Funções *hash*

Introdução

Faça o seguinte experimento. Visite o endereço

`https://dbastos0.github.io/crush/`

e observe que há a oferta de um executável chamado `crush.exe` com o número

`0b0dc1f3f5919b3ba4e85775adfe58e3af811bfb`

chamado `sha1sum`. O nome `SHA-1` representa uma função *hash* chamada de *secure hash algorithm 1* que foi projetada pela N.S.A. nos Estados Unidos. A função `SHA-1` não é mais considerada segura, mas, pro propósito para o qual a usaremos em instantes, ela ainda é perfeitamente adequada. (Ela não deve ser usada, entretanto, pra assinaturas digitais.)

De posse de um programa que implemente `SHA-1`, você pode computar a função no arquivo `crush.exe` e você provavelmente[1] obterá o mesmo número acima, o que confirma que sua cópia está intacta bit a bit com altíssima probabilidade. Por exemplo, em nosso computador, obtemos

```
%sha1sum.exe crush.exe
0b0dc1f3f5919b3ba4e85775adfe58e3af811bfb *crush.exe
```

confirmando que nossa cópia tem os mesmos bits da cópia remota.

Uma função *hash* é certamente uma função determinística. Por isso podemos usá-la pra confirmar a impecabilidade de uma cópia de arquivo. A função `SHA-1` vai sempre produzir o mesmo número desde que seu argumento sejam os bits do arquivo `crush.exe`.

Você não precisa fazer o experimento com o `crush.exe` necessariamente: escreva um arquivo-texto qualquer no seu computador e compute o `sha1sum` dele. Você vai obter um número de aproximadamente 160 bits como o acima. Troque uma letra do seu arquivo e recompute o `sha1sum` dele. Você obterá um número completamente diferente. Ao trocar uma letra, você pode ter trocado oito bits. Se você trocar um caractere `'0'` por um caractere `'1'`, por exemplo, você estará trocando apenas um bit—mesmo assim, o número produzido por `sha1sum` será completamente diferente do anterior. São propriedades assim que se deseja numa função *hash*.

[1] Você não obteria o mesmo número se porventura sua cópia não procedeu como deveria.

Eis o experimento. Num arquivo `hello.txt`, escrevemos a mensagem "`Tenho 0 truques na manga.\r\n`", exibimos seus bytes e computamos seu `SHA-256`. O byte destacado por colchetes será o byte que modificaremos no próximo passo.

```
%cat hello.txt
Tenho 0 truques na manga.

%od -t x1 hello.txt
0000000 54 65 6e 68 6f 20 [30] 20 74 72 75 71 75 65 73 20
0000020 6e 61 20 6d 61 6e 67 61 2e 0d 0a
0000033

%sha256sum.exe hello.txt
44ae3e977c679071cd8e827e3ecbf771c1b456ae4fa889b0bab2b0c022abbe02 *hello.txt
```

Em seguida, alteramos a mensagem para "`Tenho 1 truques na manga.\r\n`" e repetimos. Observe que o único byte alterado é o $30 \to 31$, o que destacamos com colchetes, mas o *hash* parece suficientemente diferente.

```
%cat hello.txt
Tenho 1 truques na manga.

%od -t x1 hello.txt
0000000 54 65 6e 68 6f 20 [31] 20 74 72 75 71 75 65 73 20
0000020 6e 61 20 6d 61 6e 67 61 2e 0d 0a
0000033

%sha256sum.exe hello.txt
66aae169812f2ebf4ba65635049b5c65ae58c4cfe43c40a949ad4ed7c7af8968 *hello.txt
```

A função `SHA-1` tem como domínio o conjunto de todas as sequências de bytes possíveis e como contradomínio o conjunto $\{0, 1, 2, ..., 2^{160} - 1\}$, isto é, `SHA-1` produz um número de até 160 bits. Por isso vemos esses números grandes e que, por isso mesmo, é conveniente que sejam expressos em base 16—por isso vemos os dígitos `a`, `b`, `c`, `d`, `e`, `f`, além dos dígitos 0–9. O mesmo se aplica à função `SHA-256`, exceto que seu contradomínio é $\{0, 1, 2, ..., 2^{256} - 1\}$.

Exemplos de funções *hash*

Funções *hash* não precisam ser sofisticadas como `SHA-1` ou `SHA-256`. Elas podem ser mais simples. Por exemplo, Donald Knuth [1, capítulo 6, seção 6.4, página 515] menciona

$$h(K) = K \bmod 1009,$$

sendo M algum número natural, como um exemplo de uma função *hash*.

Como um outro exemplo de função *hash*, dê uma olhada no ISBN do volume 3 do livro do Knuth nas referências deste documento. O último dígito do IBSN 0-201-89685-0 é o dígito zero. Esse ISBN foi emitido antes do ano de 2007 e, por isso,

tem nove dígitos, sem considerar o *hash* do número, que é o último dígito. A função *hash* para um ISBN anterior a 2007 é multiplicar o primeiro dígito por 10, o segundo por 9, o terceiro por 8 e assim sucessivamente até multiplicar o último dígito por 2. Soma-se todas essas multiplicações, obtendo-se a soma S e, por fim, computa-se $11 - S \bmod 11$. Em símbolos, a computação é

$$S = 10 \cdot 0 + 9 \cdot 2 + 8 \cdot 0 + 7 \cdot 1 + 6 \cdot 8 + 5 \cdot 9 + 4 \cdot 6 + 3 \cdot 8 + 2 \cdot 5$$

seguida de $11 - S \equiv 0 \bmod 11$.

O nome "*hash*"

Na língua inglesa, o verbo *to hash* significa cortar alguma coisa em pedaços pequenos ou de alguma forma misturar e bagunçar a coisa. Por exemplo, um café-da-manhã típico nos Estados Unidos é *hash browns* (ou *hashed browns*), que é um prato de batatas fritas cortadas bem fininhas e fritadas até dourarem. Veja fotos na Internet pra você ver a bagunça que fica: é uma bela ilustração do que uma competente função *hash* faz com seu argumento.

Nas palavras de Knuth,

> [...] the idea in hashing is to scramble some aspects of the key and to use this partial information as the basis for searching. – Donald Knuth, "The Art of Computer Programming", volume 3, capítulo 6, seção 6.4, página 514.

Knuth usa a palavra *scramble*—que significa "embaralhar"—, o que é bem empregado. Por outro lado, os americanos não dizem *hashed eggs* quando o café-da-manhã é ovos mexidos. Eles dizem *scrambled eggs*. (É a vida.)

Ter em mente essa ideia de que uma função *hash* tem como objetivo embaralhar o seu argumento e usar a informação como uma localização do argumento em alguma estrutura de dados é útil pra gente compreender algumas aplicações de funções *hash*, o que mencionamos brevemente na seção a seguir.

Funções *hash* em estruturas de dados

Funções *hash* são importantes elementos de importantes estruturas de dados. Por exemplo, os dicionários de Python—também conhecidos pelo nome "tabelas *hash*"—dependem enfaticamente da qualidade de uma função *hash*. Podemos pensar em tabelas *hash* como uma generalização de índices de *arrays* de forma que possamos, por exemplo, dizer `array["nome"]` e assim localizar o dado armazenado no índice inteiro do *array* que esteja associado à cadeia de caracteres `nome`. Uma implementação disso poderia ser a ingênua estratégia de consultar uma tabela para encontrar o inteiro associado a cadeia de caracteres `nome` e então usar esse inteiro e obter o dado no *array*. Outra estratégia é—em vez de consultar uma tabela—, computar o número inteiro usando-se uma função *hash*. Assim funcionam os dicionários de Python—você já consegue implementá-los em C. O que ocorre quando a cadeia de caracteres `nome`

produz o mesmo *hash* que uma outra cadeia? Ocorre uma colisão, uma situação em que a tabela *hash* precisa resolver de alguma forma. Colisões devem ser minimizadas, exigindo assim que uma função *hash* consiga bem distribuir seu *output* pela extensão do *array*. Veja também a propriedade desejada para uma função *hash* criptográfica ideal em que se deseja a imprevisibilidade de um bit qualquer do *output* da função—último parágrafo da seção sobre segurança de funções *hash*.

Eis uma outra ilustração da aplicação de funções *hash* em estruturas de dados. Um *Bloom filter* é uma estrutura de dados concebida por Burton Howard Bloom em 1970 com o propósito de verificar rapidamente se um elemento pertence a um conjunto. (Imagine um conjunto muito, muito grande.) É uma estratégia probabilística em que é possível obter uma resposta "sim, o elemento x está presente no conjunto" quando na verdade x não está presente, mas não é possível obter uma resposta "não, x não está presente" quando na verdade ele está presente. Em outras palavras, um *Bloom filter* produz respostas como "sim, possivelmente x pertence ao conjunto" ou então "não, com certeza x não pertence ao conjunto".

A segurança de funções *hash*

Nosso interesse, entretanto, é criptografia. Funções *hash* criptográficas têm grande importância em, por exemplo, assinaturas digitais. Sem uma função *hash* em assinaturas digitas—assinando a mensagem diretamente—, não só temos um desperdício de tempo e espaço (em computar um *hash* da mensagem inteira), mas também a encriptação da mensagem em si seria (se a mensagem tiver sido encriptada) trivialmente desfeita. Mas lembre-se de que assinaturas digitais não são apenas empregadas quando há uma encriptação. Podemos assinar uma mensagem sem qualquer desejo de encriptá-la. A assinatura confirma o autor da mensagem, que pode desejar exatamente pela publicidade de sua mensagem. Por exemplo, o presidente da república poderia postar em sua *homepage* mensagens endereçadas ao povo, terminando-a com sua assinatura digital pra que todos pudessem verificar que a mensagem de fato é do presidente e não de um impostor.

Em criptografia, usualmente se deseja uma *função hash criptográfica*. Mas o que é uma *função hash criptográfica*? É uma função *hash* que, idealmente, teria todas as propriedades definidas abaixo. É concebível que uma aplicação possa dispensar alguma dessas propriedades, optando por uma função *hash* alternativa com alguma fraqueza que seja desimportante para a aplicação. Por isso dizemos que, *idealmente*, *função hash criptográfica* teria todas as propriedades abaixo.

Definição (*pre-image resistant*, *one-way*)**.** Uma função $X \to Y : h(x)$ é *pre-image resistant* se, dado $y \in Y$, for computacionalmente caro encontrar $x \in X$ tal que $h(x) = y$.

Funções resistentes a pré-imagem são chamadas de *one-way functions*, ou seja, uma *one-way function* é uma função difícil de "inverter". Nem toda função *hash* é injetora e, logo, nem sempre possuem inversas—por isso estamos falando em "pré-imagem". Eis uma definição precisa de "pré-imagem".

Definição (pré-imagem)**.** Seja $X \to Y : f$ uma função. Seja $B \subseteq Y$. Diz-se que a pré-imagem de B relativa a f é $f^{-1}(B)$, que é, por definição, o conjunto de todos os elementos de X tal que $f(x) \in B$. Em símbolos,

$$f^{-1}(B) = \{x \in X \text{ tal que } f(x) \in B \subseteq Y\}.$$

Quando se fala em pré-imagem de um único elemento y, pressupõe-se que $B = \{y\}$.

Definição (*second pre-image resistant)***.** Uma função $X \to Y : h(x)$ é *second pre-image resistant* se, dado $x \in X$, for computacionalmente caro encontrar $x' \in X$ tal que $x' \neq x$ e $h(x') = h(x)$.

Em outras palavras, uma função é *second pre-image resistant* se não for viável encontrar um segundo elemento do domínio que seja mapeado ao mesmo y no contradomínio.

Definição (*collision resistant)***.** Uma função $X \to Y : h(x)$ é *collision resistant* se for computacionalmente caro encontrar dois elementos distintos $x, x' \in X$ tal que $h(x) = h(x')$.

Terminamos nossas anotações com uma noção de uma função *hash* ideal. Uma função *hash* criptográfica ideal teria a propriedade que se chama *indistinguishability*: mudando-se um bit no *input*, perde-se completamente qualquer capacidade de previsão do *output*, ou seja, a probabilidade de você adivinhar um bit qualquer do *output* é 1/2. Uma função *hash* que possua essa propriedade tem as três propriedades acima. Em outras palavras, *indistinguishability* implica as três propriedades acima. Faz sentido—se não consigo aprender nada sobre a função dado seu *output*, então a função é indistinguível de um processo verdadeiramente aleatório. Na prática, nenhuma função *hash* é ideal assim, mas as que são consideradas seguras hoje costumam possuir as três propriedades acima—SHA-256, por exemplo.

Referências

[1] Donald E. Knuth. *The Art of Computer Programming, volume 3, searching and sorting*. 2ª ed. Redwood City, CA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.