CS255: Introduction to Cryptography **Winter 2012**

Assignment #2: Solutions

Answer 1. **(a)** `(2 points)` Suppose we can find two message/hash pairs $\langle M_1, h(M_1)\rangle$ and $\langle M_2, h(M_2)\rangle$ such that $M_1 \neq M_2$ and $h(M_1) = h(M_2)$. Then, there exists two distinct Merkle hash trees T_1 and T_2 whose outputs are identical.

We can find a collision for the compression function using a top-down side-by-side comparison of our two trees, looking across trees for a case where the outputs of f are the same, but the inputs differ. Starting at the root, suppose that either the first block or `msg-len` (thus height) of our trees differ, in this case, clearly we have found a collision in f, so we can stop our search.

We can continue down examining the i^{th} node of T_1 and T_2 and compare the inputs. If they differ, then we have found a collision for the compression function, and the procedure terminates. If not, we proceed to level $i+1$ and repeat the same procedure. Note that since $M_1 \neq M_2$, we cannot go through the entire tree without finding a collision. It follows that one can find a collision in the compression function.

(b) `(1 point)` To construct a collision without the `msg-len` block, proceed as follows: Choose any M_1 and construct the Merkle Tree. Without loss of generality, we assume that $|M_1| = 2^{\ell}$, and therefore, the depth of the Merkle Tree is $\ell + 1$. Next, consider the outputs of the compression function $f(\cdot)$ applied to the leaves to get $2^{\ell-1}$ first-level nodes. Let the outputs be $f_1, \ldots, f_{2^{\ell-1}}$. It is easy to see that by setting $M_2 = f_1 \| f_2 \| \cdots \| f_{2^{\ell-1}}$, the Merkle Tree remains the same (except it has depth ℓ, rather than $\ell+1$). Thus, (M_1, M_2) is a collision. This attack does not work if `msg-len` block is used because, as seen above, the two messages are of different lengths.

Note: It is important to note that unlike in the definition of semantic security, there are no restrictions on the *lengths* of the two colliding messages for breaking collision resistance of a hash function.

■

Answer 2. `(1 point each)` To construct a collision for the first hash construction with $f_1(x,y) = E(y,x) \oplus y$, choose any y and non-zero x so that $y \neq (x \oplus y)$.

Let:

$$x_1 = D(y, x \oplus y), y_1 = y,$$
$$x_2 = D(x \oplus y, y), y_2 = x \oplus y.$$

So then:

$$\begin{aligned} f_1(x_1,y_1) &= E(y,x_1) \oplus y = E(y, D(y, x \oplus y)) \oplus y = (x \oplus y) \oplus y = x. \\ f_1(x_2,y_2) &= E(y_2,x_2) \oplus y_2 = E(x \oplus y, D(x \oplus y, y) \oplus (x \oplus y) = y \oplus (x \oplus y) = x \\ &= f_1(x_1,y_1). \end{aligned}$$

For the second compression function $f_2(x,y) = E(x,x) \oplus y \oplus x$, choose any $x_1 \neq x_2$, any y_1, and let $y_2 = E(x_1,x_1) \oplus E(x_2,x_2) \oplus y_1 \oplus x_1$,

$$\begin{aligned}\implies f_2(x_2,y_2) &= E(x_2,x_2)\oplus y_2 = E(x_2,x_2)\oplus \big(E(x_1,x_1)\oplus E(x_2,x_2)\oplus y_1\oplus x_1\big)\\ &= E(x_1,x_1)\oplus y_1\oplus x_1\\ &= f_2(x_1,y_1).\end{aligned}$$

■

Answer 3. `(2 points)` Consider a two block message $M_1 = x\|y$. Then, $\text{rawCBC}_E(k,M_1) = E(k,E(k,x)\oplus y)$. Next, take $M_2 = x'\|y'$. Here x' is arbitrary and y' will be constructed such that (M_1,M_2) is a collision. Set $y' = E(k,x)\oplus E(k,x')\oplus y$. Note that:

$$\begin{aligned}\text{rawCBC}_E(k,M_2) &= E(k,E(k,x')\oplus y') = E\left(k,E(k,x')\oplus(E(k,x)\oplus E(k,x')\oplus y)\right)\\ &= E(k,E(k,x)\oplus y)\\ &= \text{rawCBC}_E(k,M_1).\end{aligned}$$

Thus, rawCBC_E is not a collision resistant hash function.

■

Answer 4. (a) `(1 point)` Since A and all $B_i \in B$ share the secret key k, any B_i can send to the parties $B-\{B_i\}$ a message M appended with the MAC under k of M. To the recipients this looks exactly like a message sent by A, and hence, the recipient is not sure the message came from A.

(b) `(2 points)` B_i can successfully fool B_j ($i\neq j$) iff B_i has every key that B_j has. This is easy to see since B_j verifies a message by verifying the MACs corresponding to the keys he has. Thus, for the scheme to work, each pair B_i, B_j ($i\neq j$) must have at least one key not shared between them. Then, no B_i can fool a B_j because B_i will lack one of the keys that B_j uses to verify the message.

Note that we use the assumption of non-collusion to ensure that if some proper subset of parties $B^*\subset B$ have between them every key, they cannot work together to fool the parties in $B-B^*$.

(c) `(1 point)` Let the keys be k_1,k_2,k_3,k_4,k_5. We note that $10=\binom{5}{2}$. Hence each S_i need only contain two keys. The subsets are:

S_1:	$\{k_1,k_2\}$	S_2:	$\{k_1,k_3\}$	S_3:	$\{k_1,k_4\}$	S_4:	$\{k_1,k_5\}$
S_5:	$\{k_2,k_3\}$	S_6:	$\{k_2,k_4\}$	S_7:	$\{k_2,k_5\}$	S_8:	$\{k_3,k_4\}$
S_9:	$\{k_3,k_5\}$	S_{10}:	$\{k_4,k_5\}$				

Note that for any two S_i,S_j ($i\neq j$), S_i and S_j differ in at least one element.

■

Answer 5. `(3 points)` Recollect how a valid padding works. To pad d bytes at the end of the message, add d bytes each with the integer value `d-1`. After truncating the padding block, (for the sake of notational convenience) let the last block be denoted by c_1 and the second last block be denoted by c_2. By changing c_2 to $c_2' = c_2\oplus x$, upon decryption, the last message block $m_1' = m_1\oplus x$.

Now, given a target byte g, to check if the last byte of m_1 is g, modify c_2 as follows: $c'_2 = c_2 \oplus \texttt{000}\ldots\text{g}$. Thus, $m'_1 = m_1 \oplus \texttt{000}\ldots\text{g}$. If the last byte of m_1 were g, then upon decryption, m'_1 ends in a $\texttt{0}$ byte. Note that this is a **valid padding**.

However, if m_1 ends in some other byte $\text{g}' \neq \text{g}$, then the last byte of $m'_1 = \text{g} \oplus \text{g}'$. This is a valid valid padding iff the last $\text{g} \oplus \text{g}' + 1$ blocks each decrypt to $\text{g} \oplus \text{g}'$. When $\text{g} \oplus \text{g}' = 1$, this occurs with probability $2^{-2\cdot\text{byte size}} = 2^{-16}$. When $\text{g} \oplus \text{g}' = 2$, this occurs with a probability 2^{-24}. Thus, over all possible padding blocks $(\texttt{1}, \ldots, \texttt{15})$, the probability that $\text{g} \oplus \text{g}'$ leads to a valid padding is $< \sum_{i=2}^{\infty} 2^{-8i} \approx (\frac{1}{255})(\frac{1}{256}) \approx 0.0015\%$ which is a very low probability of error.

Therefore, by modifying c_2 as described above, we can check depending on the server response to padding, whether the last byte of the message is some target byte g or not. The probability of error can be reduced even further by repeating this experiment with different XOR values.

Technical note: The above analysis is only true for messages that are uniformly distributed. To avoid this assumption, there is a clever way to fix the attack. Instead of modifying $c'_2 = c_2 \oplus \texttt{000}\ldots\text{g}$, you instead pick random bytes $\texttt{r}_1, \ldots, \texttt{r}_{15}$ and set $c'_2 = c_2 \oplus \texttt{r}_1\texttt{r}_2\ldots\texttt{r}_{15}\text{g}$. Now, the probability analysis is correct.

■

Answer 6. **(a)** `(1 point)` For a given $y = g^x$, with x unknown, we know that y is a Quadratic Residue if and only if x is even. Further, by Euler's criteria, we know that y is a QR if and only if $y^{(p-1)/2} = 1 \pmod p$. So, if $y^{(p-1)/2} = 1 \pmod p$ then we conclude that x is even, and otherwise that x is odd.

(b) `(1 point)` Assuming that x is even, we know that $g^{x/2}$ is a QR if and only if $x/2$ is even. As before, we also know that $g^{x/2}$ is a QR if and only if:

$$(g^{x/2})^{(p-1)/2} = y^{(p-1)/4} = 1 \pmod p.$$

This gives us the 2nd least significant bit of x.

(c) `(2 points)` Let $b_n, b_{n-1}, \ldots b_1 \in [0,1]$ represent the bits of x, and $y = g^x$. Then the following algorithm will recover the bits of x:

```
$i = 1$
while $y \neq 1$
   if $y^{(p-1)/2^i} = 1 \pmod p$
      $b_i = 0$
   else
      $b_i = 1$
   $y = y / g^{b_i 2^{i-1}}$
   $i = i + 1$
```

Note that simply taking the square root and setting $y \leftarrow y^{1/2}$ and using the method of part (a) does not work. Indeed if it did, we could compute the discrete log of y for any prime. In fact, the algorithm for taking square roots returns both $\pm y^{1/2}$, which in $\mathbb{Z}_p$ cannot be distinguished as "positive" and "negative" square roots without prior knowledge of the discrete log (Recall $-r = p - r \pmod p$). Since only the "positive" root will give the correct next bit, the method fails.

(d) `(1 point)` This algorithm does not work for a random prime p because at each step i, we are relying on $(p-1)/2^i$ being an integral value. Although for a random odd prime, the argument in part (a) still holds.

■

Answer 7. **(a)** `(2 points)` We know that $a_i \in [1, q-1]$. Therefore, there exists a c_i such that $a_i \cdot c_i = 1 \pmod q$ (in other words, $c_i = a_i^{-1}$). Now player computes c_i given a_i and q using the Extended Euclid's Algorithm. Then player i can reconstruct y as follows:

$$\begin{aligned} z_i^{c_i} &= x_i^{bc_i} \\ &= (g^{a_i})^{bc_i} = g^{a_i b c_i} \\ &= g^{a_i c_i b} = (g^{a_i c_i})^b \\ &= (g^{1+rq})^b = g^b \cdot (g^{rq})^b && \text{because } a_i c_i = 1 \pmod q \text{ and } g^q = 1 \\ &= g^b \cdot (1)^b = g^b = y. \end{aligned}$$

(b) It is easy to see how every party i can compute the secret y efficiently. An adversary cannot determine y because that requires him to compute g^b given only g^a and g^{ab} which turns out to be as hard as the Computational Diffie-Hellman Problem as is proved in part (c).

(c) `(This and the previous part carries 3 points at my discretion.)`

Preface: Although the basic idea in the hint is correct, as mentioned in class and the discussion section, a correct reduction proof requires you to *simulate* the *entire view* of the adversary. I understand this is a little technical but the idea was to see if you could work out setting up the system given just one Diffie-Hellman instance. Almost no one got it in class (and we're grading on a curve), so the points don't matter, but if you want to try it out, please try working it out before seeing the solution. You need to simulate the interactions between all players that the adversary can see. The whole idea behind the fact that you *can simulate* all these interactions given just one Diffie-Hellman instance actually shows that if Diffie-Hellman is hard, then basically all the things seen by the adversary really give him no extra information.

Consider a computational Diffie-Hellman problem instance given to $\mathscr{A}$. Given a 3-tuple $(g, X, Y) = (g, g^x, g^y)$ for unknowns x and y $\mathscr{A}$ need to compute g^{xy}. $\mathscr{A}$ is required to setup the conference key protocol that $\mathscr{B}$ claims to break. Choose a player i at random. Without loss of generality, we assume player 1 is chosen. $\mathscr{A}$ needs to simulate the communications x_i from parties A_1 through A_n and z_i, the replies from B.

First publish h, the public element $\in \mathbb{Z}_p^*$ to be X from the Diffie-Hellman tuple. Therefore $h = g^x$. For player 1, set $x_1 = g$. For all other players, choose random $r_i \overset{R}{\leftarrow} \mathbb{Z}_p^*$ and set $x_i = g^{r_i}$. This simulates the first round of the protocol. One can verify that given h there are random $a_1, \ldots, a_n$ that players might have chosen such that $h^{a_i} = x_i$. Therefore this is indistinguishable from a legitimate instantiation of the protocol (as far as $\mathscr{B}$ is concerned).

For the second round, set $z_1 = Y$ from the Diffie-Hellman tuple. Set z_i for all other i to be Y^{r_i}. It is easy to check that these correspond to a legitimate instantiation of the protocol where party B chooses $b = y$ to be his secret. This is true if $z_i = x_i^y$. By construction, $x_1 = h^{1/x}$ therefore $z_1 = h^{(1/x)\cdot y} = g^y = Y$ as required. For all other i, by construction, since $x_i = g^{r_i}$, we get $z_i = x_i^b$.

Now, adversary $\mathscr{B}$ breaks the protocol and outputs h^b, the conference key. By construction $h^b = h^y = (g^x)^y = g^{xy}$, which allows $\mathscr{A}$ to break the computational Diffie-Hellman problem.

■