Let's begin by recalling our overall setting and goals:

Example scenarios:
- Alice ----(public network)----> Bob
- Alice ------->(disk)----------> Alice

Overall goals
* CONFIDENTIALITY: Keep other people from reading Alice's messages/data
* INTEGRITY: Keep other people from tampering with Alice's messages/data
* AUTHENTICITY: Ensure that data purportedly from Alice really is from Alice

==========

THE SHORTCOMINGS OF SYMMETRIC KEY CRYPTOGRAPHY

We have seen how symmetric key crypto achieves confidentiality (through block ciphers) and integrity (through MACs). It also gave us a limited form of authenticity:

1. Alice and Bob establish a shared symmetric key and agree to keep it secret.
2. Alice sends Bob a message with a valid MAC.
3. If Alice and Bob both kept the key secret, then thanks to the MAC's resistance to existential forgery, only Alice or Bob could have generated the MAC; if Bob knows that he did not generate it, then he knows that it must have come from Alice.

This is limited in a few ways. First, Alice's message could only be authenticated by Bob, because they are the only ones who know the key. If she wanted to broadcast a message to N people, then she would have had to establish N symmetric keys, and generate N unique MACs. What happens if Alice decides (or is forced) to go offline---how can she prove to the N+1'st person that the message is from her? What we would like to have is a way to achieve authenticity without requiring that all users be online all the time to establish symmetric keys; ideally, the messages would remain authentic long after the sender can do any more work.

WE WILL OVERCOME THIS LIMITATION WITH A NEW "BLACK BOX" CALLED PUBLIC KEY ENCRYPTION.

The second major limitation of achieving authenticity with a protocol like the one above is establishing the key in the first place! The process of establishing the key must itself achieve our three security goals:

- Key exchange must achieve confidentiality, or else an eavesdropper could learn the key.
- It must achieve integrity, or else an active attacker could tamper with the keys in transit.
- Finally, it must itself achieve authenticity, or else Bob could be establishing a key with someone other than Alice.

We have here a circular problem: we seem to need authenticity in order to achieve authenticity. This might seem impossible (and without making a few extra assumptions, it is!), but let's first consider a simple key exchange protocol that is capable of achieving all of these properties:

1. Alice and Bob meet *in person*.
2. They run the Diffie-Hellman key exchange protocol by speaking it aloud.

For this to work, Alice and Bob must be able to identify one another in person. If they know one another, then this is relatively straightforward (but possibly more difficult if one of them has an identical twin).

This is feasible for friends to establish keys and achieve authenticity, but what we want is the ability to secure communication online; how are you expected to meet amazon.com "in person"? How would you know that it was actually amazon.com you were meeting with? Perhaps you have a friend who works there, but what about bobs-socks-emporium.com? Moreover, with such a protocol, you would be expected to meet with everyone you would want to authenticate in the future. This does not scale.

Let us now consider an alternative protocol which will get us better properties through the use of TRUST:

0. Suppose that both Alice and Bob trust a third party, Trent.
1. Alice and Bob both each follow the above meet-in-person protocol, and separately establish keys with Trent: let's call the Alice-Trent key K_AT and the Bob-Trent key K_BT.
2. When Alice wishes to send a message m to Bob, she sends to Trent E(K_AT, m) (along with the appropriate MAC).
3. Trent decrypts this, and sends to Bob E(K_BT, m || "From Alice")

First the good news: if there is some Trent that both you and amazon.com trust, then you could use him as an intermediary to establish a key without having to "meet with" amazon.com in person. Now the bad news:

Trust is absolutely fundamental to a protocol like this: Trent could easily forge messages from Alice, and there is nothing in place to stop him from doing so.

Moreover, Trent learns what Alice and Bob are sending to one another. Note, however, that Alice and Bob can overcome this limitation by running Diffie-Hellman over this protocol to establish a shared key that only they know about, allowing them to subsequently decouple from Trent. But to perform this key exchange, Trent must be online. Thus, another way that Alice and Bob are implicitly trusting Trent is to remain online whenever they need him to communicate with one another.

Because it requires Trent to be online all the time, this protocol is not practical in a setting like the Internet (as we will see, however, there are some settings where it is feasible, like authenticating users within a more constrained network, like within an enterprise).

But let's step back for a moment and take stock of what this protocol showed us: through the application of TRUST, we reduced the problem of authenticated key exchange to the problem of requiring a trusted third party to be ONLINE.

Note that this was the same exact issue we had earlier: for Alice to authenticate messages to N people, she had to be online to establish keys with all of them.

So ultimately what we seek is a way for Alice (or Trent) to prove that they sent a message without having to remain online to perform key exchanges every time a new party joins the system.

Enter public key cryptography.

==========

PUBLIC KEY CRYPTOGRAPHY (aka asymmetric key cryptography)

Our next black box provides a different means of performing encryption and decryption. The fundamental difference with symmetric key crypto is that, instead of using a single shared key for both encrypting and decrypting, we will be using a PAIR of keys: one key for encrypting a message, and another for decrypting that message. As such, this is referred to as "asymmetric key cryptography", or more commonly as public key cryptography. In

general, such a mechanism consists of three algorithms:

(1) A KEY GENERATION algorithm G

- Inputs: a source of randomness and a maximum key length L
- Outputs: a key pair, both of length <= L
  * PK - the "public key"
  * SK - the "secret key" (aka "private key")
- This algorithm is randomized: calling it more than once must return different outputs each time with extremely high probability.

(2) An ENCRYPTION algorithm E(PK, m)

- Inputs: the public key PK and plaintext message m. The message is no longer than the maximum key size L.
- Outputs: a ciphertext c
- This algorithm is PROBABILISTIC: calling it with the same inputs (PK,m) will result in different ciphertexts. It typically achieves this by including random padding. For example, the well-known RSA algorithm (provided later) is technically deterministic (for a given set of inputs, it will always provide the same output---that is why, strictly speaking, RSA is never really used in practice. What is used instead is RSA-PKCS, which first tacks on some random padding to the message before invoking RSA. At a high level, you can think of public-key encryption as providing its own IV.

(3) A DECRYPTION algorithm D(SK, c)

- Inputs: the secret key SK and ciphertext c
- Outputs: the original message m
- This algorithm is also DETERMINISTIC: provided the same inputs (SK,c), it must return the original message.

In terms of security and correctness, we want very similar properties that we had with symmetric key crypto, but the key difference (pun intended) is that with public key crypto, we assume that the attacker is allowed access to the public key. Let's make this a bit more rigorous:

CORRECTNESS:
A public key crypto scheme is correct so long as

D(SK, E(PK, m)) = m

That is, decrypting an encrypted message should yield the original message.

SECURITY:
The encryption algorithm should approximate a one-way trapdoor function: E(PK, m) should appear random (and thus we get our nice property that changing a single bit in m should result in each bit in the ciphertext changing with probability 1/2). The trapdoor here is the secret key SK: only by knowing SK can one "invert" E, even if everyone knows the public key PK.

Like with symmetric key mechanisms, these are also subject to brute force attacks. This is where the length of the keys comes into play: generally speaking, longer keys take longer to attack in a brute force manner. Recommended key lengths increase over time to reflect the computational power available to attackers. For RSA, a public key crypto scheme which we'll see in more detail later, key lengths of 1024 bits are now considered crackable by a reasonably powerful attacker; 2048 bits are expected to be crackable within the next couple decades, after which using 3072

bit keys will be necessary.

A BASIC PUBLIC KEY CRYPTO PROTOCOL

Given this black box, we can now start to develop protocols that will allow us to achieve properties that symmetric key crypto alone could not. Let's begin by considering how Alice sends an encrypted message to Bob in this new paradigm:

(1) Bob runs the key generation algorithm G to obtain a public/private key pair (PK,SK).

(2) Bob keeps SK secret---he never reveals it to anyone---but he makes his public key (you guessed it) public: he announces it to everyone, e.g., by publishing it in the NY Times.

(3) When Alice wants to send a message m to Bob, she obtains his public key PK and computes c = E(PK, m), and sends it to Bob.

(4) Bob computes m = D(SK, c) to recover the original message.

This high-level protocol captures the general use of public key crypto for encryption and decryption: because the public key is used for encrypting, public keys are often referred to as the "encryption key".

The main property that this approach gives us is that Alice and Bob did not need to have any direct communication with one another to establish a key. In fact, Bob did not have to send any messages whatsoever directly to Alice (he simply "broadcasted" his public key to anyone).

Nonetheless, so long as Bob is doing his job at keeping SK secret (and so long as Alice does her job with picking a proper random nonce with which to pad her message), Alice can be certain that the only person who can decipher the message is Bob.

As far as key exchange goes, this is a strict improvement over symmetric key crypto, so why not use public key crypto for everything?

WHY THIS PROTOCOL IS NOT USED FOR ARBITRARY MESSAGES

The main problem with public key crypto schemes is that they are slow: several orders of magnitude slower than their symmetric key counterparts. As a result, public key crypto schemes are not used for messages of arbitrary size. While one could conceptually derive "modes" of encryption for them (like CBC), such things are not used in practice. Instead, we can leverage the best of both worlds by using a HYBRID scheme:

HYBRID ENCRYPTION

-- To make things clear, let's use E & D to denote encryption
-- and decryption using public key crypto, and let's use e & d
-- to denote encryption and decryption using symmetric key
-- crypto.

(1&2) Proceed as above: Bob generates a public/private key pair (PK,SK) and publishes PK while keeping SK secret.

(3) When Alice wants to send a message m to Bob, she obtains his public key PK.

(4) She then generates a random SYMMETRIC key K and computes c_m = e(K, m) -- that's an efficient symmetric key encryption scheme, e.g., AES in CBC mode.

(5) She finally computes c_K = E(PK, K) -- that is, she encrypts the symmetric key K using Bob's public key PK. At this point, she

throws away K.

(6) She sends c_m and c_K to Bob.

(7) Bob computes K = D(SK, c_K) to recover the symmetric key, and uses this to compute m = d(K, c_m).

The intuition behind the security of such a protocol is that, because Bob is the only one who can decrypt c_K (the encrypted key K), then he is the only one who can decrypt c_m (the encrypted message m).

This gets us the best of both worlds: we benefit from public key crypto in that it did not require Alice and Bob to directly communicate with one another, and we benefit from symmetric key crypto in that it is efficient in encrypting and decrypting an arbitrarily large message m. Public key crypto is only used here to encrypt a small symmetric key K.

Consider what this implies about the relative sizes of symmetric keys and the maximum key sizes used in the key generation algorithm G.

The above protocol is at a high level what is used in practice when sending large messages, e.g., with encrypted email.

==========

WHAT'S MISSING?

The above protocols still lack authentication. How does Alice know that the person who published Bob's public key was really Bob? And when Alice sends an encrypted message to Bob, how does he know that it really was from Alice?

To solve these issues, we will need one final black box: a way to "sign" messages.

==========

DIGITAL SIGNATURES

Suppose that Alice wants to send a message m to Bob, and she wants to provide proof that she was the one who sent it.

A digital signature scheme makes use of a public-private key pair (PK,SK) and consists of two algorithms:

(1) A SIGNING function Sgn

- Inputs: Alice's SECRET key SK and message m
- Outputs: a SIGNATURE s = Sgn(SK,m)

(2) A VERIFICATION function Vfy

- Inputs: Alice's PUBLIC key PK, the message m, and the signature s
- Outputs: Vfy(PK,m,s) = "Yes" if s is a valid signature of m generated by using SK, and "No" otherwise.

Because the secret key SK is used to sign, it is commonly referred to as the "signing key".

At a high level, digital signatures are very similar to MACs, the exception being that we are again using different keys for signing and verifying.

CORRECTNESS:
A digital signature scheme is correct so long as

Vfy(PK, m, Sgn(SK,m)) = "Yes"

SECURITY:
Once again, we have the same notions of security as in the symmetric key setting, even when the attacker has access to Alice's public key: in particular, even after seeing many message/signature pairs, the attacker cannot produce an existential forgery (see the notes on MACs for more details of this).

In addition to having similar definitions of correctness and security, digital signatures share several other similarities with MACs:

Just as with the symmetric key equivalent of MACs, digital signatures do not seek to provide confidentiality---when using digital signatures alone, the message would be sent in the clear. This is useful if Alice wishes to send a message that she would not mind everyone seeing (e.g., ads or a software patch).

Also, like MAC tags, the size of the signature is typically much smaller than the original message. In fact, it *has* to be: the message m and ciphertext c must be no longer than the maximum key length (typically around 2048 bits). Thus, rather than sign the entire message m, one signs H(m), the hash of m using a pre-image resistant hash function H, such as SHA-256 (see the earlier discussion of hash functions for these definitions).

==========

PROPERTIES OF DIGITAL SIGNATURES

To better understand what digital signatures give us, it is useful to compare them to their physical counterparts: handwritten signatures. There are many issues with handwritten signatures that digital signatures overcome. In particular, there are three main properties we want from a signature:

AUTHENTICITY: Bob can prove that a message signed by Alice is truly from Alice.

Handwritten signatures do not readily provide this property because they are easy to forge. Digital signatures overcome this by using one-way trapdoor functions that are difficult to invert without knowing Alice's secret key.

INTEGRITY: Bob can prove that no one tampered with a message that Alice signed.

Even if handwritten signatures were impossible to forge, they still do not provide integrity. To see this, imagine a letter with Alice's signature at the bottom. One could tamper with the letter by adding or removing words, or even by cutting out Alice's signature and pasting it to another letter. Digital signatures overcome this by making the signature a function of the message itself; any modification to the message would result in a different, difficult-to-predict signature.

NON-REPUDIATION: If Alice signed a message, she cannot later assert to a third party that she was not the one who signed it.

There is a subtle distinction between non-repudiation and authenticity+integrity. The best example I can think of that captures the difference is when we include a third party: suppose that Alice signs a message and sends it to Bob, and that Bob then shows the signed message to Charlie. Non-repudiation dictates that this is sufficient proof to Charlie that the signature came from Alice. This arises from the fact that Alice uses a public key that

is widely distributed. A MAC, on the other hand, would provide authenticity and integrity, but because it uses a secret that only Bob and Alice know, he cannot provide proof to a third party without revealing their shared secret (and thus potentially divulging all of their communication).

We now have all of the tools we need in order to solve our problem of online authentication. But before we explore those protocols, let us take a moment to consider concrete instances of how to implement these public key crypto primitives. There are several that are in wide use today, including RSA, ElGamal, and schemes based on elliptic curves. An undergraduate course in cryptography would cover all of these, but we will consider just one: RSA.

==========

RSA: AN EXAMPLE CONSTRUCTION OF PUBLIC KEY CRYPTO

RSA is built on the assumption that it is computationally difficult to factor a number that is the product of large primes. It makes use of modular exponentiation (numbers raised to a power mod N). Modular exponentiation has many interesting properties, but the most critical in understanding RSA is the following:

Suppose that p and q are two primes. Then the following holds:

$$a^b \pmod{p*q} = a^{(b \bmod (p-1)*(q-1))} \pmod{p*q}$$

What this is saying is that if we are working mod p*q, then the exponents work mod (p-1)*(q-1).

This function (p-1)*(q-1) is a specific example of what is known as Euler's totient function, but we will not consider it more deeply here (I encourage you to take an undergraduate course in cryptography or number theory to learn more about it and to see proofs of the above property).

==RSA ENCRYPTION AND DECRYPTION==

(1) Key generation G:

- Choose two large primes p and q at random
- Compute N = p*q
- Choose a small number e (there are a few typical values)
- Compute d such that $d*e = 1 \pmod{(p-1)*(q-1)}$
- Throw away p and q

> The public key is (e,N) -- e is the encryption key
> The secret key is (d,N)

(2) Encryption E(e, m):

- Return $c = m^e \pmod{N}$

(3) Decryption D(d, c):

- Return $c^d \pmod{N}$

First let's check that this algorithm is correct:

$$c = m^e \pmod{N}$$
$$D(d,c) = c^d \pmod{N} = (m^e)^d \pmod{N}$$
$$= m^{(e*d)} \pmod{N}$$
$$= m^{(e*d \pmod{(p-1)*(q-1)})} \pmod{N}$$

= m^1 (mod N)
= m

Why is this secure? After all, given N and e (the public key), couldn't anyone compute d? At a high level, to do so seems to require the ability to compute (p-1)*(q-1), which in turn seems to require knowing both p and q. In other words, to compute the private key (d) given the public key (e and N), one would have to first factor N, which is believed to be hard.

Digital signatures using RSA simply follows in reverse. It makes use of a publicly known, pre-image-resistant hash function H (such as SHA-256).

==DIGITAL SIGNATURES WITH RSA==

(1) Key generation proceeds as above, resulting in public key e and secret key d.

(2) Signing Sgn(d, m):

- Return s = H(m)^d (mod N)

(3) Verifying Vfy(e, m, s):

- Return "Yes" if and only if H(m) = s^e (mod N)

WHAT YOU SHOULD TAKE AWAY FROM THESE CONSTRUCTIONS

This is a good time for me to reiterate: in production code, do NOT implement RSA or any other construction yourself; use existing libraries to do so. The reason I even show the construction here (in addition to it being darn cool) is to emphasize the following points:

- Its algorithms are deterministic: in and of themselves, they involve no randomness, and thus "encrypting" on the same message twice yields the same ciphertext. As a result, the above is often referred to as "textbook RSA" and should NEVER be used for encryption in practice. Instead, use padding schemes like OAEP and PKCS with prepend a random nonce to the message before invoking RSA, thus getting us a true encryption algorithm.

- The "heavy lifting" is in the key generation phase: it is more complex than generating symmetric keys (for which almost any random key suffices), and all of our security comes from whether or not the key generation algorithm operated correctly. As a result, there are many guidelines for ensuring that p, q, e, and d are chosen well. To this end, you should only ever use libraries that generate and test RSA keys, rather than rolling your own.

==========

BACK TO AUTHENTICATION

We are at last able to solve our problem of online authentication: how can Bob know that messages purportedly from Alice really are from her, without without having to meet in person ahead of time?

Recall from earlier in this document that we developed a protocol involving a TRUSTED THIRD PARTY, Trent, to act as an intermediary between Alice and Bob, and to vouch for them to one another. The problems with this approach were that Trent was able to learn who was communicating with whom, and Trent needed to be online for Alice to initiate secure communication with Bob.

We can now consider a protocol that makes use of public key crypto's ability to allow entities to securely communicate without having to both be online at the same time.

0. Suppose that both Alice and Bob trust a third party, Trent.

1. Trent generates a public/private key pair (PK_T, SK_T) and publishes his public key as widely as possible.

2. Alice generates a public/private key pair (PK_A, SK_A), and sends PK to Trent.

3. Trent verifies that he is communicating with Alice, e.g., by visiting her or by asking her questions that only she could know.

4. If he is able to verify that PK_A came from Alice and that she knows the corresponding secret key SK_A, then Trent uses his secret key SK_T to sign the message "Alice's public key is PK_A". This message along with Trent's signature is called a CERTIFICATE.

5. Trent gives this certificate to Alice.

6. When Alice sends an encrypted message to Bob, she signs it using her secret key SK_A, and she also sends the certificate she got from Trent.

7. Bob verifies the certificate to see that Trent has indeed asserted that Alice's public key is PK_A.

8. He uses PK_A to verify Alice's signature: if it verifies, then it must have come from someone who knows SK_A, and if Bob trusts Trent, then the only person who knows SK_A is Alice. Thus the message must have come from Alice.

In the case of web browsing, Alice would be amazon.com, Bob would be a user, and Trent would be one of many so-called CERTIFICATE AUTHORITIES. These include Verisign, Comodo, Thawt, and many others.

In practice, certificates have more information; open up a web browser of your choice, go to a secure website (like bankofamerica.com), and inspect the certificate by clicking on the information in the address bar.

There are a few natural questions that this raises:

(a) How do Alice and Bob learn about Trent's public key in the first place?

> In practice, the most common way of disseminating the keys of certificate authorities is to include them directly with an operating system or a browser. When you bought your machine, you inherited trust in companies like Verisign. Mac OS X ships with over 200 public keys of various certificate authorities.

(b) Step #8 above assumes that the only person who knows Alice's secret key is Alice: what happens if Alice's key gets compromised, say, because someone stole her laptop.

> In practice, Alice would have to inform Trent that her key was compromised. Trent would then have to try to inform anyone who may have obtained a copy of Alice's certificate that it is no longer valid. We call this process CERTIFICATE REVOCATION. Certificate revocation typically comes in one of two flavors:

- Certificate revocation lists (CRLs): Certificate authorities publish lists of revoked certificates; users are expected to periodically obtain these CRLs to see if any of the certificates they encounter have been revoked (and should thus be ignored). If a certificate was revoked since the last time the user downloaded the CRL, then the user is vulnerable.

- Online Certificate Status Protocol (OCSP): To reduce users' vulnerability, the idea behind OCSP is to ask the certificate authority "Have you revoked this certificate?" before accepting it. On the one hand, this improves how up-to-date users are, but on the other hand, they have to reveal to certificate authorities all of their communication habits.

A revocation system that balances between delivering timely information, maintaining users' privacy, and keeping costs low ... simply does not exist in practice. Whether one exists is an open question. What do you think such a system would look like?

(c) What if there is no Trent whom both Alice and Bob trust?

The above protocol which assumes a few, widely trusted certificate authorities represents one broad way to achieve a PUBLIC KEY INFRASTRUCTURE (PKI); in fact, it is so common, that any system that uses a protocol like this is often referred to as a PKI.

But there are other forms of PKIs that do not require trust to be so heavily concentrated. A canonical example is a system called PGP (PRETTY GOOD PRIVACY). In practice, it works very similarly: Alice and Bob share their public keys with many people, and in return get certificates from those people attesting to the fact that that is indeed their public key. Alice sends not just one certificate, but potentially many certificates she has obtained that all assert that she is the owner of her public key. Bob can then decide on his own whether this collection of certificates convinces him that Alice truly owns that key. Perhaps one of the certificates comes from a close, trusted friend of his; or perhaps Alice has five signatures from UMD computer science professors. The point is that Bob can apply his own trust metrics.

We have at last solved our problem of authentication in an online setting, but in so doing, we have also identified several shortcomings of the web: users are required to trust a relatively small set of third parties for virtually all secure web browsing. Also, users are often out-of-date with knowing what certificates have been revoked. There are many active areas of research that seek to solve both of these shortcomings.

==========

CONCLUSION

Driven by the goal of solving the problem of online authentication, we have added two new black boxes to our arsenal: public key encryption and digital signatures. These have very similar counterparts in the symmetric key setting, yet the seemingly minor difference of using two keys instead of one allowed us to construct protocols that disseminate keys without requiring all parties to be online at all times.

There are several variants of all of these schemes that are mostly of

academic interest, but that in time may make it into more mainstream systems. For example, BLIND SIGNATURE schemes allow Alice to ask Bob to sign a message m without revealing that message to Bob. This has been shown to be useful in developing anonymous currencies. MULTI-SIGNATURES allow a set of N people to all sign the same message, yielding a signature that is of size O(1). This has been shown to be useful to compress data when sent over very slow links.

In sum, the techniques covered in this document are extremely powerful when developing systems for use on the Internet. Staying up to date with new techniques as they are discovered is important as a security expert.