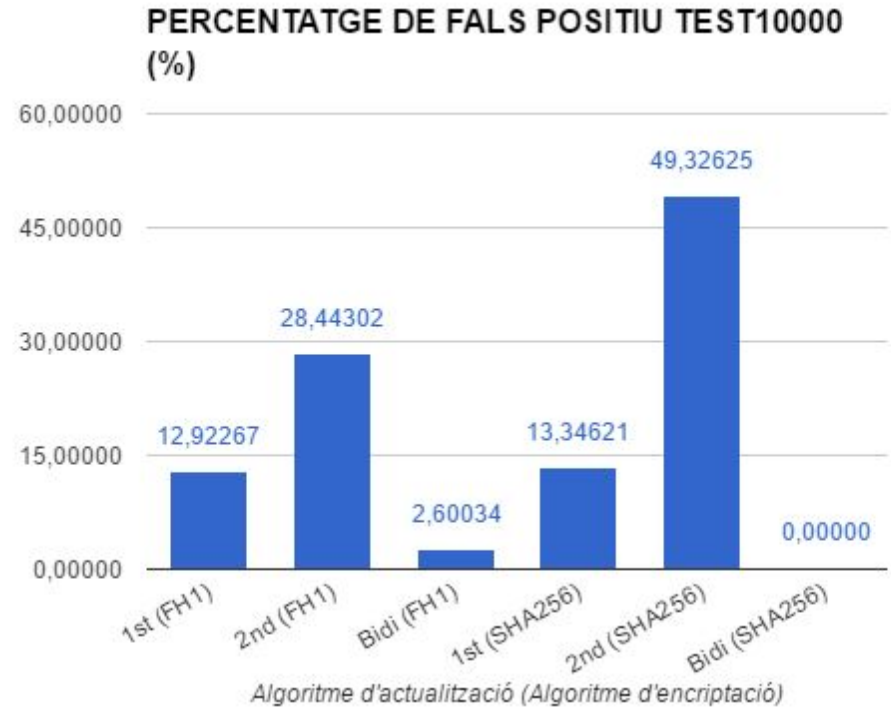
Resultats amb 10000 claus:

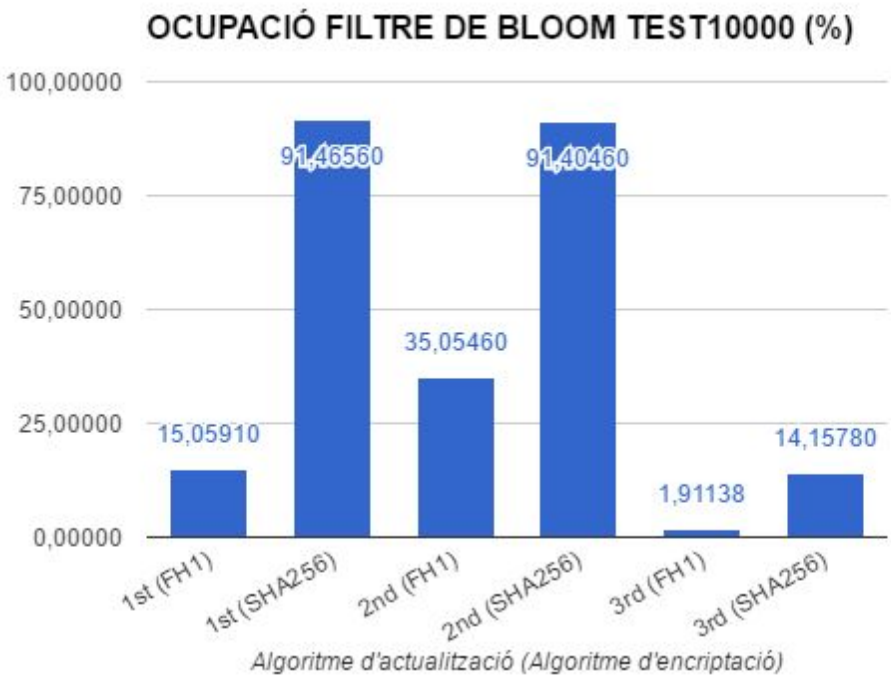
test10000								
TEMPS (ms)		restest0	restest1	restest2	restest3	restest4	restest5	AVG
	1st	277.939	277.863	273.787	264.716	283.209	274.903	275.4028333
	2nd	965.108	964.936	950.713	956.086	980.809	960.219	962.9785
FH1	Bidimensional	277.298	277.822	272.146	268.15	284.389	273.758	275.5938333
	1st	191.91	191.703	188.812	189.312	198.43	191.769	191.9893333
	2nd	877.223	875.758	863.011	878.119	891.568	874.719	876.733
SHA256	Bidimensional	188.466	188.163	184.822	186.112	197.704	188.625	188.982



test10000								
FALS POSITIU (%)		restest0	restest1	restest2	restest3	restest4	restest5	AVG
	1st	1,43931	8,87378	12,35090	16,06160	18,59310	20,21730	12,92267
	2nd	3,12062	19,59360	27,56040	35,12850	40,87770	44,37730	28,44302
FH1	Bidimensional	0,16558	1,82981	2,32263	3,25034	3,91122	4,12245	2,60034
	1st	1,54120	9,18954	13,07130	16,23340	19,30480	20,73700	13,34621
	2nd	5,41332	33,93250	48,00300	61,20800	70,90240	76,49830	49,32625
SHA256	Bidimensional	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000

% D'OMPLERT DEL FILTRE DE BLOOM					
1r Algorisme		2n Algorisme		Bidimensional	
FH1	SHA256	FH1	SHA256	FH1	SHA256
15,05910	91,46560	35,05460	91,40460	1,91138	14,15780


PERCENTATGE DE FALS POSITIU TEST10000 (%)
60,00000
45,00000
30,00000
15,00000
0,00000
12,92267
28,44302
2,60034
13,34621
49,32625
0,00000
1st (FH1)
2nd (FH1)
Bidi (FH1)
1st (SHA256)
2nd (SHA256)
Bidi (SHA256)
Algoritme d'actualització (Algoritme d'encriptació)


OCUPACIÓ FILTRE DE BLOOM TEST10000 (%)
100,00000
75,00000
50,00000
25,00000
0,00000
15,05910
91,46560
35,05460
91,40460
1,91138
14,15780
1st (FH1)
1st (SHA256)
2nd (FH1)
2nd (SHA256)
3rd (FH1)
3rd (SHA256)
Algoritme d'actualització (Algoritme d'encriptació)

Conclusions dels experiments:

En aquest apartat extraurem les conclusions obtingudes a partir dels experiments detallats en l' apartat anterior. Hem dividit els resultats en tres parts: Temps, Falsos Positius i Ocupació del Filtre.

Temps emprat en actualitzar el filtre:

A partir dels experiments hem pogut observar que el segon algorisme obté els pitjors resultats en quant a cost temporal i que tant el primer com el tercer es comporten de forma semblant. Aquests resultats compleixen el que s'esperava després de calcular la complexitat temporal dels 3 algorismes.

Referent a les funcions d' encriptació, podem afirmar que el SHA-256 és millor que el FH1. Tot i que per a poques claus la diferencia es poca, a mesura que el nombre augmenta la diferencia es comença fer més i més notable.

Falsos Positius:

Com es pot apreciar en els resultats obtinguts, el segon algorisme, combinat amb qualsevol de les dues funcions d' encriptació, es el que produeix més Falsos Positius. Entre el tercer i el primer algorisme, hem observat que el tercer es bastant millor que el primer, tal i com es pot observar en l' últim conjunt de proves. Era un resultat bastant esperable ja que el tercer algoritme utilitza un espai 16 cops més gran que el segón.

En quant a les funcions d' encriptació hem observat que el SHA-256 es en general millor que l' FH1, sobretot quan s' utilitza juntament amb l' algorisme bidimensional. Tot i així ens va resultar sorprenent que amb el conjunt de deu mil claus el FH1 aconseguís millors resultats que el SHA-256 quan s' utilitzaven amb el primer algoritme.

Ocupació en el filtre de bloom:

Com podem observar a les gràfiques quan s'utilitza encriptació SHA-256 el filtre de bloom queda més dispers que quan utilitzem FH1. Això ens fa deduir que quan utilitzem FH1 per encriptar els valors resultants són semblants i per tant es solapen moltes posicions al filtre de bloom. En tots els tests dóna un resultant semblant al descrit.

En l'últim test el resultat d'ocupació quan s'utilitza SHA-256 és increiblement alt, això intueix a pensar que donarà molts fals positius però realment el resultat es perquè el fitxer utilitzat té més d'un 80 % de positius reals, es a dir, es van utilitzar pràcticament totes les claus per fer el test i el nombre de positius es molt elevat.

Respecte el Bloom bidimensional, els resultats diuen que s' omple poc. Això es degut a que estem donant els resultats en %. L'espai utilitzat pel bloom bidimensional és 16 vegades més gran però introduïm el mateix nombre de claus que en els altres tests. Per tant quedarà sempre significativament més buit que els altres filtres.