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

Due: **11:59pm**, Thursday 11/10/19
Total Points: 50 (+5 possible extra credit)

1. **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 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:

   - (13 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. *Hint: start with `d[0]`, and generalize your solution for that special case.*
   - (7 pts) How many times do you need to run the code above to recover the full $L$-bit key `d`?

2. **Constant-time fix (15 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 $\text{mod}(x, N)$ 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?

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 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 Mem(0) and Mem(1). Namely, if $b = 1$, then this program releases some information about Mem(0), and otherwise it releases something about Mem(1).

```
$b := \mathsf{flip}(0.5)$
if $b = 1$ then
  $o := \mathsf{flip}(0.5) + \texttt{Mem}(0)$                (1)
else
  $o := \mathsf{flip}(0.5) + \texttt{Mem}(1)$
```

   Does this program satisfy differential privacy for any value of $\epsilon > 0$?

   - If so, use Definition 2 from Lecture 16 to explain why. In particular, argue that the probabilities of any output $o$ obtained from neighboring memories (e.g., $\{0 \mapsto 0, 0 \mapsto 0\}$ and $\{0 \mapsto 0, 0 \mapsto 1\}$ would be neighboring as they differ only in one location) satisfy the inequality for some $\epsilon$.
   - If not, give a counterexample pair of neighboring memories for which the bound in Equation 8 (Lecture 16) cannot hold for any $\epsilon > 0$, and explain how to modify the program to make it satisfy differential privacy for some $\epsilon$. You do not need to prove that it satisfies it for a particular $\epsilon$, but you should articulate how your change addresses any problems in the program above.
   - You may asume that the contents of Mem are always either 0 or 1.

4. **Extra credit (5 points).** This problem refers to the program listed in Question 2. For extra credit, given the following timings for each initial value of $C$ below where $L = 4$, 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.

| $C$ | *runtime* | $C$ | *runtime* | $C$ | *runtime* | $C$ | *runtime* |
|---|---|---|---|---|---|---|---|
| 0 | 19 | 4 | 22 | 8 | 31 | 12 | 46 |
| 1 | 19 | 5 | 76 | 9 | 64 | 13 | 118 |
| 2 | 31 | 6 | 40 | 10 | 55 | 14 | 76 |
| 3 | 58 | 7 | 46 | 11 | 94 | 15 | 145 |