Assignment 5: Relaxed Secrecy and Privacy
15-316 Software Foundations of Security and Privacy

Total Points: 50 (+15 possible extra credit)

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: `gt`(`guess`, `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: `ecm`(`guess`, `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: `pref`(`guess`, `pin`) evaluates to `true` if and only if `guess` is a prefix of the n-bit string `pin`.

2. **Exclusive interference (20 points).**

Consider the following program under the policy $\Gamma = (a : \texttt{H}, b : \texttt{H}, c : \texttt{L})$.

```
if(a > 0) {
  if(b > 0) {
    c := 0;
  } else {
    c := 1;
  }
} else {
  if(b > 0) {
    c := 1;
  } else {
    c := 0;
  }
}
```

Part 1 (10 points). Although this program does not satisfy noninterference, does it leak any information about the H variables a and b to an observer who sees the initial and final values of c? Describe the feasible set of initial values of a, b to justify your answer.

Part 2 (10 points). Notice that when a and b take values in $\{0, 1\}$, this code implements the exclusive-or operation, storing the result of $a \oplus b$ in c. Building on your insights from Part 1, design a corresponding declassification rule for exclusive-or by filling in the missing parts of the premises and conclusion in (DeclassXor) below.

$$\text{(DeclassXor)} \quad \frac{\Gamma \vdash e : \square \quad \Gamma \vdash \widetilde{e} : \square \quad e, \widetilde{e} \text{ evaluate to either } \{0, 1\}}{\Gamma \vdash e \oplus \widetilde{e} : \square}$$

You may assume that the security lattice is $\texttt{L} \sqsubseteq \texttt{H}$. Your rule should permit cases where learning the output of the operation will not reduce the uncertainty about H-typed variables of an attacker who observes the value of all variables that Γ types as L.

3. **Randomized enough? (15 points).** Recall the randomized response mechanism. 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 the input (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 computations. Which of them satisfy differential privacy for some $\epsilon > 0$? Note that in cases where the functions *do* satisfy differential privacy, you do not need to compute ϵ exactly, just explain why the definition is satisfied for some ϵ. In cases where privacy is not satisfied, provide a pair of neighboring inputs for which the bound in Equation 8 of the privacy lecture cannot hold for any $\epsilon > 0$.

   (a) (5 points) Assume that that the contents of z are always either 0 or 1:

   $$o := \mathsf{flip}(0.5) + z$$

   (b) (5 points) Assume that the contents of z are in the range of $[0, 100]$:

   if $\mathsf{flip}(0.5) = 1$ **then**
   $o := 100 - z$
   else
   $o := z$

   (c) (5 points) Assume that that the contents of z are always either 0 or 1:

   if $\mathsf{flip}(0.5) = 1$ **then**
   $o := \mathsf{flip}(0.5) \times z$
   else
   $o := z$

4. **Not quite perfect timing (15 points, extra credit).**

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 $C = \text{mod}(M^e, N)$. Likewise to perform decryption given C to recover M, one computes $M = \text{mod}(C^d, N)$. 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. 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;
while (j < L) {
  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 the side channel vulnerability is by describing how to exploit it.

- As the attacker, you may assume that you can run the above code as many times as you like with whatever values of `C` and `d` that you choose, and observe the execution time.
- The attacker is also allowed to run the program arbitrarily many times, providing their choice of `C`, with the secret value of `d` provided as input, and observe the execution time.
- You should assume that each arithmetic operation, comparison, and assignment takes one unit of time, and that you are able to observe the exact number of operations executed when you run the program.

Your answer does not need to be a formal algorithm, but to receive full credit, you should explain the following:

- (10 pts) How to recover a single chosen bit of `d` by observing timing differences when running the program on different inputs for `C`. Your answer to this should not require brute-forcing all of `d` and then selecting the chosen bit. How many times do you need to run the code above to recover the full L-bit key `d`?
- (5 pts) 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 $\text{mod}(x, N)$ primitive, but you must also assume that it runs in $\left\lfloor \frac{x}{N} \right\rfloor$ units of time. What is the runtime of your fixed implementation?