Assignment 5: Relaxed Information Flow & Side Channels
15-316 Software Foundations of Security and Privacy

1. **Exclusive interference (15 points).**

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

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

   Part 1 (5 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`.

2. **Not quite perfect timing (8 points, 10 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 Python program below implements modular exponentiation efficiently. Note that the modulo operation used here is implemented in a very simple manner by repeated subtraction, to make its cost in runtime clear.

```
def exponentiate(C: int, d: int, N: int) -> int:
  r = 1
  # loop until the exponent is 0
  while d > 0:
    # if the exponent is odd, multiply by C
    if d % 2 == 1:
      r *= C
      # only take modulo if necessary
      if N <= r:
        r = r % N
    # square the base
    C = (C*C) % N
    # reduce the exponent
    d = d // 2
  return r
```

For this question, you should assume that all variables except d are public, i.e., the policy is $\Gamma = (\texttt{C} : \texttt{L}, \texttt{d} : \texttt{H}, \texttt{N} : \texttt{L})$.

- **(5 pts).** Identify the timing vulnerability in this code, and explain how it could threaten the security of RSA assuming that an attacker can run `exponentiate` an arbitrary number of times on their chosen values of ciphertext `C` and modulus `N`, with a fixed (but secret) value of `d`. Assuming that $\texttt{N} = 37, \texttt{d} = 13$, provide values of `C` that result in timing discrepancies that leak information about `d`, and explain how these discrepancies refine the attacker's feasible set.

- **(5 pts).** Fix the timing vulnerability in this program so that timing observations no longer leak information about `d`. Your changes should not affect the correct performance of the function, and you should try to make them as minimal as possible. What is the worst-case runtime of your new implementation compared to the original one?

- **(Extra credit, 10 pts).** Provide an algorithm to recover an attacker's chosen bit of `d` by observing timing differences. You should assume that each arithmetic operation, comparison, and assignment takes one unit of time, and that the attacker is able to observe the exact number of operations executed by `exponentiate`. How many times do you need to run the code above to recover the full key `d`, assuming that it is L-bits long?