**Problem 1**

a) First, look at the two FDNs you received, and take one of the email addresses where delivery failed (for example, the email might be [a@stanford.edu](mailto:a@stanford.edu)). Let’s say the person you are trying to attack is named Vern, and Vern’s email is [vern@vern.com](mailto:vern@vern.com). You can send an email to [a@stanford.edu](mailto:a@stanford.edu) and spoof the sender as [vern@vern.com](mailto:vern@vern.com). Thus, Vern will receive a FDN, which is two times the size of the original email you emailed to [a@stanford.edu](mailto:a@stanford.edu).

b) The Stanford server could create a filter that only generates one FDN for each unique set of intended recipient and source IP address. That way, [vern@vern.com](mailto:vern@vern.com) would only receive one FDN for the spoofed emails to [a@stanford.edu](mailto:a@stanford.edu) since the source IP addresses and the intended recipient is the same.

c) You could create a random IP address for each email you send and spoof the source IP address with this random address; thus, the Stanford filter would not filter out those emails, and would still generate a FDN for each email to [a@stanford.edu](mailto:a@stanford.edu) from a different source IP address.

**Problem 2**

a) N could launch a denial-of-service against R by sending “No Such Domain” messages for each query Q that R sends, and deny R access to valid websites. If R receives the “No Such Domain” message before the actual reply that DNSSEC returns, then R will not receive the IP address that the DNSSEC server sends, and it will think that the domain it requests does not exist.

b) N could send multiple queries to S for Q’s that it knows does not exist. Because signatures take a long time to create, N could overwhelm S and deny services for anyone else trying to send queries to S. N could not still launch the attack from part a – S sends a different signed message for each query that does not exist, so the only way to spoof R would be if R’s query actually didn’t exist. Although N could not do the same DoS attack, it could still overwhelm R by repeatedly sending messages since decrypting signatures may take a while.

c) S will respond, “The hash that in alphabetical order comes after 80a4cb36 is c218f96a.” R can find the hash of abc.cs161.com offline, and find that abc.cs161.com comes after 80a4cb36 and before c218f96a. Now, R knows that abc.cs161.com does not exist.

d) Using the attack from part c, the attacker would be able to determine if all of these names exist. It could send the query request; hash the name offline (using the hash function, the salt, and the number of iterations, which it knows), and then check if the hash of the name falls alphabetically between the two hashes that are given. If it does, we know that the given name is not in the domain.

e) A new salt increases the computing that an attacker must do, as the attacker must compute hashes for the salt. A s-bit salt would increase computation by a factor of 2s if we decided to pre-compute all of the hashes for the dictionary, and also increase the size of the pre-computed hashes if the attacker is storing them in a dictionary.

f) The purpose of the iteration parameter is to make it more expensive to compute a hash and add a level of distance between the actual code and its hash. An iteration parameter of size k calls the hash function k-1 times.

g) An upper bound on the iteration parameter prevents the security costs from outweighing the benefits. If the iteration parameter is too high, computing it over and over again is costly. At some point in time, it is good to set a bound on the iteration parameter so benefits are still more than cost.

**Problem 3**

a) TCP does not preserve text boundaries, and also IP does not guarantee in-order arrival (from the “root” example in lecture). Thus, something containing the 4-byte pattern may be able to get through. For example, let’s say you’re trying to detect 1234. However, the actual pattern may be displayed as 34... | |….12 and your system would not be able to detect this even though it is part of the worm.

b) You can try to reassemble the entire TCP bytestream by using the sequence numbers and ignore the text boundaries. This revised approach eliminates most cases of false negatives because now you would be able to see the 4-byte pattern reassembled. However, some packets may be discarded because TTL hop count expires; thus, faulty packets may still be able to make their way through because you cannot fully reassemble.

c) There are 2^32 possibilities for a 4-byte message (each byte has 2^8 possible bit combinations). We have a stream of 150\*2^20 (150 megabytes) bits per second. So the total number of possibilities divided by the combinations per second returns 27.31 seconds, the number of seconds for us to cause a false alarm.

d) Now, there are 2^64 possible combinations of bits for the 8-byte signature. The stream is at the rate of 2^30 \* 20 bits per second. 2^64 / (2^30 \* 20) is 858993459.2 seconds, or 14316557.65 minutes, or 238609.29 hours, or 9942.05 days, or 27.22 years.

e) One solution is through polymorphic code. A worm author could encrypt their worms with different keys (and attach a decryptor for each unique key with that individual worm). Thus, the user and the server will not be able to detect the worm based on past experiences with similar byte patterns, as the patterns will have changed. Another solution would be to use metamorphic code, as discussed in lecture.