Assignment 6: Side Channels
15-316 Software Foundations of Security and Privacy

1. **Short circuited (15 points).**

   Short circuit evaluation of Boolean expressions is widely-adopted in programming languages; it is straightforward to implement, and can lead to performance improvements in certain programs. It works as follows:

   - If the expression is a conjunction and the first argument evaluates to $\bot$, then the second argument isn't evaluated.
   - If the expression is a disjunction and the first argument evaluates to $\top$, then the second argument isn't evaluated.
   - Otherwise, both arguments must be evaluated.

   However, one must take care when using short-circuit evaluation over secret data, as it may introduce a timing channel.

   - **(10 points.)** The cost semanatics for "normal" (non-short circuit) evaluation of binary Boolean expressions, which we discussed in lecture, is shown below. Extend these semantics to account for short-circuit evaluation of conjunction and disjunction by providing big-step cost semantics rules for conjunction ($\wedge$) and disjunction ($\vee$) operators that refine the "general" rule for binary operators shown below.

$$\frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}}^{r_1} b_1 \quad \langle \omega, Q \rangle \Downarrow_{\mathbb{B}}^{r_2} b_2}{\langle \omega, P \ \odot \ Q \rangle \Downarrow_{\mathbb{B}}^{r_1 + r_2 + 1} b_1 \odot b_2}$$

- **(5 points).** Demonstrate your understanding of the short-circuit side channel by providing a small program whose timing leaks information about a secret varible.
  - Clearly state which variables are considered secret and which are not.
  - The leak should be specific to short-circuit evaluation: if "normal" evaluation of Boolean expressions is used instead, then there should be no leak.
  - Your program does not need to leak the entire value of its secret through this side channel. Any violation of cost-sensitive non-interference is good enough.
  - Explain in words how the leak works.

2. **Not quite perfect timing (20 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 $C = \text{mod}(M^e, N)$. 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, using the bits of the exponent to potentially reduce the number of multiplications that are needed.

```
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 $\Gamma = (\texttt{C} : \texttt{L}, \texttt{d} : \texttt{H}, \texttt{N} : \texttt{L})$.

- **(10 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.

- **(10 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?