Azrael: one streaming crypto hash for cloud computing

medinarosas.jorgealberto@gmail.com

Jun 2024

1. Contexto

En la actualidad, el cómputo distribuido es una necesidad, cada vez más necesitamos estar comunicados con los demás, y la conectividad de los individuos puede llegar a ser crítica. A partir, de esta necesidad, se plantea una definición de "la nube", no como algo concreto, sino más bien, como una acción de conectar a los individuos. Este trabajo, busca brindar un algoritmo para buscar objetos, si nosotros podemos identificar los objetos de una mánera ágil y determinística, entonces, tener la capacidad de alcanzar cosas u objetos en la nube será algo factible, rentable y plausible.

En la actualidad, la enorme cantidad de datos que se almacenan en la nube nos plantea identificar los objetos fuera y dentro de la misma de manera ágil y certera, y puede resultar en un gran reto tanto en manejo de espacio como de tiempo. Por ejemplo, buscar en una lista ligada nos podría llevar mucho tiempo si revisaramos de uno en uno.

Indexar la nube con identificadores rápidos de tipo hash nos ayuda a agilizar los procesos en el cómputo, y nos ayuda a mejorar el rendimiento/performance de las búsquedas y/o queries. La seguridad en el uso de estos identificadores es también un factor fundamental en el diseño de esta función. Esta característica nos obliga a buscar una función que además tendría que cifrar la entrada: hash criptográfico. De tal manera, que sea imposible o relativamente imposible adivinar las salidas de nuestra función no invertible[1].

La función deberá cumplir tanto en pseudo aleatoriedad como en dificultad para encontrar posibles colisiones. Nuestra función tiene una parte para inicializar, un ciclo principal recorriendo la cadena de entradas, rondas para rotar los bits, y una finalización.

2. Función

Tenemos la siguiente tarea: encontrar un algoritmo o función, con $\mathbf{X}$ el dominio de los textos planos o plain texts, y $\mathbf{Y}$ la imagen de la evaluación del hash, pero por simplicidad también se puede pensar como que el dominio y la imagen son el conjunto de los números naturales $f : \mathbb{N} \mapsto \mathbb{N}$ y pedimos que la función cumpla las siguientes condiciones:

- Que la función f sea inyectiva idealmente, con pocas colisiones.
- Que devuelva un valor $\mathtt{f(x)} = \mathtt{y}$ de tamaño fijo: $\left|\mathtt{y}\right| \leq \mathtt{k}$.
- Que sea muy difícil invertir la función f: que para cualquier $\mathtt{y} \in \mathbb{N}$ la tarea de encontrar una $\mathtt{x}$ tal que $\mathtt{f(x)} = \mathtt{y}$, nos lleve más que tiempo polinomial.
- Que sea muy difícil encontrar una colisión: tal que para una $\mathtt{y} \in \mathbb{N}$ tal que $\mathtt{f(x)} = \mathtt{y}$, la tarea de encontrar una $\mathtt{x_2}$ tal que $\mathtt{f(x_2)} = \mathtt{y}$, nos lleve más que tiempo polinomial.
- Que sea muy difícil encontrar colisiones: que la tarea de encontrar una pareja $(\mathtt{x_1}, \mathtt{x_2}) \in \mathbf{X} \times \mathbf{X}$ tal que $\mathtt{f(x_1)} = \mathtt{f(x_2)}$, nos lleve más que tiempo polinomial[2].

3. Posible Solución

Incrementalmente aproximar una función f que cumpla al menos experimentalmente lo antes mencionado.

4. Construcción[3]

Supongamos que la supuesta función f existe, y supongamos que existe una cinta infinita que está siendo procesada por el algoritmo, lo que buscamos es que en cada instante se conserve una memoria o un registro acerca de los bits que están pasando, y ya que tenemos la restricción de un tamaño fijo para la salida del hash entonces nos es imposible almacenar todos los bits, de tal manera que tendremos que usar algún método para acumular los bits.

En nuestro caso haremos sumas y reduciremos módulo la arquitectura del hardware:

$$a + b \equiv c \mod n \tag{1}$$

En la practica estaremos haciendo el wrapping del overflow a complemento-2.

Y necesitamos alguna función g para realizar la compresión, y además queremos que los estados anteriores afecten los bits, de tal manera que nuestra función se convierte en una función recursiva, en nuestro caso usaremos dos estados anteriores:

$$g_n(x) = g(g_{n-1}, x_{i-1}, x_i, x_{i+1}, g_{n-2}) \tag{2}$$

En cada iteración estaremos preservando las 5 últimas evaluaciones de la función de compresión y estaremos acumulando los bits nuevamente con sumas.

La suma nos provee una mejor dispersión de los bits que el xor debido al carry.

Además dispersamos contra el anterior, el actual y el siguiente: x_{i-1}, x_i, x_{i+1}.

5. Compresión

Para generar la función booleana, nos dimos a la tarea de evaluar todas las funciones booleanas de 5 operadores y recolectamos las mejores 12 en nuestras pruebas.

La función de compresión es la siguiente:

```
g(x1,x2,x3,x4,x5) =
    (( x1 + x2 ) ^ ( x3 ^ x4 ) ^ x5) +
    (( x1 & x2 ) ^ ( x3 + x4 ) ^ x5) +
    (( x1 ^ x2 ) + ( x3 + x4 ) ^ x5) +
    (( x1 ^ x2 ) ^ ( x3 + x4 ) ^ x5) +
    (( x1 & x2 ) + ( x3 + x4 ) + x5) +
    (( x1 & x2 ) + ( x3 + x4 ) ^ x5) +
    (( x1 ^ x2 ) ^ ( x3 + x4 ) ^ x5) +
    (( x1 | x2 ) ^ ( x3 + x4 ) ^ x5) +
    (( x1 | x2 ) | ( x3 + x4 ) ^ x5) +
    (( x1 + x2 ) + ( x3 + x4 ) ^ x5) +
    (( x1 + x2 ) & ( x3 + x4 ) ^ x5) +
    (( x1 ^ x2 ) + ( x3 ^ x4 ) ^ x5) +
    (( x1 | x2 ) ^ ( x3 ^ x4 ) ^ x5) +
    (( x1 + x2 ) + ( x3 ^ x4 ) ^ x5) +
    (( x1 + x2 ) ^ ( x3 + x4 ) ^ x5) +
    (( x1 + x2 ) & ( x3 + x4 ) + x5)
```

6. Cifrado

Dentro de nuestra función de compresión usamos la operación de la suma, y para poder invertir la función tendríamos que revisar todas las combinaciones, que están dadas por la siguiente ecuación:

$$C(n+k-1, k-1) = \frac{(n+k-1)!}{(k-1)!n!} \tag{3}$$

Donde k es igual a 16 y n a lo más esta dado por la precisión de la arquitectura, por ejemplo: $2^{63} - 1$

De igual manera utilizamos solamente dos estados anteriores, ya que más no nos proveen mejores resultados:

Sea $n \in \mathbb{N}$, definimos φ_n de la siguiente manera:

$$\begin{aligned} \varphi_0 &= 0 \\ \varphi_1 &= 1 \\ \varphi_n &= \varphi_{n-1} + \varphi_{n-2} \end{aligned}$$

Entonces tenemos que:

Teorema 1. *Sea* $k, n \in \mathbb{N}$

$$\sum_{k=0}^{n} \varphi_k + 1 = \varphi_{n+2} \tag{4}$$

Y podemos ver que sumar los dos anteriores números de fibonacci es equivalente a sumar los n-2 anteriores.

7. Main Loop

El main loop es el siguiente y dispersará los bits de la entrada en sumas módulo la arquitectura.

```
for( i=1; i<input.Length-1; i++ ) {
    x1 += sumaAnt1
    x2 += x3
    x3 += x4
    x4 += input[ i+1 ]
    x5 += sumaAnt2
    sumaAnt5 += sumaAnt4
    sumaAnt4 += sumaAnt3
    sumaAnt3 += sumaAnt2
    sumaAnt2 += sumaAnt1
    sumaAnt1 += g( x1,x2,x3,x4,x5)
}
```

8. Ronda Primera, i=0

```
x1 += input[ input.Length-2 ]
x2 += input[ input.Length-1 ]
x3 += input[ 0 ]
x4 += input[ 1 ]
x5 += input[ 2 ]
sumaAnt5 += sumaAnt4
sumaAnt4 += sumaAnt3
sumaAnt3 += sumaAnt2
sumaAnt2 += sumaAnt1
sumaAnt1 += g( x1,x2,x3,x4,x5)
```

9. Ronda Última, i=input.Length-1

```
x1 += sumaAnt1
x2 += x3
x3 += x4
x4 += input[ 0 ]
x5 += sumaAnt2
sumaAnt5 += sumaAnt4
sumaAnt4 += sumaAnt3
sumaAnt3 += sumaAnt2
sumaAnt2 += sumaAnt1
sumaAnt1 += g( x1,x2,x3,x4,x5)
```

10. Rotate de bits

En nuestra implementación estamos haciendo shift a la izquierda, esto es para acumular el hash en los bits altos:

```
for( i=1; i<input.Length-1; i++ ) {
    x1 += sumaAnt1
    x2 += x3
    x3 += x4
    x4 += input[ i+1 ]
    x5 += sumaAnt2
    sumaAnt5 += rotate( sumaAnt4, 8*(i+3)+d )
    sumaAnt4 += rotate( sumaAnt3, 8*(i+2)+c )
    sumaAnt3 += rotate( sumaAnt2, 8*(i+1)+b )
    sumaAnt2 += rotate( sumaAnt1, 8*i+a )
    sumaAnt1 += g( x1,x2,x3,x4,x5)
}
```

Podemos hacer a lo más 64 desplazamientos hacia la izquierda, y nos vamos moviendo en grupos de 8 con un offset.

11. Ronda Final Primera, i=0

```
x1 += input[ input.Length-2 ]
x2 += input[ input.Length-1 ]
x3 += input[ 0 ]
x4 += input[ 1 ]
x5 += input[ 2 ]
sumaAnt5 += rotate( sumaAnt4, d )
sumaAnt4 += rotate( sumaAnt3, c )
sumaAnt3 += rotate( sumaAnt2, b )
sumaAnt2 += rotate( sumaAnt1, a )
sumaAnt1 += g( x1,x2,x3,x4,x5)
```

12. Ronda Final Última, i=input.Length-1

```
x1 += sumaAnt1
x2 += x3
x3 += x4
x4 += input[ 0 ]
x5 += sumaAnt2
sumaAnt5 += rotate( sumaAnt4, 32+d )
sumaAnt4 += rotate( sumaAnt3, 24+c )
sumaAnt3 += rotate( sumaAnt2, 16+b )
sumaAnt2 += rotate( sumaAnt1, 8+a )
sumaAnt1 += g( x1,x2,x3,x4,x5)
```

13. Rotación

```
rotate(x,o) {
    o %= 64
    return x >>> o
}
```

14. Finalización

Primeramente dispersamos nuestras 5 sumas de bits dentro de la función de compresión:

```
sumaAnt1+=g(sumaAnt1,sumaAnt1,sumaAnt1,sumaAnt1,sumaAnt1)+IV[0]
sumaAnt2+=g(sumaAnt2,sumaAnt2,sumaAnt2,sumaAnt2,sumaAnt2)+IV[1]
sumaAnt3+=g(sumaAnt3,sumaAnt3,sumaAnt3,sumaAnt3,sumaAnt3)+IV[2]
sumaAnt4+=g(sumaAnt4,sumaAnt4,sumaAnt4,sumaAnt4,sumaAnt4)+IV[3]
sumaAnt5+=g(sumaAnt5,sumaAnt5,sumaAnt5,sumaAnt5,sumaAnt5)+IV[3]
```

Y apilamos las 5 sumas dentro de un entero de 64 bits:

```
hash = ((sumaAnt1 << 48) & 0xffffffffffffffffL ) |
       ((sumaAnt1+sumaAnt2 << 32) & 0xffffffffffffffffL ) |
       ((sumaAnt1+sumaAnt2+sumaAnt3 << 16) & 0xffffffffL) |
       ((sumaAnt3+sumaAnt4+sumaAnt5) & 0xffffffffL)
```

Y finalmente dispersamos el hash dos veces con la función de compresión:

```
hash += g( hash,hash,hash,hash,hash) + IV[2]
hash += g( hash,hash,hash,hash,hash) + IV[7]
```

Y por último, sumamos la longitud de la entrada y el número de pasos de la finalización:

```
hash += input.Length + 5 + 1 + 2;
```

En la implementación estamos usando 10 IV's o inicializadores.

```
IV[10] = {
  0x6a09e667bb67ae85,
  0x3c6ef372a54ff53a,
  0x510e527f9b05688c,
  0x1f83d9ab5be0cd19,
  0x428a2f9871374491,
  0xb5c0fbcfe9b5dba5,
  0x3956c25b59f111f1,
  0x923f82a4ab1c5ed5,
  0xd807aa9812835b01,
  0x243185be550c7dc3
}
```

15. Inicialización[4]

```
    x1=IV[0];
    x2=IV[1];
    x3=IV[2];
    x4=IV[3];
    x5=IV[4];
    sumaAnt1 = IV[5];
    sumaAnt2 = IV[6];
    sumaAnt3 = IV[7];
    sumaAnt4 = IV[8];
    sumaAnt5 = IV[9];
    a = 2;
    b = 3;
    c = 4;
    d = 5;
```

16. Padding[5]

El padding es algo necesario en los estándares de hash criptográficos y nos ayuda a mejorar el algoritmo.

```
padding( data ) {
  length = len(data)
  tail = length % 64
  if (64 - tail >= 9) {
    padding = 64 - tail
  } else {
    padding = 128 - tail
  }

  bits = length * 8
  pad[padding]
  pad[0] = 0x80
  for( i=1; i<=padding-1-8; i++ ) {
    pad[i] = 0;
  }
  for( i=0; i < 8; i++ ) {
    pad[padding - 1 - i] = (bits >> (8 * i)) & 0xFF;
  }
  memcpy(output,data,length);
  memcpy(output+length,pad,padding);
  return output;
}
```

[1] https://en.wikipedia.org/wiki/One-way_function

[2] Merkle, R. C.:One WayHash Functions and DES. Advances in Cryptolog y -Crypto'89 , LNCS 435 , Springer

[3] https://de.wikipedia.org/wiki/Merkles_Meta-Verfahren

[4] https://en.wikipedia.org/wiki/SHA-1#SHA-1_pseudocode

[5] https://csrc.nist.gov/files/pubs/fips/180-4/final/docs/draft-fips180-4_feb2011.pdf