This homework is due by the **21 DEC 15:00** via the course Moodle page. Start early!

Instructions. Solutions must be typeset in LaTeX. Your work will be graded on *correctness*, *clarity*, and *conciseness*. You should only submit work that you believe to be correct; if you cannot solve a problem completely, you will get significantly more partial credit if you clearly identify the gap(s) in your solution. It is good practice to start any long solution with a "summary" that describes the main answer.

1. If the attacker has access to no side-channel (i.e., it does not have any knowledge about the program execution), at most how many attempts will be required to guess the password correctly? You can assume that the attacker has access to the program source code, and hence is aware of the assumptions about the password. Please state the number of attempts required and your reasoning. (3 points for number of attempts, 2 points for reasoning)

   Solution: In the worst case it would take 26^{15} for T1 password checker and 26^{31} for T2 password checker. This is due to the fact that the attacker only obtains information about the correctness of his/her guess, i.e. if it exactly matches the password or not. Since the only other source of information from a run program, apart from it's output, is the knowledge of it's execution (timing, cache pattern...) which is not available to us, we can't deduce anything else apart from this correctness.

2. For Task T1, if you as an attacker were allowed to select the input i.e., guess the password:

   (a) for all possible passwords accepted by the program, can you extract the secret password with a single guess? (2 points)

   Solution: Yes. The attacker can give any string that is of length 15 (e.g. $a * 15$) as a guess input to the check password program and he/she will be able to extract the whole password by reading through the execution trace produced.

   (b) If so, what is the exact attack strategy to craft the guess and analyze the trace? (3 points)

   Solution: As said above the guess provided can be any string of length 15. Length 15 is necessary since we don't know the exact length of the password, but know that it can be a maximum of 15 chars long.
   The strategy for analyzing the trace would be the following:

   1. Run `objdump -S -M intel password_checker1` in order to produce the assembly code of the executable `password_checker1` with embedded corresponding source code.
   2. From this extract the virtual address of one assembly instruction responsible for `k++`, one for `pow(p[pos], i[pos])` one that identifies the end for the for loop `for (j = distance; j > 0; j--)` and one for the `ADD(p[pos], i[pos])`.

3. Manage a bijection between 0,...,25 and $a, ..., z$ such that $0 \rightarrow a$, $1 \rightarrow b$, $2 \rightarrow c$ ... $25 \rightarrow z$
4. Keep two counters, `char_pointer` and `diff_char_size`. The former corresponds to the current character in the password we are trying to extract. The latter is responsible for counting the wrap around difference between the current password and guess character.
5. For each value of `char_pointer` increment `diff_char_size` each time an `ADD(p[pos], i[pos])` address occurs. When `for(j = distance; j > 0; j--)` address' reached, map the current guess character to it's corresponding number via the bijection. To this number add the `diff_char_size` modulo 26. This is the current password char. (This works because the for loop runs for the `diff_char_size` amount of times, where `diff_char_size = p[char_pointer] - i[char_pointer] mod 26`, `p` is the password and `i` is the guess string)
6. Increment `char_pointer` each time `k++` or `for(j = distance; j > 0; j--)` address is found. In the first case the guess character is correct and hence we know the password character at this `char_pointer` is the same as the guess at that index. In the second we have counted the `diff_char_size` between the guess and password character at `char_pointer`, and hence can derive the password character by doing the above `mod 26` calculation. In either case we deduce the password character at `char_pointer`.
7. We then repeat the process and increment `char_pointer` up to `MIN(p_size, i_size)`, where `p_size`, `i_size` is the size of the password and guess respectively. Therefore with this method we can deduce the password characters at index 0 through `MIN(p_size, i_size)`. Since the length of our guess is 15 and we know that the password has a maximum length of 15, we know that `MIN(p_size, i_size) = p_size`, and hence at the end of this process we will obtain the whole password.

3. For Task T2, for all possible passwords accepted by the program, at most how many attempts will be required to guess the password correctly? You can assume that the attacker has access to the program source code, and hence is aware of the assumptions about the password. Please state the number of attempts required and your reasoning. (2 points for number of attempts, 3 points for reasoning)

Solution: At most 25 attempts will be necessary to guess the password correctly.
The strategy is as follows:

1. Run `objdump -S -M intel password_checker2` in order to produce the assembly code of the executable `password_checker2` with embedded corresponding source code.
2. From this extract the virtual address of one assembly instruction responsible for `k++`,and one for `j++`.
3. Set the guess to be a*31, where 31 is the maximum length the password could be and execute the tracer on `password_checker2` with that guess as input.

4. Maintain a `char_pointer` that represents the index in our guess we are currently testing against the password at that same index. Create a list of $-$ characters of length 31. We will use this list to be save the password we retrieve letter by letter. Finally maintain a counter `pass_length` that will count the number of characters in the password.
5. If the password and guess character at index `char_pointer` are the same, the instruction corresponding to `k++` would be in the execution trace. If on the other hand the guess character and password don't match, `j++` would.
6. Therefore if we read `k++` we will write a into the list at index `char_pointer` and increment `char_pointer` by 1. If we read `j++` we would only increment `char_pointer` by 1. (since `j++` occurs in case of us making the wrong guess at that index)
7. We continue this process until there are no more `k++` or `j++` in the trace.
8. On top of this, we establish the length of the password by incrementing `pass_length` counter by 1 any time a `k++` or `j++` has happened in this trace generated by a*31. We only establish the length in this first trace. We then create a new list of $-$ characters of length `pass_length` and copy the first `pass_length` characters from the 31 character long list we created at the start.
9. We repeat the same process with traces generated with guess b*31 and so on until and including y*31.
10. After that, all the remaining empty slots (denoted in our list by $-$) in the password list must be z. Hence only 25, not 26 traces are necessary to establish the password.

4. Imagine that the `password_checker_1` program was being executed inside Intel SGX enclave. A real-world attacker cannot directly access the password file. To crack the password, the attacker wants to extract some information, similar to what we provided to you in the sample execution traces. State what exact information should the attacker extract and the corresponding side channels that they can use. Please limit to maximum two types of information, one sources for each. (2 point each for stating type of information, 2 point each for stating the corresponding side-channel)

Solution: The attacker can make use of different execution times of `password_checker_1` that are caused solely because of the different guesses provided to it. Therefore the side channel that can be used for this is the timing side channel.
The strategy behind extracting the password is as follows:

1. Run `password_checker_1` 26 times, where the guess for the first run is `a`, for the second `b` and so on until guess 26 that is `z`. For each of these 26 runs measure the time it takes to execute it.
2. From these 26 executions take the one that took the shortest amount of time. The guess given to this execution is the first character of the password. Let this character be α.

3. Run `password_checker_1` 26 times and measure it's execution time, but now first on guess α`a`, followed by α`b` and so on until α`z`. The guess with the shortest execution time is the correct 2 character long prefix of the password.
4. Repeat this process until a `correct` is printed on standard output by the binary. Each time concatenate the password prefix obtained so far to one of the 26 possible letters. Time the execution of `password_checker_1` for each of the 26 possible prefixes and take the one with the shortest execution as the next prefix.

The reason why the shortest execution is the correct prefix is because a correct character will only result in `k++` being executed instead of all the instructions on lines 21-32 which are executed in case of the appending character being wrong. Going character by character ensures that the execution times variation occurs solely because of the last character we are appending being wrong or right. In this way we can obtain the password in 26×15 iterations at most, much less than the whole solution space of 26^{15} passwords.

5. Pick one type of information + side-channel you stated in Q4, describe 2 concrete defenses to prevent that leakage. (3 points per defense scheme)

Solution: For the timing side channel we can make use of the following defenses:

1. In case of the wrong character, we can do `j++` instead of all the instructions on lines 21-32. This way every guess of the same length has the same execution time, and the only information that can be obtained from guesses with different lengths is that one has less characters than the other or the password length.
2. Adding random noise in order to make the timings indistinguishable on different inputs.