CS255: Cryptography and Computer Security **Winter 2009**

Assignment #3: Solutions

Problem 1

(a) We have $d_{eve} \equiv e_{eve}^{-1} \pmod{\phi(N)}$, so that $\phi(N)$ divides $d_{eve} \cdot e_{eve} - 1$.

(b) Pick a random nonzero g in $\mathbb{Z}_N$. In the unlikely event that $g \equiv 0 \pmod{p}$, then $\gcd(p, N) = p$, and we have factored N, so assume that this didn't happen. Let $\tau_p(g), \tau_q(g)$ (which we don't yet know) be the largest powers of 2 that divide $\mathrm{ord}_p(g)$ and $\mathrm{ord}_q(g)$, respectively. Without loss of generality, let $\tau_p(g) \leq \tau_q(g)$.

Note that if g is not a quadratic residue mod p, which happens with probability $\frac{1}{2}$, then $\tau_p(g) = \tau(p-1)$, and otherwise $\tau_p(g) < \tau(p-1)$. Since $\tau_p(g)$ and $\tau_q(g)$ are independent, no matter what $\tau_q(g)$ is, the probability that $\tau_p(g) = \tau_q(g)$ is at most $\frac{1}{2}$.

Now compute $x := g^{K/\tau(K)}$, then repeatedly square x. We know that $x^{\tau_q(g)} \equiv 1 \pmod{N}$, so consider the previous term $y := x^{\tau_q(g)/2}$, if there is one. If $\tau_p(g) < \tau_q(g)$, which happens with probability $\frac{1}{2}$, then $y \equiv 1 \pmod{p}$, but $y \not\equiv 1 \pmod{q}$. Note that in this case we are guaranteed to have a y, because $\tau_q(g) > \tau_p(g) \geq 1$. Then $\gcd(y-1, N) = p$, and we have factored N.

Since the above takes roughly cubic time in $\log N$, and works with probability $\frac{1}{2}$ each time, we will factor N in cubic expected time.

Problem 2

(a) First, let us define $t := \lceil |X|/B \rceil$ as the hint suggests. We start by building a table T in the following way: First, choose a random element k, and add k to T. Repeatedly apply f to k until one of the two following things happen: if we reach k again before we have applied f to k t times, then delete k from T. If we have not reached k again after applying f to k t times (i.e. we are in a cycle of length $\leq t+1$) then add the element we get after applying f to k t times to the table T. Suppose we call this new element j. We then apply f to j either until we reach k again (in which case we stop) or until we have applied f t times, at which point we add the resulting element into the table and continue onward in the same fashion as we have been doing. When we do reach k again, which must happen eventually due to the fact that every element in the group must be a cycle, we choose some element that we have not come across and repeat the process above as though the new element were k. We can repeat this process until we have gone through all of the elements. Of course, this algorithm takes quite some time, but since we have no constraints at this time, we are free to use a sluggish algorithm (and this is why we also don't bother to mention implementation here–any implementation that works is fine).

Now we explain how to invert an input $y \in X$ in time $O(t)$. Our procedure is relatively simple. First, if y is in our table, then we are done, so we clearly do not have to worry about these cases. So suppose that y is not in the table. In this case, we apply f to y repeatedly until we reach an element in our table or until we reach y again, always keeping track of the previous element. Note that if y is in a cycle that does not contain a table entry, then the cycle length must be less than t, so in this case we will find x in time $\leq t+1$. Thus, we may

assume that y is in a cycle that contains a table entry. So, we may assume that after applying f enough times to y, we will reach some element z in our table. Suppose we now consider the element in the table immediately preceding z, which we will call a. Clearly a must be in the same cycle as both z and y and x since we have included the first element of every cycle with $> t+1$ elements in our table, as well as one additional element. Thus, we can just apply f to a until we reach y, at which point we will know both y and x, which will give us x. If this is the case, we have certainly applied f less than or equal to $t+1$ times, which concludes our algorithm.

Since all branches of our algorithm run in $t+1$ calls to f, it should be clear that our algorithm runs in time $O(t)$. Additionally, it should also be clear that our table has maximum size $\leq 2B$, which is also permissible by the problem (if one wishes to be strict about the B size requirement, it is possible to use $2t$ instead of t when making the table, and our algorithm will run in $2t+1$ time but our table will have exactly B entries). The correctness of our algorithm follows simply from the explanation.

Problem 3

(a) We will first prove that g and h generate the same subgroup G of order q. Briefly, let γ be a generator of $\mathbb{Z}_p^*$, and let $g = \gamma^a$ and $h = \gamma^b$. Let $c = \gcd(a, b)$.

Now, by the extended gcd algorithm, $c = ax + by$ for some x, y, so that $\eta := \gamma^c$ is in the subgroup G generated by g, h. But by the definition of gcd, a/c and b/c are integers, so that $g = \eta^{a/c}$ and $h = \eta^{b/c}$ are generated by η. Therefore every element of G is generated by η. But η has order q, because g and h do, and so $|G| = q$. Furthermore, g and h both generate G.

Whew. On to the meat of the proof. Note that any valid commitment b must be in G, because it's generated by g and h, and each $x \in [0, q-1]$ gives a different value of g^x. Since h also generates G, there is exactly one $r \in [0, q-1]$ such that $h^r = b/g^x$. Since r was chosen uniformly, and each x can give $b = g^x h^r$ in exactly one way, b gives no information about x.

This proof is very similar to the proof of the one-time pad: since h^r is uniformly random in G, multiplying it hides g^x perfectly.

(b) Suppose that given g and h, Alice can construct a commitment b that she can open as x and also as x'. That is, she can produce $x \neq x', r, r'$ such that

$$g^x h^r = b = g^{x'} h^{r'}$$

Dividing through, $g^{x-x'} = h^{r'-r}$. Since $x \neq x'$, $g^{x-x'} \neq 1$ and so $r \neq r'$. Since $\mathbb{Z}_q$ is a field, we can compute $\frac{1}{r'-r} \pmod q$. Raising both sides of the above equation to this power, we have

$$g^{\frac{x-x'}{r'-r}} = h$$

Problem 4

(a) The user computes

$$b' := \prod_{j \in S \setminus \{e_i\}} e_j$$

so that $b'/b = 1/e_i$; then

$$\left(R^{1/b}\right)^{b'} = R^{1/e_i} = key_i$$

(b) Suppose we are given a number X and wish to find $X^{1/d_2} \pmod N$. By running $\mathcal{A}(X^{d_1}, X)$, we will obtain the value of $Y := X^{d_1/d_2}$. But since d_1, d_2 are relatively prime, we can find integers a, b such that $ad_1 + bd_2 = 1$. Then

$$Y^a X^b \quad = \quad X^{ad_1/d_2} X^{bd_2/d_2} \quad = \quad X^{1/d_2}$$

as desired.

(c) The contrapositive of this is: if it is easy for a user to obtain key_i for some $i \notin S$, then it is easy to take e_i-th roots mod N.[1] Since e_i is relatively prime to e_j for all $j \neq i$, we know that $\gcd(e_i, b) = 1$; therefore this user is performing an algorithm $\mathcal{A}$ that we can use for part (b).

Problem 5

Let us first introduce some notation. In particular, let u_0 be the root.

(a) Let bit b_j be the jth bit of i from the left, with b_n being the *least* significant bit and b_1 is the *most* significant bit. For each j, we will store

(a) pk_j
(b) pk'_j
(c) sk'_j if $b_j = 0$
(d) the signature $s_{j-1} := \text{sign}(sk^\dagger_{j-1}, pk_j || pk'_j)$ where $sk^\dagger_j = sk_j$ if $b_j = 0$ and sk'_j if $b_j = 1$.

We also store sk_n if $b_j = 0$, and we store sk'_n regardless of b_j.

As setup with private key sk_0:

(a) Choose random secret keys sk_j, sk'_j for $j \in [1, n]$ and their corresponding public keys.
(b) For each j, compute $s_j = \text{sign}(sk_{j-1}, pk_j || pk'_j)$.
(c) Store the items listed in the previous section.

To sign a message m:

(a) Compute the sub-signature $s_n := \text{sign}(sk^\dagger_n, m)$.
(b) Output the grand signature

$$(i, (s_0, pk_1, pk'_1), (s_1, pk_2, pk'_2), \ldots, (s_{n-1}, pk_n, pk'_n), s_n)$$

To update the internal state after signing the message:

(a) Increment i. b_j will change from 0 to 1, and the bits after it will change from 1 to 0.
(b) Choose new random secret keys sk_k, sk'_k for $k \in [j+1, n]$ and their corresponding public keys.

[1] This is not exactly the contrapositive. I've strengthened it slightly because taking "first roots" is easy, yet $\gcd(1, \phi(N)) = 1$).

(c) Compute $s_j, s_{j+1}, \ldots, s_n$.

(d) Forget sk_j, as required.

(e) Store the items listed in the first section.

To verify a message m:

(a) Verify for each $i \in [0, n-1]$ that s_i is a valid signature of $pk_{i+1}||pk'_{i+1}$ for public key pk_i.

(b) Verify that s_n is a valid signature of m.

(b) A given secret key $sk_j^\dagger$ where $j < n$ is used only during the initialization or state update step. If it's a "left" secret key sk_j, it's used once during this step and then forgotten. If it's a "right" secret key sk'_j, it's used only when b_j becomes 1, only once, and then forgotten.

Similarly, sk_n and sk'_n are used only to sign one message m, and are then forgotten.

(c) Grand signatures contain an n-bit integer i, plus n Lamport signatures and $2n$ Lamport public keys (which have $2n$ one-way function evaluations each). If the hash function maps messages to $\{0,1\}^a$, and the one-way function maps $\{0,1\}^b$ to $\{0,1\}^c$, this takes up $n + n \cdot a \cdot (b + 4c)$ bits.