Modern Cryptography, Part II

This section of notes covers a few odds and ends that are important in modern cryptography.

Measuring security

The strength of a cipher is often measured by the number of operations needed to break it. This number of operations is often given as a number of bits. For instance, 16-bit security means that a cipher takes at least $2^{16} = 65,536$ operations to break. The term "operations" is purposely left a little vague. An operation could be something as simple as multiplying two numbers, but it could be considerably more complex.

For example, the Caesar shift cipher has 25 possible keys, and we can break the cipher by trying each key. If we define an "operation" in this context to be trying out a key to see if we get readable English, then the cipher takes at least $2^4 = 16$ but no more than $2^5 = 32$ operations, so we can say it provides around 4 to 5 bits of security. The measurements are given for the worst-case scenario. It's always theoretically possible that the first key we try could work, but that's rare, and usually the worst-case is the most useful measure.

Notice that 3-bit security means something takes at least $2^3 = 8$ operations to break and 4-bit security means at least $2^4 = 16$ bits are needed. So 4-bit security requires double the operations of 3-bit security. This is true in general, $n+1$-bit security is twice as strong as n-bit security. In particular, 130-bit security is not just 30% more secure than 100-bit security, it's $2^{30} \approx$ *one billion times more secure*.

We can do a little back of the envelope calculation to estimate what type of security we could break with a simple laptop. A typical laptop has a CPU that runs around 3 GHz, which means that it can do around 3 billion basic operations (adds, bit shifts, etc.) in a second. The operations needed to crack a cipher are more complex these this, so let's go with a rough estimate of 100 million cipher-cracking operations per second as what a laptop could handle. If we run it for 24 hours straight, then the total number of operations we could do $100,000,000 \cdot 24 \cdot 60 \cdot 60 = 8.64 \times 10^{12}$. To get the number of bits we can take the log base 2 of this, which around 43. So, as a rough estimate, a laptop crack around 43-bit security.

If we get a few GPUs, which are much faster than ordinary CPUs for computations, along with a few friends with GPUs, we could probably break security in the middle 50s of bits. Some medium to large tech companies probably have enough spare computing power to break cryptography into the 60s and possibly into the 70s. Large nation states probably have enough power to get into the 80s. The most computing power I know of is the entire bitcoin network. If bitcoin miners were to direct their computing power at cracking a cipher, they could probably crack 100-bit security.

Most modern cryptographic methods use at least 128-bit security. This currently requires close to a billion times more power than even the bitcoin network has. But computing power has been increasing exponentially for some time, so within a few decades this may be reached. However, 256-bit security, which is often used, is probably out of reach, based on some physics. Just for emphasis, here are a few powers of 2. These are the number of operations needed to break that level of security. Notice especially how large 2^{256} is.

```
2^43 =  8796093022208
2^100 = 1267650600228229401496703205376
2^128 = 340282366920938463463374607431768211456
2^256 = 115792089237316195423570985008687907853269984665640564039457584007913129639936
```

Pseudorandom numbers

Pseudorandom numbers are random numbers generated by a mathematical process. They are designed to look random, but they aren't truly random because they come from a predictable mathematical formula. The *pseudo* in the name means "false" or "fake".

Some pseudorandom number generators (PRNGs) are cryptographically secure, while others, particularly the standard ones built into most programming languages, are not. With the insecure ones, it is possible to observe a few random numbers and use them to predict all of the future and past numbers generated by the PRNG. For a PRNG to be cryptographically secure, it needs to be impractically difficult to predict new values based on past values. There also should not be any biases in the numbers generated. For instance, if a certain values are statistically more likely to appear than others or if certain values tend to follow or not follow others, then that could lead to a means of breaking ciphers implemented with a biased PRNG.

To understand how PRNGs work, let's look at a simple type of PRNG called a *linear congruential generator*. This is what Java and C/C++ use to generate random numbers. The mathematical formula used is $(ax+b) \bmod m$. For our simple example, we will choose $a=7$, $b=4$, and $m=11$.

PRNGs always have a *seed*. This is an initial value used to start the random number generation. Different seeds produce different sequences of random numbers. Let's start with the seed $x=8$. We then apply the formula $(ax+b) \bmod m$ to get $(7\cdot 8+4) \bmod 11 = 5$. So 5 is our first random number. We then use this number as our new x and plug it into the formula. This gives $(7\cdot 5+4) \bmod 11 = 6$. Then we use 6 for x in the formula to get $(7\cdot 6+4) \bmod 11 = 2$. We keep repeating this process to get more and more random numbers. In this simple example, the numbers will pretty soon start to repeat. To prevent this, m should be chosen to be very large. Java uses $a = 25214903917$, $b=11$, and $m = 2^{48}-1$.

Python uses a PRNG called the Mersenne Twister. It is harder to break than a linear congruential generator, but still easy to break by people who know what they are doing.

Birthday problem

Here's a question: If we continually generate random numbers from 1 to 100, how long will it take until we start seeing repeats? In the worst-case scenario, it could take 100 numbers, but the odds of that are astronomically small. The surprising fact is that repeats will start becoming likely after only about 10 numbers are generated.

We could then ask about generating random numbers from 1 to 10,000. Again, the surprising fact is that repeats start becoming likely after only around 100 numbers are generated.

The most famous instance of this is the *birthday problem*. It asks how many people you have to have in a room before there's a 50/50 chance of two people in the room having the same birthday. Despite the fact that there are 365 (or 366) possible birthdays, the 50/50 point is reached at only 23 people in the room. Once you get to 40 people in the room, there's almost a 90% chance of a shared birthday, and at 100 people it's a 99.99997% chance.

This strange fact about repeats turns out to have several applications in computer security, especially in cryptography. The general rule of thumb turns out to be this:

> *When generating random numbers from 1 to n, repeats start becoming likely after about $\sqrt{n}$ numbers are generated.*

Here is a practical example. Suppose you are running a website and you want to randomly assign session IDs to those users. A simple approach would be to use random Java integers. The Java `int` type is 32 bits. The number of possible integers is then 2^{32}, which is around 4 billion. According to the birthday problem, repeats will start becoming likely around the point when $\sqrt{2^{32}} = 2^{16}$ session IDs are generated. This is only around 65,000. So even though you have around 4 billion possible IDs, repeats will start happening considerably sooner.

If you want to test out the birthday problem for yourself, there is a simple tool at `https://www.brianheinold.net/repeats.html`.