Assignment 4 – Due Tuesday, April 16

1. In this problem you are to experiment with SHA-256 by writing some programs that gather some statistics. In addition to answering some relevant questions about the distribution of SHA-256 digests, you will gain experience using a real cryptographic library in your code. You can code this up in C, C++, Java, C#, or Python — examples in each of these languages are given at `http://www.uncg.edu/cmp/faculty/srtate/580/digest_ex.php`. Submit your programs for parts a and b through Blackboard, and include the output or results in your written solution that you turn in during class.

   (a) Write a program that computes and prints the SHA-256 hash value of your name. The output should be printed in hexadecimal.

   (b) Write a program that computes hashes of many different inputs, and counts how many 1-bits are in the output for each hash value. The output for this problem should be a histogram of the number of times each bit count occurred. I don't care how you generate the different inputs — they can be random, or you can make a big binary counter — just make sure your inputs are all different (at least with high probability). You can either write code to output a histogram from your program, or you can output the counts and use Excel or some other program that can create the histogram. For an example of what the histogram might look like, see `http://www.uncg.edu/cmp/faculty/srtate/580/dighistogram.txt`. Your program should be able to run through at least a million hash values in a reasonable amount of time — my solution processed 1,000,000 hashes in a little less than a second on UNCG's `prdile.uncg.edu` system.

   (c) Time your program, and report both the overall time and number of hashes, and then report the time as the number of hashes computed per second. How long would it take to compute 2^{128} hash values (the number needed for a birthday attack against weak collision resistance)? How long would it take to compute 2^{256} hash values (the number needed for a brute-force attack on strong collision resistance)?

   (d) If the output hash values were completely random, then each of the 256 output bits would be a one with probability ½. Therefore, the distribution of bit counts would be a binomial distribution, and so the probability of an output having 128 bits set to one would be `Binomial[256,128]/2^256`. What is this probability? (Hint: Use Mathematica — the format of the formula in the preceding sentence is a valid Mathematica formula.) What is the probability of having just 100 bits set to one?

   (e) If you run 1,000,000 trials, what is the probability that at least one trial will have exactly 100 bits set? (Hint: Calculate the probability that none of the trials will have 100 bits set — if this probability is p then the answer to this problem is $1 - p$.) Note that there is really no

way to do this problem without something that does very high precision arithmetic — we'll do some more example with Mathematica in class so that you can see how this works.

(f) If you run 1,000,000 trials, what is the probability that at least one trial will have exactly 90 bits set? What about 80 bits?

2. Since a MAC is conceptually like adding a key to a hash function, let's try taking the key away from a MAC and seeing if it makes a good hash function. In particular, consider the Data Authentication Algorithm (sometimes called CBC-MAC) using DES, which is described in Section 12.6, where the input consists of an integer number of 64-bit blocks: $D_1, \ldots, D_N$. We then "take away" the key by setting it to zero, and computing

$$\begin{aligned} O_1 &= DES(0, D_1) \\ O_2 &= DES(0, D_2 \oplus O_1) \\ &\cdots \\ O_N &= DES(0, D_N \oplus O_{N-1}) \end{aligned}$$

The hash output is O_N. Is this a one-way function (i.e., preimage resistant)? Can you find collisions efficiently (given $D_1, D_2, \ldots, D_N$ can you find a different input that gives the same hash value)? Clearly justify your answers, giving attack algorithms as appropriate.

3. The Digital Signature Algorithm (see Figure 13.4 on page 405) starts by selecting a random k value, after which $r = (g^k \bmod p) \bmod q$ becomes part of the signature. Since k is just a random value, not related to the signer's private key, is it important to protect k? In particular, what would be the consequences if an attacker could learn k in addition to the signature (r, s)?

4. Public key algorithms are really nice for key management purposes: you don't have to have a pre-shared secret with another party in order to protect confidentiality (using public-key encryption) or integrity (using digital signatures). However, public-key operations are computationally intensive. Consider a scenario in which remote sites are sending streams of small data packets to a central data monitoring facility, and you want to protect the integrity of each data packet to insure that it hasn't been tampered with (imagine that they are alarm signals, for instance). There are no pre-shared secrets between the central data monitoring system and the remote sites, but the central site does know each remote system's *public keys* for both encryption and signing. One solution would be to have the remote site digitally sign every packet that it sends, and the monitor can verify these packets using the remote site's public verification key. However, this involves a public key signing and verification operation for each packet, which is inefficient.

   Design a solution for this problem that uses public key cryptography to set up some values when a remote site initially connects to the monitoring site, and then uses fast algorithms for each packet. "Fast algorithms" could include hash functions and MACs, but no public key encryption or signature operations. Carefully describe your system, and justify that it provides good security (I'm not looking for a formal statement and proof, but I do want a clear, logical argument).

5. The "textbook RSA" signature algorithm uses RSA keys as discussed in RSA encryption, with a public verification key $K_V = (n, e)$ and private signing key $K_S = (n, d)$, and signs a short message M using the formula $RSASign(K_s, M) = M^d \bmod n$. In practice two things differ from

this description: First, to allow the message M to be longer than the modulus n, we sign a hash of M rather than M itself. Second, padding is added before applying the RSA function, much like OAEP padding is used in encryption (a common padding technique for digital signatures is called PSS). It's interesting to explore the $RSASign$ function *without* these security/practice improvements though, since it has some interesting properties.

Let's say Sigourney runs a public "signing service" — she publishes her public key K_V, and will sign any value a customer sends to her as long the customer pays $10 per signature. Alice would like to use this service to sign a value $x \in \{1, \ldots, n-1\}$, but the data Alice wants signed contains confidential information that she doesn't want to reveal to Sigourney.

(a) Since $K_v = (n, e)$ is public, Alice knows these values and can do the following: Alice picks a random value $S \in \{1, \ldots, n-1\}$ and computes $B = S^e \bmod n$. The value B is uniformly distributed over $\{1, \ldots, n-1\}$ (meaning every value has the same probability) — justify this statement!

(b) Next, Alice computes $y = x \cdot B \bmod n$. What can you say about the distribution of y (justify your answer)?

(c) Alice sends y to Sigourney, pays her $10, and gets back the result $z = RSASign(K_S, y)$. Write out the formula that describes z in terms of S, x, d, and n.

(d) Alice now has z, a signature on y created by Sigourney. How can this signature be turned into a signature on x, the original value that she wanted signed?

Note: The process described in this problem is known as "RSA signature blinding" and was the basis of the first "electronic cash" system, designed in the mid-1980's by David Chaum.