• Remember to register your attendance using the UoB Check-In app. Either

1. download, install, and use the native app[a] available for Android and iOS, or
2. directly use the web-based app available at

https://check-in.bristol.ac.uk

noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.

• The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

https://www.bristol.ac.uk/it-support

• The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

https://cs-uob.github.io/COMS30048/vm

• The purpose of the worksheet is to provide a) a tutorial-style introduction to selected technologies or concepts, and/or b) a means to explore them via hands-on tasks and challenges. Note that the worksheet is not assessed *at all*: if you are confident that you already understand the content, there is no problem with nor penalty for totally ignoring it.

• Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance

[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.

[c] https://www.vagrantup.com

COMS30048 lab. worksheet #4

Before you start work, download (and, if need be, unarchive[a]) the file

https://assets.phoo.org/COMS30048_2025_TB-2/csdsp/sheet/lab-04.tar.gz

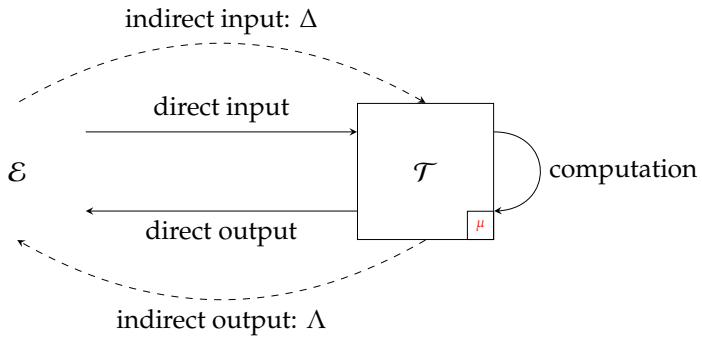
somewhere secure[b] in your file system; from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content. The archive content is intended to act as a starting point for your work, and will be referred to in what follows.

[a] For example, you could 1) use `tar`, e.g., by issuing the command `tar xvfz lab-04.tar.gz` in a terminal window, 2) use `ark` directly: use the `Activities` desktop menu item, search for and execute `ark`, use the `Archive→Open` menu item to open `lab-04.tar.gz`, then extract the contents via the `Extract` button, or 3) use `ark` indirectly: use the `Activities` desktop menu item, search for and execute `dolphin`, right-click on `lab-04.tar.gz`, select `Open with`, select `ark`, then extract the contents via the `Extract` button.

[b] For example, the `Private` sub-directory within your home directory (which, by default, cannot be read by another user).

1. Introduction

Concept. Although greater precision can be important, a reasonable and intuitive way to define the concept of a (cryptographic) implementation attack is to pitch it as an alternative to traditional cryptanalysis: the latter focuses on an abstract specification (e.g., the design of a protocol or primitive) of a given target, whereas the former focuses on a concrete implementation (i.e., hardware and/or software realising said protocol or primitive) and so behaviour of said target. By implication, this positions the two styles of attack as practice-centric (e.g., Engineering-centric) and theory-centric (e.g., Mathematics-centric) respectively, although, in reality, they overlap; a clean separation is impossible. At a high level, implementation attacks can be modelled as follows:



There are two parties[1] involved, namely a) the attacker $\mathcal{E}$ (left) and b) the attackee, or target $\mathcal{T}$ (right); you could think of $\mathcal{E}$ as a malicious user of $\mathcal{T}$, with the latter executing some functionality (i.e., performing computation) on behalf of the former. Communication between the parties uses a pre-defined protocol (or API), such that $\mathcal{T}$, for example, expects to

1. receive direct input from $\mathcal{E}$,
2. perform some sort of computation, then
3. send direct output to $\mathcal{E}$.

If the model captures security-related functionality, security-critical data is typically involved. Examples include a) key material embedded in $\mathcal{T}$ (in this case denoted by μ), or b) encrypted messages sent by $\mathcal{T}$. $\mathcal{E}$ cannot access this target data directly, so their goal will be to recover it by other means; using the same examples, this implies their goal is to mount a successful key recovery or plaintext recovery attack respectively. Based on the *direct* input and output alone, one strategy would be to use traditional cryptanalytic techniques. However, the model allows an alternative: it is *also* possible for $\mathcal{E}$ to harness *indirect* inputs and/or outputs to and/or from $\mathcal{T}$. The former is exemplified by $\mathcal{E}$ actively exerting influence over, and so altering the behaviour of $\mathcal{T}$ (cf. fault attack[2]), and the latter by $\mathcal{E}$ passively monitoring the behaviour of $\mathcal{T}$ (cf. side-channel attack[3]). The idea is that by doing so, $\mathcal{E}$ might improve their attack with respect to metrics such as effectiveness (e.g., whether or how precisely they can recover μ) or efficiency (e.g., the resources, such as time, required to recover μ).

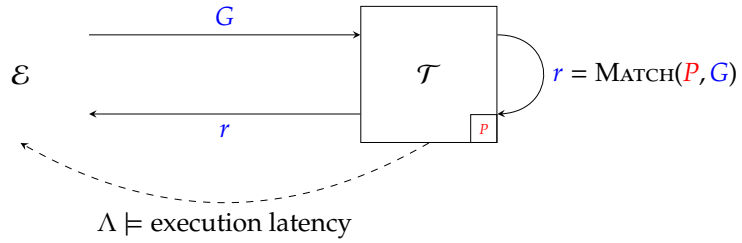
[1] This model is intended to capture a broad set of possibilities: one can see, for example, that it captures a traditional client-server application (with $\mathcal{E}$ as a web-browser and $\mathcal{T}$ as a web-server), or a payment application (with $\mathcal{E}$ and $\mathcal{T}$ as chip-and-pin terminal and card). The communication medium and wire protocol used are irrelevant, but use of local (e.g., USB) or remote (e.g., the Internet) instances are valid depending on the scenario.

[2] http://en.wikipedia.org/wiki/Differential_fault_analysis

[3] http://en.wikipedia.org/wiki/Side-channel_attack

Example. The importance of implementation attacks can be motivated by observing that with a goal of sufficient value, an attacker might not behave as they *should* (i.e., what we expect) but however they *can*. To render ineffective the security afforded by some encryption scheme, for example, they might target 1) the abstract, "on paper" specification *or* 2) the concrete, in practice implementation of said scheme. Typically they will select the option for which they can succeed most easily (colloquially, the weakest link). If security of the scheme is (provably) related to some hard Mathematical problem, traditional cryptanalysis is unlikely to be easy at all. Opting for an implementation attack, however, offers the potential to bypass the associated difficulty: the hard problem might be rendered moot if the attack instead focuses on a feature (e.g., flaw) in the implementation of it.

Consider a simple example scenario, intended to offer a starting point within this field and thus focus on the central concepts. Rather than cryptography, it deals with a more general security challenge: imagine $\mathcal{T}$ as representing a UNIX-based computer, which a user (there is only one, so no user name is required) can log into[4] by supplying the correct password[5]. Per the abstract overview above, this now concrete scenario can be modelled as follows:



The protocol, which involves both indirect input and output, is such that $\mathcal{T}$

1. receives a password attempt, or guess G from $\mathcal{E}$,
2. compares G with P, the correct, stored password, then
3. sends the result r to $\mathcal{E}$.

Even if r was *not* sent to $\mathcal{E}$, it would clearly still learn whether the associated attempt G was correct: if the attempt is incorrect then access to $\mathcal{T}$ is allowed, whereas if the attempt is correct then access is disallowed. Modulo the (non-trivial) challenge of selecting a "strong" P, if the attacker $\mathcal{E}$ wants access to $\mathcal{T}$ they *seem* forced into a strategy of guessing the password. However, notice that some indirect output (per the lower, dashed arrow) is also available. More specifically, $\mathcal{E}$ can measure how long $\mathcal{T}$ takes to compute r (i.e., the execution latency of MATCH), by simply measuring how long it takes respond.

The goal of this lab. worksheet is to explore the premise that, in this case and often more generally, doing so enables $\mathcal{E}$ to mount an effective, efficient attack on $\mathcal{T}$. It uses a hands-on but pragmatic approach, centered around simulated instances of $\mathcal{T}$ and $\mathcal{E}$. Put another way, it ignores what $\mathcal{E}$ and $\mathcal{T}$ *are*, in specific terms, and instead models them using programs; doing so is less realistic by definition, but, as a trade-off, allows a focus on what $\mathcal{E}$ and $\mathcal{T}$ *do*, and hence on the underlying concepts.

2. Developing simulated attack target and attack implementations

2.1. An attack target implementation

The first problem is how to realise a simulated version of $\mathcal{T}$, the attack target; the C source code in Figure 1 acts as an example solution. Focusing on Figure 1b specifically, two functions constitute the implementation:

1. `match` (Lines #10 to #31 of Figure 1b) compares `P` and `G`: it produces

$$\texttt{r} = \begin{cases} 0 & \text{if } \texttt{P} \neq \texttt{G} \\ 1 & \text{if } \texttt{P} = \texttt{G} \end{cases}$$

   The function has three main steps:

   - the `strlen` function is used to compute the lengths of (i.e., number of characters in) each string,
   - an `if` statement checks whether or not the lengths are equal (if not, the strings do not match), then
   - a `for` statement compares corresponding characters within the strings (if any are unequal, the strings do not match).

   Note that, in doing so, it uses `Lambda` to keep track of the number of steps taken; this is used as a proxy for the real, wall-clock execution latency.

[4] http://en.wikipedia.org/wiki/Login
[5] http://en.wikipedia.org/wiki/Password

```c
#ifndef __TARGET_H
#define __TARGET_H

#include <stdbool.h>
#include  <stdint.h>
#include   <stdio.h>
#include  <stdlib.h>

#include  <string.h>

#endif
```

(a) *${ARCHIVE}/target.h*

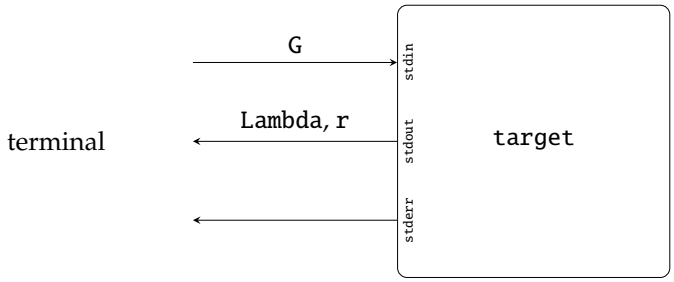
```c
#include "target.h"

int match( int* r, const char* P, const char* G ) {
  int Lambda;

  int l_P = strlen( P );
  int l_G = strlen( G );

  Lambda = 0;

  if( l_P != l_G ) {
      *r = 0; return Lambda;
  }

  Lambda = 1;

  for( int i = 0; i < l_G; i++, Lambda = Lambda + 1 ) {
    if( P[ i ] != G[ i ] ) {
      *r = 0; return Lambda;
    }
  }

      *r = 1; return Lambda;
}

int main( int argc, char* argv[] ) {
  char G[ 8 + 1 ], P[] = "password";

  while( true ) {
    int Lambda, r;

    if( 1 != fscanf( stdin, "%8s", G ) ) {
      abort();
    }

    if( feof( stdin ) ) {
      break;
    }

    Lambda = match( &r, P, G );

    fprintf( stdout, "%d\n", Lambda );
    fprintf( stdout, "%d\n", r      );

    fflush( stdout );
    fflush( stderr );
  }

  return 0;
}
```

(b) *${ARCHIVE}/target.c*

Figure 1: *A simulated attackee, or target matching the example in Section 1.*

2. `main` (Lines #33 to #57 of Figure 1b) implements the protocol required to model $\mathcal{T}$, using a `while` statement to iterates (indefinitely) over three main steps: each iteration will
   - read input from `stdin` (breaking from the loop if there is no input to read),
   - invoke `match` to compare `P` and `G`, then, finally,
   - write output to `stdout`.

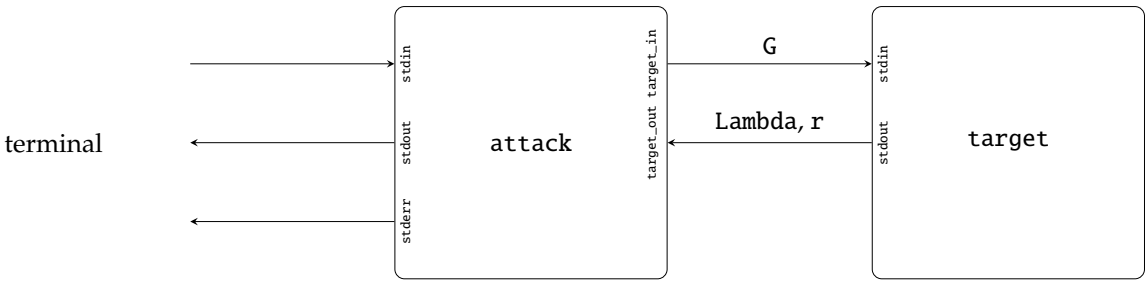
Using a terminal[6] to execute the compiled result will reproduce this diagram:



Put another way, the executing instance of `target` has, in common with any other UNIX-based process, standard streams[7] termed `stdin`, `stdout`, and `stderr`, which it uses for input, output, and errors respectively, and which are connected to the terminal in which it was executed by default. As such, a) input to the executing instance of `target` is read from the terminal via `stdin`, and b) output from the executing instance of `target` is written to the terminal via `stdout`.

2.2. An attack(er) implementation

Next we turn our attention to a simulated version of $\mathcal{E}$, the attack(er). Assuming it is realised as another executable, this time called `attack`, the goal is to reproduce the following



which implies the (simulated) attacker executes the (simulated) attack target as a sub-process; the former essentially automates interaction with the latter, using the same mechanisms (i.e., `stdin` and `stdout`) as in the previous, manual case.

The problem is, *how* can this be achieved? To provide an answer, several implementations of `attack` are provided: each one has a roughly similar structure, but uses a different programming language. In each case, execution is initiated from some entry point (e.g., a `main` function in C) which produces two separate processes (as implied by the diagram above):

1. a (sub-)process representing $\mathcal{T}$ formed by executing `target`, and
2. a process representing $\mathcal{E}$ formed by calling the `attack` function. This process interacts with the executing instance of `target` with support from handles (called `target_in` and `target_out`) on the `stdin` and `stdout` streams attached to `target`, and a function (or method) called `interact` which essentially acts as a layer of abstraction: given a guess `G`, this performs the steps required to interact with `target`.

We focus on the Python implementation shown in Figure 2 as an exemplar, because it is more concise (and arguably "cleaner") and illustrates the concepts more clearly. Beyond this, some language-specific detail follows:

- `attack.py` is a Python-based implementation; it is relatively simple, in part due to use of the powerful `subprocess` module.
- `attack.[ch]` is a C-based implementation; it is relatively complex, in part because use of C demands lower-level interaction with the kernel (via system calls for process creation and control).

[6] That is, within a BASH shell (or prompt, e.g., a terminal window) or similar.
[7] https://en.wikipedia.org/wiki/Standard_streams

```
7  import sys, subprocess
8
9  def interact( G ) :
10   # send              G   to   attack target.
11
12   target_in.write( ( '{0:s}\n'.format( G ) ).encode( 'ascii' ) ) ; target_in.flush()
13
14   # receive ( Lambda, r ) from attack target.
15
16   Lambda = int( target_out.readline().strip() )
17   r      = int( target_out.readline().strip() )
18
19   return ( Lambda, r )
20
21 def attack() :
22   # select a hard-coded guess ...
23
24   G = 'pencil'
25
26   # ... then interact with the attack target.
27
28   ( Lambda, r ) = interact( G )
29
30   # print all of the inputs and outputs.
31
32   print( 'G      = {0:s}'.format(      G ) )
33   print( 'Lambda = {0:d}'.format( Lambda ) )
34   print( 'r      = {0:d}'.format(      r ) )
35
36 if ( __name__ == '__main__' ) :
37   # produce a sub-process representing the attack target
38
39   target = subprocess.Popen( args   = sys.argv[ 1 ],
40                              stdout = subprocess.PIPE,
41                              stdin  = subprocess.PIPE )
42
43   # construct handles to attack target standard input and output
44
45   target_out = target.stdout
46   target_in  = target.stdin
47
48   # execute a function representing the attacker
49
50   attack()
```

(a) *${ARCHIVE}/attack.py*

Figure 2: *A simulated attacker matching the example in Section 1.*

The standard C library includes `system`, which can execute external programs, but not interact with them. It is possible to extend this using `popen`, then allowing uni-directional interaction only (e.g., from attacker to attack target *or* vice versa, but not both at once). The most complete approach is therefore to use the low-level `fork` function to create two processes, and have them communicate via two uni-directional pipes (one in each direction) constructed using `pipe`.

Each pipe is represented by a raw file descriptor; this mandates careful use of the unbuffered `read` and `write` functions. The implementation instead converts these to buffered file handles (of type `FILE*`) using `fdopen`, to allow convenient use of `fprintf` and `fscanf`.

3. Some hands-on tasks and challenges

To recap, we now have a simulated attack target and attack(er). However, the latter is benign in the sense that it currently lacks a guessing (or attack) strategy: although it includes the infrastructure required to operate more generally, it simply sends a single, specific guess to the attack target. As such, the natural next step(s) would be to extend the attack implementation so it acts maliciously by actively trying to guess (i.e., recover) the password.

3.1. Build and execute the attack target implementation

1. Fix the working directory:

```
cd ${ARCHIVE}
```

2. Build any material related to the attack target implementation:

```
make all
```

3. Execute the attack target implementation:

```
./target
```

Per Section 2.1, use the terminal to interact with it to 1) provide an attempt/guess as input, then 2) verify whether the output matches what you expect.

3.2. Build and execute the attack implementation

1. Fix the working directory:

```
cd ${ARCHIVE}
```

any material related to the attack implementation:

```
make all
```

2. Execute the Python-based[8]

```
python3 attack.py ./target
```

or C-based

```
./attack ./target
```

attack implementation, and, given it uses the hard-coded attempt/guess `G` = "pencil", verify whether the output matches what you expect. Change the hard-coded guess the attack makes, and see how doing so changes said output.

Note that in *all* cases, use of the file name `target` with no associated path could cause an error if the current (working) directory is not in your `${PATH}` environment variable: basically, the simulated attacker will be unable to find and hence unable to execute it (much like executing `target` manually would fail). This issue is easy to resolve: we simply include a relative or absolute path to the file, e.g., per use of `./target` in the above.

3.3. Implement a naive attack (i.e., using direct input and output only)

1. Obviously the attacker does not know the correct, stored password `P` itself. However, in order to support an attack, what reasonable assumptions could/should be made about `P`?
2. To recover `P`, both brute-force[9] and dictionary-based[10] attack strategies *seem* viable. For *each* strategy, extend the starting point provided to do so. Once the attacks work correctly, compare their efficiency by using big-O notation; are there any *other* metrics that could usefully characterise the strategies?

[8] Depending on the platform you have opted to use, satisfying the dependencies required *may* demand use of a Python virtual environment; Appendix A offers an overview of how to do so.

[9] http://en.wikipedia.org/wiki/Brute-force_attack

[10] http://en.wikipedia.org/wiki/Dictionary_attack

3.4. Implement a side-channel attack (i.e., also using the indirect output)

Thus far, the attacks developed only use `r`, the direct output from the attack target indicating whether or not a guess `G` is correct; the next step is to utilise `Lambda`, the indirect output which models the execution latency.

Looking again at the `match` function in Figure 1 should highlight some facts which can help. Assuming `r` = 0, meaning we know a guess `G` was incorrect, the question to ask is *why* it was deemed incorrect? Roughly speaking, a larger `Lambda` suggests `match` must have successfully completed more comparisons of either the lengths of, or characters in the strings. Or, put another way, a larger `Lambda` suggests `G` is more similar to `P`. For example,

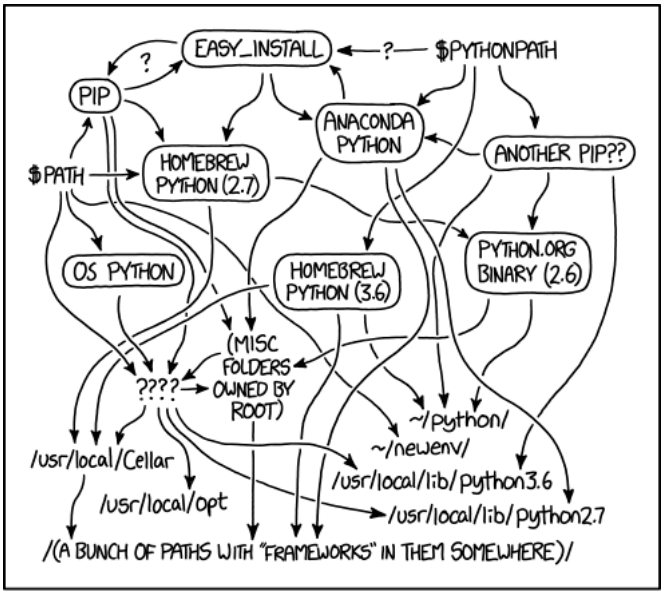
- if `Lambda` = 0, we know `G` was the incorrect length (because `match` *must* have returned in the `if` statement),
- if `Lambda` = 1, we know `G` was the correct length but `P[ 0 ]` ≠ `G[ 0 ]` (because `match` *must* have returned in the iteration of the `for` loop where `i` = 0), and
- if `Lambda` = 2, we know `G` was the correct length and `P[ 0 ]` = `G[ 0 ]` (because `match` *must* have returned in the iteration of the `for` loop where `i` = 1).

Simply put, observing how `Lambda` offers information about where and how to change `G` in order to guess `P` more efficiently.

1. Extend your brute-force attack implementation to capitalise on the indirect output `Lambda` leaked by the attack target. A sensible approach is to realise the new, side-channel attack using two steps: 1) recover the length of, i.e., number of characters in, `P`, then 2) work character-by-character to recover `P` itself. Again using big-O notation, compare this attack strategy with those from above (i.e., brute-force and dictionary attacks).
2. Within the majority of the above, there is an (implicit) assumption that the attack target implements `match` as described. In reality, however, this may *not* be reasonable. If/when an attack is known, it would be more reasonable to assume the implementation is altered to prevent (or at least mitigate) it; this would normally be termed a countermeasure.

   What countermeasures might be implemented in this example? Can you alter the implementation in Figure 1 to realise one such instance? How robust are the countermeasures identified: could one alter the attack strategy to bypass the countermeasure, for example?

A Using a Python virtual environment



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

As alluded to by use of "dependency hell"[11] as a colloquialism, management of software dependencies can represent a significant and, at times, frustrating challenge. Set within the context of Python, the concept of a virtual environment specifically attempts to address this challenge. Working in combination with an associated package manager[12], a virtual environment is a self-contained directory structure into which packages beyond the standard library can be installed. This, for example, avoids the need for their centralised installation (which will often require a higher, e.g., super-user[13] privilege level): the entire mechanism can be user-managed.

Use of virtual environments does not solve *every* problem, as alluded to by the cartoon above; if you opt to work using a unit-specific Vagrant box then you may not even have to use them, because relevant packages are often installed centrally as part of the provisioning step. However, particularly if you opt to work using UoB-managed[14] equipment, virtual environments can offer an important solution you will need to make use of. The Python documentation[15] itself contains an accessible overview, but the general workflow can be summarised as follows:

1. Store the virtual environment path in an environment variable, so it can be referenced easily later:

```
export VENV="${PWD}/venv"
```

2. Initialise the virtual environment:

```
python3 -m venv ${VENV}
```

3. Activate the virtual environment:

```
source ${VENV}/bin/activate
```

   While the virtual environment is active, implying use of packages installed locally within it rather than some centralised installation, the indicator `(venv)` is normally included as part of the shell prompt.

4. Install packages in the virtual environment, e.g.,

```
python3 -m pip install pycryptodomex
```

5. Use the virtual environment somehow.
6. Deactivate the virtual environment:

```
deactivate
```

Note that the virtual environment itself is persistent (in `${VENV}`), but one would need to *re*activate it when using another BASH shell (or prompt, e.g., a terminal window).

[1] http://xkcd.com/1987
[11] http://en.wikipedia.org/wiki/Dependency_hell
[12] See, e.g., https://en.wikipedia.org/wiki/Pip_(package_manager)
[13] https://en.wikipedia.org/wiki/Superuser
[14] Previously, IT Services centrally installed various packages on the workstations in, e.g., MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11). Over time, doing so on a per-unit basis become difficult to manage; this fact led to a shift of policy, meaning use of virtual environments as the default.
[15] See, e.g., https://docs.python.org/3/tutorial/venv.html