Actividad 4 - El Chachash

Juan Pablo Briones

October 28, 2021

Abstract

Dentro de esta experencia, el objetivo es crear un algoritmo de hash competente con una seguridad basica. Ademas, se inserta este en un escenario real donde se compara contra la competencia y se estudia la calidad del software escrito. Se logra crear un algoritmo que genera hashes seguros a primera vista y se compara contra distintos algoritmos.

1 Introducción

Para esta actividad, el objetivo es crear un algoritmo de hash que sea capaz de generar *digests* de no menos de 25 caracteres y que cumpla con los requisitos generales que se buscan en un hash. Ademas, es importante estudiar la calidad del programa construido de 2 formas, analizando tanto el código como la teoria propuesta. Y por otro lado, compararlo con otros algoritmos ampliamente utilizados dentro de la industria.

Para esta tarea se crea un algoritmo donde se ocupa el cifrado de flujo **Chacha20** como inspiración para lograr

2 Marco teórico

Propiedades de una función de Hash

Aun que no llevan llave, hay 3 propiedades que una función de hash debe tener para ser considerada segura:

1. **Preimage resistance**: La función debe ser de un sentido. Osea, dado un resultado x, no debe ser posible derivar el mensaje ingresado a la función.

2. **Second image resistance**: Es de vital importancia que 2 mensajes no puedan tener un output igual. Aun que sabemos que esto es teoricamente imposible, ya que solo existen $2n$ hashes disponibles. Por lo tanto, se implementan distintos mecanismos para evitar que sea posible algun tipo de ataque que explote este hecho.

3. **Collision resistance**: Si elegimos 2 mensajes x1 y x2, debe ser imposible que h(x1) == h(x2). Esta propiedad es más dicifil de emplear ya que el atacante puede modificar tanto x1 como x2 para lograr su cometido. Y como sabemos de la *paradoja del cumpleaños* nos permite calcular con cuantos mensajes podriamos generar una colición.

ChaCha20

ChaCha20 es un cifrado de flujo, sucesor a salsa20. Este resalta como cifrado de flujo ya que igualmente arma una matriz de 4x4 palabras de 32 bits, donde se generan distintas operaciones matematicas con el objetivo de obtener un mayor grado de difusión y de confusión. Despues de 20 rondas, esta matriz se convierte en una palabra que mezcla mediante un XOR con todos los bloques de 512 bits de texto plano existentes.

Este algoritmo fue uno de los participantes en el concurso para el algoritmo AES, y actualmente ha tenido una creciente utilización en distintos proyectos debido a su rapidez y bajos requerimientos. Este es la base de otro algoritmo de hash llamado **BLAKE**

el cual se habre paso dentro del rubro actualmente debido a su gran rapidez.

3 Análisis

3.1 Código de Chachash

La base del código son 4 funciones, las cuales son necesarias para poder hashear un bloque.

1. AddPadding: Esta agrega padding a cualquier bloque que no tenga 512 bits. El padding consiste en llenar los bits faltantes con el mismo mensaje y luego aplicarle un xor por el largo inicial a todo el padding.

```
size_t faltan = 64-msg_len;
char *pad = malloc( faltan*sizeof(char) );
char *res = malloc( (65)*sizeof(char) );

for(;strlen(pad)≠faltan;)
    strncat(pad,msg, (faltan-strlen(pad)<msg_len)?faltan-strlen(pad):msg_len);

for(int i=0;i<faltan;i++)
  pad[i] ^= msg_len;
```

Figure 1: Código de Addpadding

2. chacha_state: Esta funcion crea la matriz de estado usada por Chacha, haciendo uso del primer bloque de texto plano.

Figure 2: Matriz state

3. HashABlock: Dentro de esta función, aplicamos algo similar al que chacha20 ocupa para encryptar 1 bloque. En esta hay 20 rondas, 10 son aplicadas en columnas y 10 son aplicadas en diagonales. La diferencia que nosotros modificamos la función QR, para que el resultado no pueda ser desencriptado.

```
// Initialices de status matrix with the chosen msg
// TODO - Add option to add salt
const uint8_t *constant = (uint8_t *)"expand 32-byte k";
const uint8_t *mensaje  = (uint8_t *)msg;

for(int i=0;i<4;i++)
  out[i] = pack4(constant+(i*4));
for(int i=0;i<8;i++)
  out[i] = pack4(mensaje+(i*4));
```

Figure 3: Código de chacha_state

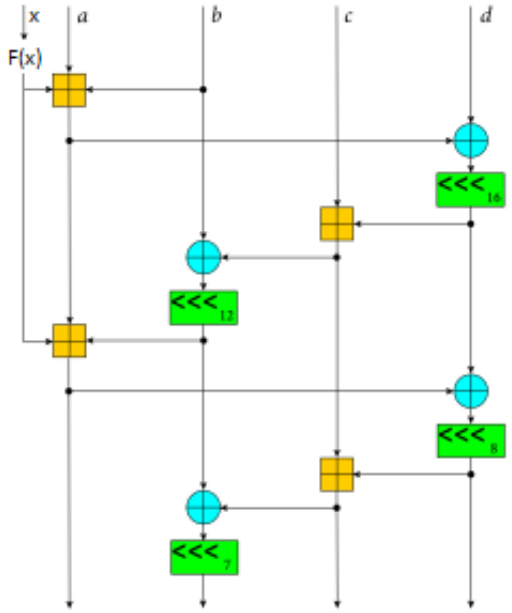

Figure 4: 1 ronda de Chachash

Esta función $F(x)$ genera un *Linear feedbak shift register* en un bit del input.

4. chacha hash: Esta es la función que junta todas las anteriores, esta función es la que produce el hash de la palabra completa. En esta se hacen las iteraciones necesarias para juntar todos los bloques del input. Como podemos ver en la imagen 5, el ciclo for hace todas ls iteraciones, donde finalmente se juntan todos los hash producidos en uno solo.

```
// Hashes a message
uint32_t state[16], hash[16], round[16];
char *temp = malloc(64*sizeof(char));
uint8_t *tmp;

for(int i=0;i<16;i++)
  hash[i] = strlen(msg);

chacha_state(state,msg);
for(;strlen(msg);msg += 64){
  if(strlen(msg) < 64)
    msg = AddPadding(msg, strlen(msg));
  strncpy(temp,msg,64);

  for(int i=0;i<16;i++)
    round[i] = ROTL(pack4(temp+(i*4)), i);
  HashABlock(round,state);

  for(int i=0;i<16;i++)
    hash[i] ^= round[i];

}
```

Figure 5: código chacha hash

3.2 Calculo de tiempos

Para comparar el rendimiento de CHACHASH contra ciertos algoritmos populares, se ocupan las utilidades presentes dentro de linux tales como

1. md5sum
2. sha1sum
3. sha256sum

4 Resultados

4.1 Comparación de robustes

Durante esta experiencia se compararon los siguientes algoritmos: Para medir el rendimiento del **CHACHASH** se prueban mensajes llenos de '0' de distinto largo: Y por ultimo, para poner este algoritmo en un contexto real, comparamos su rendimiento al hashear las N palabras del archivo *rockyou.txt* contra los algoritmos mencionados en la tabla 1.

Cifrado	Largo hash [bytes]	Entropia [bits]
MD5	16	128
SHA1	20	160
SHA256	32	256
CHACHASH	32	256

Table 1: Cifrados utilizados

Largo [bytes]	Tiempo [s]
10	0.001
100	0.001
500	0.001
1000	0.002
9678	0.003

Table 2: Tiempo en Hashear 1 mensaje

Cifrado	Tiempo [s]
MD5	0.002
SHA1	0.002
SHA256	0.003
CHACHASH	0.002

Table 3: tiempos x 1 hash

Cifrado	Tiempo [s]
MD5	0.002
SHA1	0.002
SHA256	0.003
CHACHASH	0.003

Table 4: tiempos x 10 hash

Cifrado	Tiempo [s]
MD5	0.002
SHA1	0.002
SHA256	0.003
CHACHASH	0.003

Table 5: tiempos x 20 hash

Cifrado	Tiempo [s]
MD5	0.002
SHA1	0.002
SHA256	0.003
CHACHASH	0.004

Table 6: tiempos x 50 hash

5 Análisis

5.1 CHACHASH

Se eligio chacha20 como la base de este algoritmo debido a su gran velocidad. Chacha solo ocupa operaciones basicas tales como sumas, xor o bitshifts que son operaciones de muy bajo costo para un procesador. Es por eso que incluso este algoritmo supera a AES en velocidad. Pero, para evitar que el output pueda ser revertido, fue necesario modificar ligeramente el esquema de chacha. Para esto ocupamos un *linear feedback shift register* el cual tambien prueba ser muy eficiente a nivel de cpu.

```
#define LFSR(r) (r >> 1) ^ (-(r & 1) & 0x80200003)
```

Figure 6: Formula lfsr

De todas formas, el nivel de confusión de el chachash es el punto más debil y con un análisis más profundo podria probar ser insuficiente para prevenir algun tipo de ataques. Lo que se podria hacer es aplicar una *S-Box* las cuales son ideales para este proposito.

5.2 Rendimiento

Tal como vemos en los resultados, el rendimiento de Chachash al aumentar el largo del input se mantiene practicamente igual. Eso muestra una vez más que el rendimiento es una de sus grandes fortalezas. Al momento de compararlo con otros algoritmos, vemos que su desempeño es muy similar al de sha256 lo que es un buen signo ya que ambos manejan bases similares.

6 Conclusión

Luego de haber generado este algoritmo, creemos que logramos acatar las propiedades presentes en funciones de hash seguras dentro de un rango razonable. Aun que este podria mostras ser vulnerable si lo sometemos bajo un análisis más rigurozo, su potencial en redes IoT o Celulares donde los recursos son limitados y necesitamos aprovecharlos lo máximo posible es bastante grande. Al igual que chacha20 este algoritmo logra evitar los problemas que presentan sha1 y md5 mientras que se mantiene un rendimiento similar a sha256, un algoritmo de hash que aun retiene gran popularidad.