Ungraded Homework Solutions

CSC 152 – Cryptography

Please notify me of any errors you find. If you need help, ask.

1) *Write one of the following reductions: from PreimageFinder(H, y) to 2ndPreimageFinder(H, x), or from 2ndPreimageFinder(H, x) to PreimageFinder(H, y). What security implication does your reduction establish?*

Showing Find2ndPreimage(H, x) reduces to FindPreimage(y) is straightforward.

```
Find2ndPreimage(H,x)
    do
        x' = FindPreimage(H,H(x))
    while (x' == x)
    return x'
```

I used a loop in case FindPreimage($H, H(x)$) returns x. This reduction will fail if FindPreimage($H, H(x)$) always returns x which could happen if FindPreimage($H, H(x)$) is deterministic and finds x as its first answer, or if x is the only preimage of y (unlikely).

This establishes that the existence of an efficient FindPreimage implies the existence of an efficient Find2ndPreimage. It's contrapositive tells us if there is no efficient Find2ndPreimage then there is no efficient FindPreimage, or in other words second-preimage resistance implies preimage resistance.

The other direction would look like this.

```
FindPreimage(H,y)
    ...
    x' = Find2ndPreimage(x)
    ...
```

The problem is that FindPreimage is given an element of the range and Find2ndPreimage requires an element of the domain, so we'd have to find an x where $H(x) = y$ in order to ask Find2ndPreimage to do its work.

2) *Recall that you can compute a value that is equivalent to* $x \bmod (2^a - b)$ *as* $(x \operatorname{div} 2^a) \cdot b + (x \bmod 2^a)$. *Use this fact to reduce (base-10)* $123456789 \bmod (2^{12} - 2)$ *to a smaller, equivalent number. If after doing this reduction once, the result is more than 12 bits, do it a second time to reduce it further.*

There are a couple of ways to do this problem.

The mathematical way goes like this. If you divide $123456789/2^{12}$ you get 30140 and remainder 3349. This means $123456789 = 30140 \cdot 2^{12} + 3349$. If you reduce the 2^{12} term mod $2^{12} - 2$ you get $123456789 = 30140 \cdot 2 + 3349 = 63629$. This means 123456789 and 63629 are equivalent mod $2^{12} - 2$. Since this result is not less than 2^{12} (ie, needs more than 12 bits) it needs further reduction. If you divide $63629/2^{12}$ you get 15 and remainder 2189. This means $63629 = 15 \cdot 2^{12} + 2189$. If you reduce the 2^{12} term mod $2^{12} - 2$ you get $63629 = 15 \cdot 2 + 2189 = 2219$. Since this result is less than 2^{12} it needs no further reduction.

If you wanted to write this more concisely, you might format it like this.

$123456789/2^{12} = 30140$, and $123456789 \bmod 2^{12} = 3349$
$123456789 = 2^{12} \times 30140 + 3349 = 2 \times 30140 + 3349 = 63629$

$63629/2^{12} = 15$, and $63629 \bmod 2^{12} = 2189$
$63629 = 2^{12} \times 15 + 2189 = 2 \times 15 + 2189 = 2219$

Optimized on a computer, it goes like this. 123456789 in binary is 111010110111100110100010101. If we split this into the low 12 bits 110100010101 and the remaining high bits 111010110111100, we can combine

them as $(2 \times 111010110111100) + 110100010101 = 1111100010001101$. Since this is more than 12 bits long, we know that it needs further reduction. If we split 1111100010001101 into the low 12 bits 100010001101 and the remaining high bits 1111, we can combine them as $(2 \times 1111) + 100010001101 = 100010101011$. Since this is not more than 12 bits long, we know that it needs no further reduction.

3) Below is Horner's method of polynomial evaluation with four different variations. For each variation write the equivalent polynomial. Write "..." to indicate "the pattern continues until", and use x_i instead of $x[i]$. *Hint: The first one is* $x_0k^n + x_1k^{n-1} + \cdots + x_{n-1}k$.

```
                        A      B           C                    res
res = A
                        0  res + x[i]  res * k     ----------------------------------------
for (i=0; i<n; i++)
                        0  res * k     res + x[i]  ----------------------------------------
    res = B
                        1  res + x[i]  res * k     ----------------------------------------
    res = C
                        1  res * k     res + x[i]  ----------------------------------------
```

Using the same technique shown in class, we slowly expand the polynomial being formed by the loop until we see the pattern. So, for the second one, $(0k + x_0) = x_0$ is the value of *res* after one iteration, $x_0k + x_1$ after two, $(x_0k + x_1)k + x_2 = x_0k^2 + x_1k + x_2$ after three, etc. This is enough to see the pattern and after n iterations *res* will contain the value of $x_0k^{n-1} + x_1k^{n-2} + \cdots + x_{n-2}k^1 + x_{n-1}$. Similar analysis shows that the third polynomial is $k^n + x_0k^n + x_1k^{n-1} + x_2k^{n-2} + \cdots + x_{n-1}k$ and the fourth polynomial is $k^n + x_0k^{n-1} + x_1k^{n-2} + x_2k^{n-3} + \cdots + x_{n-1}$.

4) *Recall that H is ε-almost-universal if the probability $h(a) = h(b)$ is no more than ε when $a \neq b$ and $h \in H$ is chosen randomly. The following H is a family of functions all with domain $\mathbb{Z}_6$ and co-domain $\mathbb{Z}_4$. For what value of ε is H ε-almost-universal? Show your work. H is defined as follows:*

	h1	h2	h3	h4	h5
0	2	3	0	1	3
1	3	2	1	0	0
2	0	1	3	2	1
3	0	0	2	2	3
4	2	1	1	3	2
5	0	3	3	2	0

If the adversary chooses 2 and 5, then when h is chosen randomly, there is a $3/5$ chance that $h(2) = h(5)$ because h_1, h_3, h_4 all cause a collision. No other pair of inputs has a higher probability of collision when h is chosen randomly, so the collection of functions is ε-almost-universal for $\varepsilon = 3/5$. On an exam with a small domain you should list all possible pairs of domain elements, give each probability, and identify the maximum.

For easy grading on a test, you might format it like this.

```
(0,1)  0/5
(0,2)  0/5   (1,2)  0/5
(0,3)  1/5   (1,3)  0/5   (2,3)  2/5
(0,4)  1/5   (1,4)  1/5   (2,4)  1/5   (3,4)  0/5
(0,5)  1/5   (1,5)  1/5   (2,5)  3/5   (3,5)  2/5   (4,5)  0/5

(2,5) has 3/5 chance of collision, and no other pair has higher, so H is (3/5)-almost-universal.
```

5) *Let's say you are using a polynomial hash function $k^{n+1} + x_0k^n + x_1k^{n-1} + \ldots + x_{n-1}k \bmod p$ to hash the*

three-byte data `0x 26 14 04`, and let's say that $p = 257$, k is randomly chosen to be `0x55`, and that the data is broken into 8-bit chunks before hashing. What is the resulting value?

Using Python as my calculator, the result I get is 84.

```
k=0x55
p=257
y = (k**4 + 0x26 * k**3 + 0x14 * k**2 + 0x04 * k**1) % p
print(y)
```

6) *Can you find another data string (of any length) that yields the same output value?*

Perhaps the fastest way to do this is to assume a one-byte data can cause an output of 84 and simply use a for-loop to find it. The following outputs 46.

```
k=0x55
p=257
for i in range(256):
    if ((k**2 + i * k) % p == 84):
        print(i)
```

There are algebraic ways to find an answer, but for such a small problem brute-force is easiest.

7) *We saw that an authentication tag can be generated by combining a universal hash like the one above with a random function: TagGen(x, n) = h(x) op f(n). (The operation used depends on the specifics of h and f.) Because it's readily available, let's say we are using the AES S-box for f, the hash function listed above with k=0x55 for h, and addition mod p=257 for the TagGen operation. What authentication tag is generated for the three-byte data `0x 26 14 04` when the nonce used is 0x10?)*

We know that the hash value is 84. The S-box produces 0xCA for input 0x10. Since 0xCA is 202, the tag is 84 + 202 = 286 mod 257 = 29.

CE 5.5) Let's work out the computation on m'.

$H'_1 = E(H_0 \oplus m'_1)$. Since H_0 is zero and substituting for the definition of m'_1 we get $H'_1 = E(m_2 \oplus H_1)$. This is the same input as for the definition of H_2, so $H'_1 = H_2$.

$H'_2 = E(H'_1 \oplus m'_2)$. Substituting for $H'_1 = H_2$ and the definition of m'_2 we get $H'_2 = E(m_2 \oplus H_1)$. This is the same input as for the definition of H_2, so $H'_2 = H_2$.

CE 6.2) Since $M(a||c) = M(b||c)$, this means the inputs to the final block cipher call are the same in both cases (a block cipher is invertible, so if the outputs are the same so are the inputs). In other words $M(a) \oplus c = M(b) \oplus c$. Xor both sides by c and we see $M(a) = M(b)$. This means $M(a) \oplus d = M(b) \oplus d$ which are the final inputs to the block cipher when computing $M(a||d)$ and $M(b||d)$. Thus $M(a||d) = M(b||d)$.