Needham-Schroeder, Kerberos, and Quantum Computing

Summary

A brief introduction to the Needham-Schroeder Protocols, Kerberos, and Quantum Computing.

The Needham-Schroeder Protocols

1978 - Roger Needham and Michael Schroeder at Xerox Palo Alto Research Center (PARC) publish *Using Encryption for Authentication in Large Networks of Computers*.

Protocols for authentication in large networks using either symmetric key (e.g. block or stream ciphers) or asymmetric key (public key — RSA, etc.). We will only discuss the symmetric key protocol.

Central to the protocol is a trusted *Authentication Server* (AS). The AS manages a set of users and knows each user's *Secret Key*. The secret key is typically derived from a user password — the AS knows the derived key, not the password itself.

Notation: we will use E(X; K) to denote the encryption of X using key K. Think of the encryption algorithm as AES (in 1978 it would have been DES). If we need to specify a specific key, e.g. the key belonging to Alice, we will use subscripts: K_A.

Suppose Alice (A) wants to access a service run by Bob (B) and both Alice and Bob are managed by the same authentication server (AS) so that AS knows the secret keys for Alice and Bob, K_A and K_B. First Alice carries out an exchange with the AS:

1. Alice sends her identity, Bob's identity, and a nonce to AS:
   $$A \text{ —> } AS: (A, B, I_A)$$
2. The AS randomly generates a *conversation key* $K_{A,B}$ for Alice and Bob to use in their communication.
3. The AS sends the following
   $$AS \text{ —> } Alice: E(I_A \,||\, B \,||\, K_{A,B} \,||\, E(K_{A,B} \,||\, A; K_B); K_A)$$
4. Alice uses her key to remove the outer encryption, producing I_A, B, $K_{A,B}$, and $T = E(K_{A,B} \,||\, A; K_B)$. She checks that the value of I_A returned by the server matches what she sent in her message (this prevents a server replay attack).

At this point, Alice has the conversation key $K_{A,B}$ that she will use to encrypt her messages to Bob as well as $T = E(K_{A,B} \,||\, A; K_B)$; the value T is called a *ticket* in Kerberos. Now Alice can initiate communication with Bob:

1. Alice sends Bob the ticket T:
   $$Alice \text{ —> } Bob: E(K_{A,B} \,||\, A; K_B)$$

2. Bob knows his own secret key K_B, so he can decrypt the ticket and extract A and $K_{A,B}$. He checks that A from the ticket matches the identity of Alice.

So now both Alice and Bob know the conversation key $K_{A,B}$, so they can use it to encrypt whatever data they wish to exchange.

Since Alice sent the ticket to Bob, she has confidence that only she and Bob know $K_{A,B}$. Therefore, any data she receives from Bob that decrypts correctly using the conversation key must actually be from Bob.

The situation is not as good for Bob. Unless Bob keeps a record of all conversation keys he has ever used to talk to Alice, he can't be sure that the ticket hasn't been replayed — that is, he doesn't know for sure that someone hasn't sniffed a ticket from the network in the past only to replay it now. This can be remedied with one additional exchange of information:

3. Bob sends Alice a nonce encrypted with the conversation key:

$$\text{Bob} \rightarrow \text{Alice: } E(I_B; K_{A,B})$$

4. Alice responds with a small modification of I_B, say subtracting one, also encrypted with the conversation key:

$$\text{Alice} \rightarrow \text{Bob: } E(I_B - 1; K_{A,B})$$

Bob can decrypt the value sent by Alice, to see that it really is the encryption of $I_B - 1$; this assures him that Alice really does know the conversation key and that the ticket was not simply replayed.

The paper of Needham and Schroeder goes on to describe a similar algorithm using public key cryptography; how to support multiple authentication servers; support for one-way communication (email); and support of signatures by the AS.

Kerberos

The protocols of Needham and Schroeder are the basis for *Kerberos*, a system for managing access to distributed resources that was developed at MIT in the 1980s.

Microsoft's Active Directory is based on Kerberos.

A main difference between the basic Needham-Schroeder protocol and Kerberos is the addition of a *Ticket Granting Server* (TGS) — rather than having the AS manage all access to other computers, the AS issues a *Ticket Granting Ticket*, which the client can deliver to the TGS to be given access to other resources (file servers, etc.). The initial exchange between Alice and the AS, Alice will request access to the TGS and will receive a TGT (ticket for the TGS):

$$\text{AS} \rightarrow \text{Alice: } E(I_A \,||\, \text{TGS} \,||\, K_{A,TGS} \,||\, E(K_{A,TGS} \,||\, A \,||\, \text{STIME} \,||\, \text{EXPIRE} \,||\, \text{CS}; K_{TGS}); K_A)$$

Note the addition of a start time (STIME) and expiration date and time (EXPIRE) in the TGT. A checksum (CS) may also be included. This will allow the client (Alice), to reuse the TGT for some period of time to gain access to other network resources via a request to the TGS.

Kerberos also uses a different protocol to authenticate the client (Alice) to the TGT. The client initiates the authentication protocol:

1. Alice sends the ticket and an *Authenticator*:
   Alice —> Server: TGT, E(A || SYSTIME || SYSNAME || CNTR; $K_{A,TGS}$)
2. The TGS decrypts and validates the TGT; the Authenticator is decrypted using the session key extracted from the TGT and the TGS checks that the SYSTIME is "close" to its system time and that the SYSNAME matches Alice's system name.

Now that Alice and the TGS have a session key and the client is authenticated, Alice can send requests for access to other servers to the TGS. Suppose Alice wants access to a service provided to Bob. She sends a request to the TGS that is just like her initial request to the AS, but with "B" instead of "TGS". The TGS *must* check that the checksum CS is correct. If the request is valid, the TGS generates a ticket for the requested service (Bob) and returns it to the client (Alice). Alice may now use the ticket to gain access to services hosted by Bob.

Quantum Computing

Quantum computers are good at breaking public key systems — this is not because they "try all keys at once." Even if they could do that, how would you know which key was the right key?

The reason quantum computers are good at breaking public key systems is because they have mathematical structure that just happens to make them susceptible to quantum algorithms, especially *Shor's Algorithm.*

Shor's algorithm uses the *Quantum Fourier Transform* (QFT); Fourier Transforms should be familiar to the Computer Engineers and maybe the mathematicians. They are used to extract information about periodicity (frequency) from data. Well, they're used for a lot of other things, too, but that's a common use.

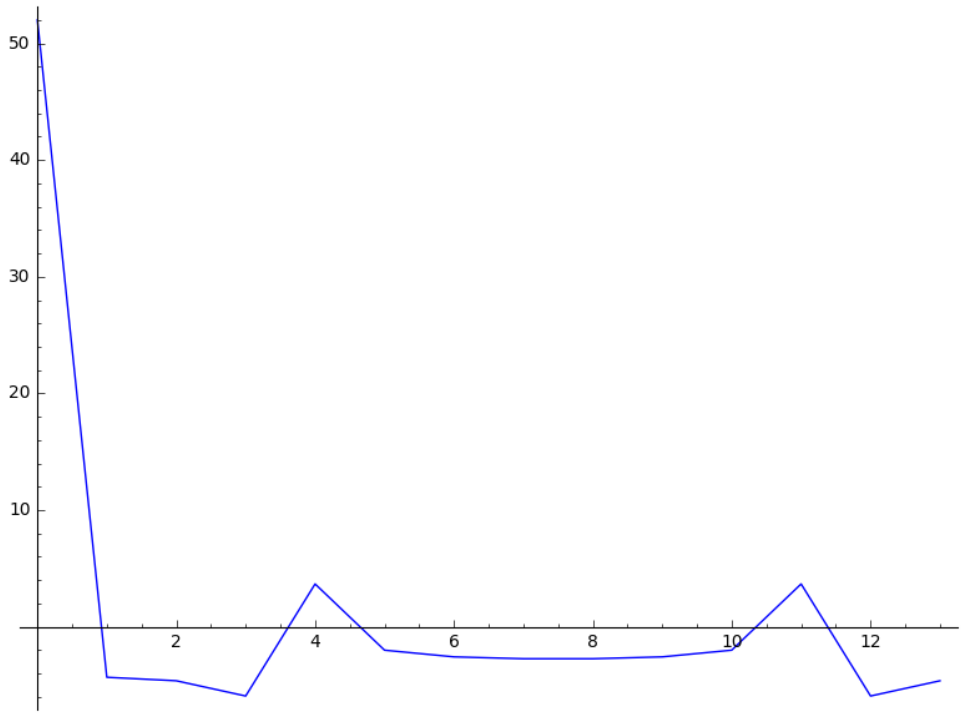
We can apply this to RSA. Recall that N = pq and Euler's totient function is phi(N) = (p-1)(q-1). There is a Theorem due to Euler that tell us that if x is a number between 0 and N that is not divisible by p or q, then the *period* of x divides the totient.

We can do some small examples of the Fourier Transform. I'm using the Sage math package. In this first example, we compute the powers of $x = 2 \bmod N = 15$, print the sequence of powers, and plot the Discrete Fourier Transform (DFT). Looking at the sequence, we can observe that the period of x is four, and in the DFT we see a good-size spike a the value four.

In [82]:
```
# Compute DFT of x^i mod N
# try x = 2, x = 7, x = 8

x = 2
N = 3*5  #phi(N) = 8

# Construct Indexed Sequence and compute DFT
J = range(N)
A = [ ZZ(x^i % N) for i in J ]
s = IndexedSequence(A, J)
t = s.dft()

# Print the data and plot the real part of the DFT

print(A)
tr = [real_part(i) for i in t.list()]
TR = IndexedSequence(tr, J)
TR.plot()
```

```
[1, 2, 4, 8, 1, 2, 4, 8, 1, 2, 4, 8, 1, 2, 4]
```

Out[82]:

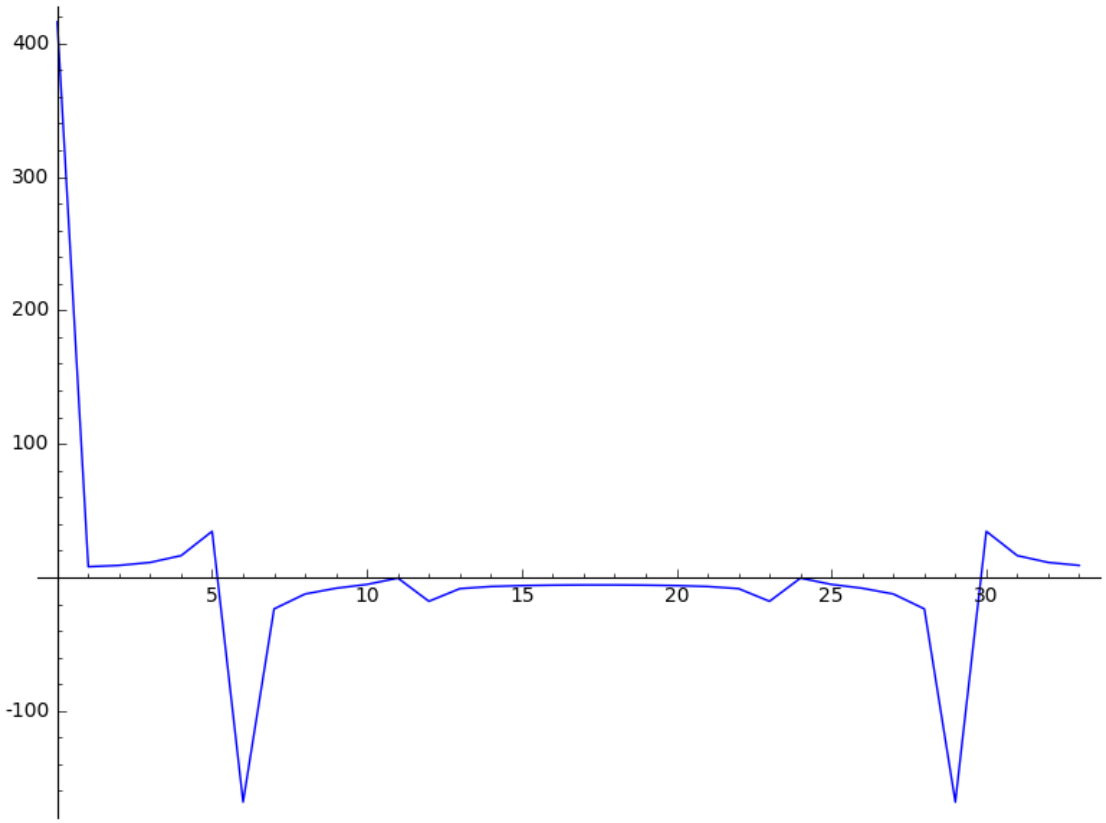


The second example is similar, but with $x = 9$ and $N = 35$. We can see from the sequence of powers that x has period six and there is a dominant spike in the DFT at six.

In [83]:
```
# Try x = 3, x = 4, x = 9
x = 9
N = 5*7  # phi(N) = 24

J = range(N)
A = [ZZ(x^i % N) for i in J]
s = IndexedSequence(A, J)
t = s.dft()

print(A)
tr = [real_part(i) for i in t.list()]
TR = IndexedSequence(tr, J)
TR.plot()
```

```
[1, 9, 11, 29, 16, 4, 1, 9, 11, 29, 16, 4, 1, 9, 11, 29, 16, 4, 1, 9
, 11, 29, 16, 4, 1, 9, 11, 29, 16, 4, 1, 9, 11, 29, 16]
```

Out[83]:



So, if we could determine the periods mod N for a bunch of different x's, say x_1, x_2, …, then we would learn about lots of divisors of $\phi(N) = (p-1)(q-1)$, which we could ultimately piece together to determine $\phi(N)$ itself. Well, that's as good as breaking RSA. Why? Suppose we can determine $\phi(N)$. Then

$$\phi(N) = pq - p - q + 1 = (pq + 1) - (p + q) = (N + 1) - (p + q)$$

$$(N + 1) - \phi(N) = p + q, \text{ which gives } q = (N + 1) - \phi(N) - p$$

$$N = pq = p((N + 1) - \phi(N) - p) = -p^2 + ((N + 1) - \phi(N))\, p$$

This gives a quadratic equation in p with known coefficients:

$$-p^2 + ((N + 1) - \phi(N))\, p - N = 0$$

We can solve this quadratic to find p (and thus q). So the question of breaking RSA is reduced to the question of finding the periods of numbers mod N, which is exactly what the QFT is good at!

The *quantum magic* creates a superposition of states which are the powers of x mod N; the QFT finds the period of this data; repeat for a different value of x until you have enough information to determine $\phi(N)$.

One thing that quantum computers are not particularly good at is finding keys for symmetric algorithms such as AES. With known algorithms, a quantum computer would reduce the work for a brute-force attack on AES to the square-root of the key space size. So, the work to break 256-bit AES would be 128 bits, or 2^{128} computations, which is still huge.

This all brings us back to Kerberos: if a quantum computer were built tomorrow, Kerberos using 256-bit AES would still be secure, but PKC-based systems (RSA, Diffie-Hellman) would not. Kerberos looks a bit old-fashioned, but it's good to keep around!

References

Roger Needham and Michael Schroeder, *Using Encryption for Authentication in Large Networks of Computers*, http://dl.acm.org/citation.cfm?doid=359657.359659

RFC 3961, *Encryption and Checksum Specification*, http://www.ietf.org/rfc/rfc3961.txt

RFC 4120, *Kerberos V5*, http://www.ietf.org/rfc/rfc4120.txt

Scott Aaronson, *Shor, I'll do it,* http://www.scottaaronson.com/blog/?p=208