Principles of Computer Security
Exercises 2

Stuart Golodetz

February 5, 2006

1. Suppose I have chosen a number in the range $1, ..., n$. Your task is to guess it. How many guesses will it take on average if:

   (a) You try each number in that range in some order, without repeating guesses;

   (b) You pick each guess at random, possibly repeating guesses?

   Answer

   (a) $\left\lceil \frac{n+1}{2} \right\rceil$ guesses

   Explanation

   Assume w.l.o.g. that our guesses are in ascending order. Then if the number chosen is m, we'll guess it on the mth attempt. Suppose we played the game n times, with each time being a different number, then the total number of guesses we'd make in the course of this would be $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. The average guesses in each game are thus $\left\lceil \frac{\frac{n(n+1)}{2}}{n} \right\rceil = \left\lceil \frac{n+1}{2} \right\rceil$.

   (b) n guesses

   Explanation

   The probability of any guess being right is $\frac{1}{n}$. All the guesses are independent of each other. The probability that we've guessed it after k guesses is hence $\sum_{i=1}^{k} \frac{1}{n} = \frac{k}{n}$. We want the probability that we've guessed it to be 1, i.e. $k = n$.

2. Suppose that you are only allowed to use the 26 lower-case letters to construct passwords, and that passwords must be at most n characters long, for $n = 4, 6, 8$. How long would a brute force attack take on average if it takes one microsecond to check a password? Repeat the exercise but assuming all alpha-numeric and punctuation characters are allowed.

   Answer

   Let's solve the general case before getting down to specifics. Denote the number of possible letters as α, then there are α^n possible passwords of length n. On average, attackers would expect to run through half the passwords before finding the correct one, which would take $\frac{1}{2}\alpha^n$ microseconds.

   Our table when $\alpha = 26$ is thus:

n	Time
4	228488 microseconds $\approx$ 0.2 seconds
6	154457888 microseconds $\approx$ 2.6 minutes
8	104413532288 microseconds $\approx$ 29 hours

   If all alpha-numeric and punctuation characters are allowed, then $\alpha = 26 \times 2 + \#\{0, ..., 9\} + \#\{`.`, `,`, `;`, `:`, `!`, `?`\}$, say, i.e. $\alpha = 68$. Whence our table looks like:

n	Time
4	10690688 microseconds $\approx$ 10.7 seconds
6	98867482624 microseconds $\approx$ 27.5 hours
8	457163239653376 microseconds $\approx$ 14.5 years

3. (Gollmann, Exercise 3.7.) If you are required to use several passwords at a time, you might consider keeping them in a password book: a protected file containing your passwords, access to which is controlled by a master password. Does such a scheme offer any real advantages? (Write brief notes to answer this question.)

Answer

The advantage is that you only have to remember the master password but you have the additional security which comes from having more than one password: it's clearly better than just having one password which you use for everything. The disadvantage is that if someone finds out the master password, they've got access to all your other passwords. In that sense, it's better to try and remember all your passwords individually without writing them down anywhere (though admittedly you do need a good memory for this).

4. Suppose passwords are checked character by character, by comparing with a stored password, and access is denied as soon as a mismatch is found. Suppose further that the user can observe page faults, and can decide where in a memory page the password guess is stored. Describe how these facts can be exploited to lead to a password guessing attack. What is the expected cost of this attack (in terms of the number of incorrect guesses)?

Answer

(a) This is an example from Gollmann (p.84). The attack works as follows: starting with $i = 1$, the attacker places a password guess in memory such that the first i characters are in one page and the remaining characters are in the subsequent page. If he observes a page fault, he knows that access wasn't denied before the next page had to be read, i.e. that the i characters in the first page were correct. In that case, he shifts the password guess along by one character (so now there are $i + 1$ characters in the first page) and tries again. Eventually either he'll gain access, or access will be denied before a page fault is observed. In the latter case, the attacker simply replaces the $(i + 1)^{th}$ character of the password guess with the next character in sequence and tries again. He might have to run through all of the characters, but if he persists long enough then he's bound to get the right one, at which point he can move on to trying to find the $(i + 2)^{th}$ character, and so on.

(b) Suppose there are n possible characters in the alphabet. Similarly to question 1(a), we'd expect that $\left\lceil \frac{n+1}{2} \right\rceil$ guesses would be needed on average for each character in the password. Suppose the length of the password is m, then we require $m \left\lceil \frac{n+1}{2} \right\rceil$ guesses on average to guess the right password (so I suppose the cost in terms of the number of *incorrect* guesses would be one less than that). This compares very favourably[1] with the expected cost of a normal brute-force attack (running through all the possible permutations), which would be something like n^m.

---

[1]Obviously if it didn't compare favourably with it, there wouldn't be much point to the method in the first place.

5. For the problem description, see the sheet (it's too long and boring to type out!)

Answer

Listing 1: Code for the Caesar cipher cracker

```java
 1  import java.util.*;

    class Decryption
    {
      public String decryptedText;
      public int key;

      public Decryption(int key, String decryptedText)
      {
        this.key = key;
11      this.decryptedText = decryptedText;
      }
    }

    public class CaesarCracker
    {
      private LinkedList[] buckets;
      private Dict dict = new Dict();
      private String cipherText;

21    private void do_decryption(int i)
      {
        String decryptedText = new CaesarCipher().encrypt(i, cipherText);

        String[] words = decryptedText.split(" ", 0);
        int score = 0;
        for(String w: words) if(dict.inDict(w)) ++score;

        buckets[score].add(new Decryption(i, decryptedText));
      }
31
      public CaesarCracker() throws Exception {}

      public void run(String[] args)
      {
        cipherText = args[0];

        int wordCount = cipherText.split(" ", 0).length;
        buckets = new LinkedList[wordCount+1];
        for(int i=0; i<=wordCount; ++i) buckets[i] = new LinkedList();
41
        for(int i=1; i<=26; ++i)
        {
          do_decryption(i);
        }

        for(int i=buckets.length-1; i>=0; --i)
        {
          LinkedList bucket = buckets[i];
          for(Object o: bucket)
51        {
            Decryption d = (Decryption)o;
            System.out.println(d.key + ", " + i + ": " + d.decryptedText);
          }
        }
      }

      public static void main(String[] args) throws Exception
      {
        if(args.length != 1)
61      {
          System.out.println("Usage: java CaesarCracker \"<text>\"");
          System.exit(0);
        }

        CaesarCracker cc = new CaesarCracker();
```

```
    cc.run(args);
  }
}
```

6. Recall that a hash function h is a *cryptographic* hash function if (i) given y, it is computationally infeasible to find x such that $h(x) = y$; and (ii) it is computationally infeasible to find x_1 and x_2 such that $x_1 \neq x_2$ and $h(x_1) = h(x_2)$.

Suppose f and g are cryptographic hash functions. Which of the following are cryptographic hash functions? In each case, justify your answer carefully.

(a) $h_1(x) = f(g(x))$
(b) $h_2(x) = f(x) + g(x)$

For those functions that you believe to be cryptographic hash functions, is it necessary for *both* f and g to be cryptographic hash functions?

Answer

(a) h_1 is a cryptographic hash function

Proof

Suppose h_1 isn't a cryptographic hash function. That is, given y, it's computationally feasible to find x such that $h_1(x) = y$. In other words, it's computationally feasible to find x such that $f(g(x)) = y$. This implies that it's computationally feasible to find $z = g(x)$ such that $f(z) = y$, which is a contradiction since f is a cryptographic hash function. $\square$

Note

It's only necessary that f be a cryptographic hash function for this to hold. Whether or not g is a cryptographic hash function is irrelevant.

(b) *Lemma 1*

If f is a cryptographic hash function, then so is $-f$.

Proof

Suppose $-f$ isn't a cryptographic hash function. In other words, given y, it's computationally feasible to find x such that $-f(x) = y$. It's equally computationally feasible to find x such that $-f(x) = z$, where $z = -y$, i.e. to find x such that $-f(x) = -y$ or $f(x) = y$. But f is a cryptographic hash function, so we've derived a contradiction and $-f$ must be a cryptographic hash function as well. $\square$

Lemma 2

h_2 isn't (in general) a cryptographic hash function (i.e. it's not a cryptographic hash function for all possible choices of f and g)

Proof

Suppose f is a cryptographic hash function and $g = -f$ (whence it too is a cryptographic hash function). Then:

$$h_2(x) = f(x) + g(x) = f(x) - f(x) \equiv 0$$

Now it's obviously computationally feasible to find x such that $h_2(x) = 0$: any x will do! Furthermore, $h_2(x)$ can't take on any value other than 0, so for any y such that $h_2(x) = y$, it's computationally feasible to find an[2] argument x to h_2. Hence h_2 isn't a cryptographic hash function. $\square$

---

[2]Obviously not 'the', since any value of x would work, as it happens!