London Computation Club - New Turing Omnibus c.37: Public Key Cryptography

Big Picture

Maths is required. Breaking is often assumed to be brute-forcing the key once you know the algorithm, so you're only as secure as advances in computational power make it easy to make many guesses per second. However, as we see there's another vector which is identifying a weakness in the algorithm which mean our assumed NP-complete to solve algorithm actually isn't.
Not covered: weakness in key generation, shared keys, &c, &c, &c.

Codes in general

Interesting to note the assumption that secrets are for military and financial or industrial use, whereas today we might think more about it for personal privacy.

The trivial code described in the opening is the Caesar Cipher (http://practicalcryptography.com/ciphers/caesar-cipher/). There's a throwaway comment about cracking the code being simple even if you don't know the algorithm. Cracking a Caesar cipher is trivial enough that I'm sure I've encountered it several times in all the video games I've played (most recently the iOS game Hundreds which includes a bunch of cipher puzzles as "Easter eggs" between levels: https://en.wikipedia.org/wiki/Hundreds_(video_game)). Cryptanalysis is probably worth a chapter (or book!) on it's own. Essentially though we use the characteristics of the crypted message to work out the algorithm and the expected contents of the message (e.g. the language it's in) to work out the key. If the algorithm is simple enough we might even be able to brute force the key once we know the algorithm (e.g. Caesar Cipher has a finite number of keys based on how many characters we rotate so we can easily try all the keys till we get something approaching a result once we've worked out that it's a Caesar Cipher). Other ciphers betray similar characteristics, we just might need longer and longer to brute force.

I guess that the realisation about PKC is that the algorithm is known to everyone, but the keyspace is so massive that the only option is brute force, and that will take so long because of the cunningness of the algorithm. As we see later, you do have to have a very cunning algorithm though, as if there is a weakness you didn't realise at first, a massive keyspace doesn't help and Maths will tear open your secrets (or better computers will brute force it).

Recent worries about the NSA in the past few years were that they'd either got a massive quantum computer that could brute force the keys quicker than we'd expect (so we need to increase our key-lengths) or that they'd magicked up some dark Maths that meant they could exploit a hitherto unknown weakness in the algorithm. Either would be worrying. (FWIW I think that current theory is that they'd brute-forced a few weak keys that happened to be widely used).

Public Key Crypto

The mention that *k* (our public key) is easy to compute from *k'* (our private key) but *k'* is hard to compute from *k* seems straightforward. What I'm not sure about is the statement that from a given *k'* we can easily generate new *k* (whenever people have talked about key issues they've always said generate a new pair, not a new public key - I could just not understand the relationship between public and private *as it is used today* though).

Subset-Sum Problem

Given n numbers: a_1, a_2, a_3, … a_{n-1}, B; do any combination of a_i sum to provide B?

A good opportunity here to work through the example in the book on the whiteboard. Maybe even to see Chris Patuzzo's solver? Is it harder or easier that it just has to be a sum problem? Is this what the numbers round in Countdown is? Is it possible that Countdown contestants are actually brute forcing keys for GCHQ?

Hellman-Merkle-Diffie

Talks about the public key as being a set of integers. This is not how I usually think of my public key, is that an artefact of the HMD algorithm, or is it actually what my public key is I just don't know it?

Encrypting

Another good opportunity to explore the problem via the whiteboard:

1. Transform the message into a representation useful for the algorithm.
    a. turn everything into binary
    b. partition into blocks of length *n* (where *n* is the number of integers in our public key)
2. convert each block into a number:

```
def crypt(block, key)
  key.map.with_index do |a, i|
    block[i] * a
  end.reduce(:+)
end
```

   where `block` is an integer and `key` is an array (of integers).
   (TIL: `Fixnum#[]` in ruby gives the binary digit at that position (0 is the LSB).)
3. send each crypted number over insecure channels in order

4. this is safe because even with knowledge of the public key and the ciphertext finding the set of numbers from the key that add up to the ciphertext number is non-trivial (it's the subset sum problem).

The example uses a key of length 7 and thus converts each letter in the plaintext to 7-bit ascii which is convenient. In reality much larger key lengths are used as 7 numbers only leaves 128 possible combinations to try before we work out the set that add up to the ciphertext. Even in 80's this was trivial enough to brute force. A larger key means more permutations, means it takes longer.

Interesting argument about why bother wasting 100 years of effort to decrypt something that won't be useful in 100 years. Is it still true that encrypted things are only useful for time sensitive information?

Aside: generating *k* from *k'*

The *i*th digit of *k* is generated as follows: $a_i = w * a'_i \bmod m$
It's not clear (to me) where *w* and *m* come from, if they are part of the private key, or some part of the algorithm, or … For now I'm taking it on faith that they JUST ARE.

Decrypting

Another place to do a worked example.

Given a ciphered digit *B* we push it through the same algorithm as we used to generate the public key to get *B'*, with a notable exception. We use w^{-1} not *w*, as follows:

$B' = B * w^{-1} \bmod m$

Where w^{-1} is such that $w * w^{-1} = 1 \bmod m$.

No, me neither. What we're talking about is the Modular multiplicative inverse: https://en.wikipedia.org/wiki/Modular_multiplicative_inverse. It's complicated but there is a process for working it out. Note that it states that there is only a w^{-1} for *w* if and only if *w* and *m* are coprime. Our first hint so far that prime numbers are important to crypto. Not mentioned directly in the book but perhaps our '80s Maths background would have told us that. Maybe we can explain this on the board (given *w*=3 and *m*=12, a value of w^{-1} is 4) if people want to know the maths, or maybe we can trust that there is a process and it's important that there is the correct relationship between *w* and *m* (e.g. you can't just pick two random numbers and hope it'll work).

Anyway, now we have another subset sum problem to solve to tell us which digits of the private key sum up to make this new *B'*. The nature of the private key means that this isn't as intractable as the standard subset sum, because each number in the key is greater than the one before.

Following the provided algorithm we get an array of length *n* with true (1) or false (0) for each number in the private key if it was used to add up to *B'* or not. This can then be treated as a

n-bit binary number and lo-and-behold, it's the first block of our plaintext. Which in the example's case is a 7-bit ASCII letter. Hurrah!

Why? MATHS.

This is all classic algebra, but the key step that most of us (well, me anyway) probably didn't get immediately is the 2nd to 3rd lines of the *B'* reduction. It relies on re-arranging to cancel out the w mod m and w^{-1} mod m because ($w * w^{-1}$) ≡ (1 mod m) or equivalently ($w * w^{-1}$) mod m = 1 (from https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/modular-inverses).

We should work this through on the board (might have to just agree that the cancellation works, unless there are mathematicians in the room who have the explanation).

Breaking Hellman-Merkle-Diffie

I imagine the proof and algorithm found here is quite complex. We may have to take it on face-value unless anyone in the room has special knowledge, but it comes down to what we suggested before - the algorithm has a weakness that can be exploited.

RSA

Turns out my public and private keys are both pairs of integers. Not quite the array of integers that we have in HMD, but still. TIL.

We should probably try another worked example here if we have time. We should use a public / private key pair of tiny proportions though.

Nice final line: The NSA are trying.