Assignment 4: Relaxed Secrecy and Privacy
15-316 Software Foundations of Security and Privacy

Due: **11:59pm**, Thursday 4/12/18
Total Points: 50

1. **Safe or unsafe relaxation (15 points).**

   For the declassification operators defined below, determine whether an attacker can always use it to figure out the value of an n-bit string `pin` in $\mathbf{poly}(n)$ time. If so, describe how. If not, prove why doing so is impossible using a similar argument to the one for `match` covered in lecture. All declassification operators below have the same typing rule as `match`.

   - (5 points) Greater than: $\mathtt{gt}(\mathtt{guess}, \mathtt{pin})$ evaluates `true` if and only if `guess`, interpreted as an integer, is greater than the n-bit string `pin`, interpreted as an integer.
   - (5 points) Error-correcting match: $\mathtt{ecm}(\mathtt{guess}, \mathtt{pin})$ evaluates to `true` if and only if `guess` is an n-bit string that differs from the n-bit string `pin` by at most 1 bit.
   - (5 points) Prefix: $\mathtt{pref}(\mathtt{guess}, \mathtt{pin})$ evaluates to `true` if and only if `guess` is a prefix of the n-bit string `pin`.

2. **Primitive badness (15 points).**

   RSA is a public key cryptosystem that performs encryption by taking powers modulo N of an exponent e, and decryption by taking powers modulo N of an exponent d. The details of how N, e and d are chosen are not important for this problem, but the pair (e, N) is the *public key* and d is the secret *private key*. To encrypt a plaintext message M, one computes the ciphertext $C = \mathrm{mod}(M^e, N)$. Likewise to perform decryption given C to recover M, one computes $M = \mathrm{mod}(C^d, N)$. Thus modular exponentiation lies at the core of the algorithm, so is the essential primitive needed to implement RSA.

   The program below implements modular exponentiation using the square-and-multiply method[1]. Given ciphertext C, the approach iterates over each bit j of the L-bit private decryption key d, squaring (mod N) the ciphertext at each step. If the current bit $d[j]$ is 1, then the current result is multiplied by the original ciphertext (again mod N). The modulo operation here is implemented in a very simple manner by repeated subtraction.

   ```
   x := C;
   for(j in 0 to L-1) {
     x := x * x;
     while (N <= x) { x := x - N; }
     if (d[j] = 1) {
       x := x * C;
       while (N <= x) { x := x - N; }
     }
   }
   ```

   Assuming that all variables except d are public and j is initialized to 0, this modular exponentiation program contains a timing side channel. Explain what it is. Then, given the following timings for each initial value of C below where $L = 4$ and $N = 16$, recover the value of d that led to these observations. You should assume that each arithmetic operation, comparison, and assignment takes one unit of time, and that the `for` loop does not take a unit of time to increment j.

[1] You may notice that this code only works when N is relatively prime to C. This is a reasonable assumption for reasons beyond the scope of the assignment, but if you are interested in learning more then we recommend reading *Introduction to Modern Cryptography, Second Edition*, Chapter 3, by Katz and Lindell

C	*runtime*	C	*runtime*	C	*runtime*	C	*runtime*
0	21	4	23	8	29	12	39
1	21	5	33	9	31	13	51
2	23	6	27	10	35	14	47
3	31	7	27	11	45	15	49

3. **Constant-time fix (10 points).** Fix the timing channel in the program from Part 2 so that the runtime no longer depends on the value of d. If it helps make your answer more clear, you can assume that the language contains a $\mathrm{mod}(x, N)$ primitive, but you must also assume that it runs in $\lfloor \frac{x}{N} \rfloor$ units of time. What is the runtime of your fixed implementation?

4. **Randomized enough? (10 points).** Recall the randomized response mechanism discussed in Lecture 14. It flips a fair coin (i.e., one with equal probability 1/2 or returning 0 or 1). If the coin comes heads, then it returned the contents of $\texttt{Mem}(0)$ (which we assumed to be either 0 or 1). If the coin comes up tails, then it flips another fair coin and returns the value. We saw that this satisfies $\ln(3)$-differential privacy.

   Consider the following variant, which computes a function of both $\texttt{Mem}(0)$ and $\texttt{Mem}(1)$.

$$
\begin{array}{l}
b := \mathsf{flip}(p) \\
\textbf{if } b = 1 \textbf{ then} \\
\quad o := \texttt{Mem}(0) \\
\textbf{else} \\
\quad o := \mathsf{flip}(p) + \texttt{Mem}(1)
\end{array}
\tag{1}
$$

   Use Definition 2 from Lecture 14 to answer this question. Does this program satisfy differential privacy for any value of $\epsilon > 0$? If so, explain why. If not, give a counterexample pair of neighboring databases for which the bound in Equation 8 (Lecture 14) cannot hold for any $\epsilon > 0$, and explain how to modify the program to make it satisfy differential privacy.