CS255 Homework 2

Rory MacQueen SUNet ID: macqueen

February 26, 2014

Problem 1

A. We can prove the contrapositive - namely that if the compression function is collision resistant, then the hash function is collision resistant. Suppose the collision function f is collision resistant, meaning it is highly unlikely that two different inputs map to the same output. We want to show that the hash function H, which is simply composed of n steps of the f function, is collision resistant. That is, we want to show that if two outputs of the hash function are equal, then the inputs to produce those two outputs must also have been equal. Now, we know then that if the final output of two different hashes are equal then the blocks of data that were fed into the final stage compression function f must have been equal (with high probability), because we assumed that f is collision resistant. But if those blocks at stage $n-1$ were equal, then also the blocks at stage $n-2$ must have been equal, since to get from $n-1$ to $n-2$ we simply fed it through the compression function f also. We can see that this reasoning will recurse backwards until we get the initial input that is fed into the first round of f functions. These initial inputs will be equal, thus we showed that the Hash function will only produce equal outputs when its inputs are equal, and hence it is collision resistant. Note that this proof holds if only if the messages are of the same length, since only then can we recurse up to the initial inputs completely. But note that if the lengths of two messages are different, it is impossible for them to have the same output, since the length itself is fed into f (which we assume to be collision resistant) at the final stage of the process, so different outputs will be produced.

B. We can show that it is not collision resistant by constructing two messages m_1 and m_2, where $m_1 \neq m_2$ but $H(m_1) = H(m_2)$. Let $m_1 = b_1||b_2$ where b_1 and b_2 are some blocks of data. Now note that since m_1 consists of two blocks, those two blocks will go through a compression function f, to produce c_1. c_1 will then go through a compression function with an all-zero block (as in question prompt) to produce the output. Now, we set $m_2 = c_1$. Since m_2 only consists of one block, it goes directly to the last step of the hash function, where it is fed into the compression function f along with the all-zero block. But since $m_2 = c_1$, this is exactly the same as what happened in the final stage of m_1's hash process. Thus the two outputs are the same, despite the fact the inputs are different. Thus we have constructed a collision, and thus H is no collision resistant.

Problem 2

A. For f_1, suppose x, y are fixed. We choose some y' and then set x' to be:

$$x' = D(y', E(y, x) \oplus y \oplus y')$$

Note that:

$f_1(x', y') = E(y', D(y', E(y, x) \oplus y \oplus y')) \oplus y'$

$= E(y, x) \oplus y \oplus y' \oplus y'$

$= E(y, x) \oplus y = f_1(x, y)$

B. For f_2, suppose x, y are fixed. We choose some x' and then set y' to be:

$$y' = x' \oplus D(x', E(x, x \oplus y))$$

We can verify that:

$f_2(x', y') = E(x', x' \oplus x' \oplus D(x', E(x, x \oplus y)))$

$f_2(x', y') = E(x', D(x', E(x, x \oplus y)))$

$f_2(x', y') = E(x, x \oplus y) = f_2(x, y)$

In both cases we have constructed x' and y' such that they collide with x and y in the functions. Hence f_1 and f_2 are not collision resistant.

Problem 3

A. Since all users (A and $B_1, \ldots, B_n$ all possess the secret key, any one of them can generate a valid MAC. So, specifically any user B_i can create a message and add a valid MAC to it to send to the other B_is. Therefore, B_1 is not assured that the packets he receives are coming from A.

B. The sets $S_1, \ldots, S_n$ need to satisfy the following property. For every pair S_i and S_j, it must be the case that $S_i \not\subseteq S_j$ and $S_j \not\subseteq S_i$. In other words, no player's set can be entirely contained within the set of another player. This ensures that no player B_i can spoof a valid MAC to another player B_j.

C. $\binom{5}{2} = 10$. This means that there are 10 unique ways to select two keys from a set of 5 keys. Thus we can have 10 recipients and give each recipient a unique set of keys that is not a subset of any other participant's set of keys. The sets to choose would be:

$\{k_1, k_2\}$, $\{k_1, k_3\}$, $\{k_1, k_4\}$, $\{k_1, k_5\}$, $\{k_2, k_3\}$, $\{k_2, k_4\}$, $\{k_2, k_5\}$, $\{k_3, k_4\}$, $\{k_3, k_5\}$, $\{k_4, k_5\}$

Problem 4

We know that if we have a 1 byte pad, that one byte ought be set to all zeros. So we construct c' as follows: The last block of c' (call it c'_n) is equal to the last block of c. However, for the second-last block of c' (call it c'_{n-1}), we take the second-last block of c and XOR the last byte of it with the byte g whose presence we are trying to detect in the message m.

Let x_1 be a block of all zeros, except the last byte, which is g.

So:

$c'_n = c_n$

$c'_{n-1} = c_{n-1} \oplus x_1$ [This is equivalent to XOR'ing g into the final byte of c_{n-1}]

We send c' to the server.

Note what will happen. The server uses CBC encryption. It will decrypt the last block of c' and XOR the result with the second last block of c'. Had we not XOR'd in g to the second-last block of c', the result would simply be the last block of m. However, we did XOR in g and so it is equivalent to performing the XOR with g now (XOR is associative).

$D(k, c'_n) = D(k, c_n) = d_n$

Now to XOR with the previous block:

$$d_n \oplus c'_{n-1} = d_n \oplus (c_{n-1} \oplus x_1) = (d_n \oplus c_{n-1}) \oplus x_1 = m \oplus x_1$$

If the final byte of m is equal to the byte of g, we will have a valid pad as the final byte (a single pad of all zeros). If the final byte of m is NOT equal to the byte of g, then we will have produced some bogus pad, and we will receive an error message saying so. Hence we, the attacker, will have learned if the final byte of m was equal to g.

Problem 5

A. This system does not provide cipher text integrity. An attacker can send m_1 to the challenger, and receive (c, c) in return. Attacker can then modify the second copy of c to be something else, and send this to the challenger to decrypt, i.e it will send (c, x) where $c \neq x$. Since the challenger only looks at the first copy of c, it will accept as valid this new message, even though parts of the cipher text have been modified. Hence, the attacker was able to successfully forge a new cipher text message, and hence this system does not provide cipher text security.

B. This system does provide authenticated encryption. The output is entirely encrypted text, and we know that (E, D) provides authenticated encryption. Since the decryption step $D_2(k, (c_1, c_2))$ checks to make sure $c_1 = c_2$, then it must be the case that those two parts are equal. So if an attacker were able to forge a new valid cipher text for this system, then we could simply take the first half of it - namely the c_1 part, and that would be a new valid cipher text for (E, D), but we know this to be impossible since (E, D) is secure. Thus, by contrapositive reasoning, this system must also be secure.

C. This system does provide authenticated encryption. Again, the output is entirely encrypted text, and $E(k, m)$ is randomized so there is no hope of breaking CPA security. If you could, then you could simply break CPA security on (E, D), which we know is impossible. Moreover, since (E, D) is a secure system, collisions are highly unlikely. Therefore, the check to make sure that $D(k, c_1) = D(k, c_2)$ is effectively equivalent to verifying that $c_1 = c_2$. As stated before, this ensures cipher text integrity. By the contrapositive again, if one were able to generate a new valid cipher text for this system E_3, then we could just take the first half of that message and have a new valid cipher text for (E, D). Clearly, this is impossible since (E, D) is secure, so (E_3, D_3) must also be secure.

D. This system does not provide CPA security. This is because the hash function H is computed on clear text and is not itself then encrypted. So an attacker can send two messages m_1 and m_2. The challenger will send back either $(E(k, m_1), H(m_2))$ or $(E(k, m_2), H(m_2))$ To determine which was sent back, the attacker can simply compute on his own $H(m_1)$ and $H(m_2)$ and compare both to the $H(m)$ he gets back. If $H(m)$ is equal to $H(m_1)$, he knows he got m_1 back; otherwise, he got m_2 back.

Problem 6

A. Party i receives from user B:

$$z_i = x_i^b$$

Since $x_i = g^{a_i}$:

$$z_i = (g^{a_i})^b = g^{a_i b}$$

Now, we want to derive g^b. We want to know what value k we need to exponentiate z_i by to get g^b:

$$(g^{a_i b})^k = (g^{a_i k})^b = g^b$$

This reduces to trying to find k such that:

$$g^{a_i k} = g$$

We know that $g \in \mathbb{Z}_p^*$ and that it has order q.This means that the set generated by exponentiating g consists of $\langle g \rangle = \{g, g^2, g^3, \ldots, g^{q-1}, 1\}$. Therefore we know that $g^q = 1$ (in fact, since it is a cycle, we know that for any positive integer t, $g^{qt} = 1$), and, since we are operating in modulo p space, $g^{q+1} = g$. Therefore, we can conclude that:

$$a_i k = qt + 1$$

where t is the number of times we had to 'cycle through' the group $\langle g \rangle$. Rearranging:

$$a_i k - qt = 1$$

We know from the problem setup that q is prime, and that $a_i < q$. Therefore, a_i and q must be relatively prime - that is, the greatest common divisor between the two must be 1. This equation can therefore be solved efficiently using the extended Euclidean algorithm. The algorithm will tell us what k and t satisfy this equation (of course, we only care about k). Using k, we can now go back and compute g^b from z_i:

$$z_i^k = (g^{a_i b})^k = (g^{a_i k})^b = g^b$$

B. $y = g^b$. Clearly user B knows y since he know g and b. We showed above how any party i can compute g^b. No eavesdropper can compute g^b because, as shown above, doing so required knowing a_i. But a_i is not sent over the network, only g^{a_i} is. Therefore, to compute a_i, an attacker would have to solve the discrete log problem, for which there is no known efficient solution.

C. We know that algorithm $\mathcal{A}$ does $(g, g^x, g^y) \rightarrow g^{x/y}$ and we will define an algorithm $\mathcal{B}$ that produces g^{xy}:

$$(g, g^x, g^y) \xrightarrow{\mathcal{A}} g^{x/y}$$

$$(g, g^y, g^x) \xrightarrow{\mathcal{A}} g^{y/x}$$

We can feed these results again through $\mathcal{A}$ to get:

$$(g, g^x, g^{y/x}) \xrightarrow{\mathcal{A}} g^{\frac{x}{y/x}} = g^{x^2/y}$$

$$(g, g^{x/y}, g^y) \xrightarrow{\mathcal{A}} g^{\frac{x/y}{y}} = g^{x/y^2}$$

And now, for the final move:

$$(g, g^{x^2/y}, g^{x/y^2}) \xrightarrow{\mathcal{A}} g^{\frac{x^2/y}{x/y^2}} = g^{xy}$$

Hence, if $\mathcal{A}$ is an efficient algorithm, then $\mathcal{B}$ can simply follow these steps and we will have broken the Computational Diffie-Hellman assumption.