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

  http://www.github.com/danpage/COMS30048/blob/COMS30048_2023/vagrant/README.md

- 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 portal) 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] http://www.vagrantup.com

COMS30048 lab. worksheet #5

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

http://tinyurl.com/5h3tux8s/csdsp/sheet/lab-5.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-5.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-5.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-5.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

This lab. worksheet offers a suite of software-based implementation attack challenges, modelled around the more general concept of a Capture the Flag (CTF)[1] exercise; as with the hardware platform introduced by lab. worksheets #1.1 and #1.2, they constitute part of our ongoing (open source) SCALE[2] project. Each challenge is focused on a specific scenario, and designed to allow you to enrich your understanding of the this general field, while, e.g., applying practical skills associated with it. In terms of the concepts involved, you could think of them as similar to but more advanced than lab. worksheet #4.

2. Guidance

- The challenges are described in an overly generic way, using `${USER}` as an identifier that *could* allow their personalisation on a per user basis. However, the lab. worksheet does not do so: there is a *single*, generic set of material denoted `${ARCHIVE}` as normal, meaning you should assume

  `${USER} = lab-5.`

- The behaviour of a given attack target is *simulated* using an executable program. Each such executable was produced (i.e., is the result of compilation) on a platform equivalent to those in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11). As such, it is only *guaranteed* to work on either the same or at least a compatible[3] platform, and, even then, only once the file has appropriate permissions set using `chmod`.
- Interaction with each simulated target requires understanding the representation and conversion of both integers and octet strings (which are essentially human-readable sequences of 8-bit bytes). Appendix A includes a detailed discussion of this issue.
- There are a many challenges, and each one alone could represent a significant amount of effort (both in terms of researching and developing an attack strategy, *and* then implementation an attack based on it). As such, if you use the lab. worksheet at all then a reasonable remit would be to focus on *one* challenge only: at least in the first instance, pick one for which the scenario presented is the most compelling, has the most value (e.g., with respect to the coursework assignment), or is the most familiar (suggesting, e.g., it is the most achievable).

3. Content

3.1. Challenge #1: an attack based on error messages

3.1.1. Background

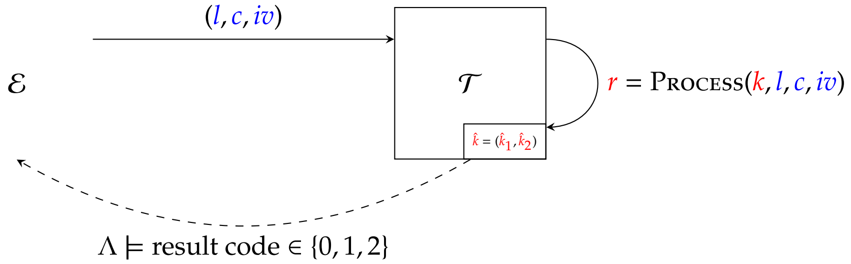
Imagine you encounter a server, denoted $\mathcal{T}$, which acts as a storage repository for sensor data. An installation of 50 or so IoT-class sensor nodes regularly transmit encrypted packets of data to $\mathcal{T}$, which stores them for later

[1] https://en.wikipedia.org/wiki/Capture_the_flag_(cybersecurity)

[2] http://www.github.com/danpage/scale

[3] For instance, one way to reduce your dependency on workstations in the CS Linux lab. is to use a compatible operating system in a VM or as a LiveCD on your own workstation. Alternatives include virtualisation and translation layers, such as Noah (http://www.github.com/linux-noah/noah) for MacOS or WSL (http://docs.microsoft.com/en-us/windows/wsl) for Windows, that allow Linux binaries to execute on other operating systems.

analysis (iff. they are deemed to be valid). Having already captured some encrypted packets transmitted by a target sensor node, you are tasked with recovery of the underlying sensor data. An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (some key material), as follows:


(l, c, iv)
$\mathcal{E}$
$\mathcal{T}$
$r = \textsc{Process}(k, l, c, iv)$
$\hat{k} = (\hat{k}_1, \hat{k}_2)$
$\Lambda \models$ result code $\in \{0, 1, 2\}$

Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (a ciphertext) to $\mathcal{T}$, then 2) $\mathcal{T}$ performs the computation detailed (i.e., it processes the ciphertext, as detailed further below); note that *no* output is produced by $\mathcal{T}$, implying, e.g., that it uses the provided input internally. In more detail, the algorithm Process which captures the computation performed by $\mathcal{T}$ can be described by the following steps:

1. decrypt the l-block c, i.e., compute

   $$m \parallel \tau \parallel \rho = \text{AES-128-CBC.}\textsc{Dec}(\hat{k}_1, iv, c)$$

   then

2. check whether ρ is valid, aborting immediately and produce result code 1 if not, then

3. check whether τ is valid, i.e., whether

   $$\text{HMAC-SHA-1.}\textsc{Ver}(\hat{k}_2, m, \tau) = \textbf{true},$$

   aborting immediately and produce result code 2 if not, then

4. process m somehow, and produce result code 0.

Since there is no (external) output from $\mathcal{T}$ (e.g., relating to r), one might question how $\mathcal{E}$ obtains the (internal) result code. Doing so is plausible, because $\mathcal{E}$ can use execution latency as a proxy: since computation by $\mathcal{T}$ early-aborts depending on the error condition, observation of the execution latency is (modulo any experimental noise) suggestive of whether, when, and what error has occured.

3.1.2. Material

`${ARCHIVE}/aes_padding/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- l, a length (represented as a decimal integer string),
- c, an l-block AES-128-CBC ciphertext (represented as a length-prefixed, hexadecimal octet string), and
- iv, a 1-block AES-128-CBC initialisation vector (represented as a length-prefixed, hexadecimal octet string),

from `stdin` using one field per line, and then write the following output (one field per line)

- Λ, a result code (represented as a decimal integer string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error. Note that:

- To avoid attacks that leverage execution latency, $\mathcal{T}$ employs a high-performance yet constant-time AES-128 implementation (operated in CBC mode) that is derived from [8]; this implies 128-bit block and cipher key lengths.
- Keep in mind some limitations, namely $0 \leq l < 256$, on maximum plaintext and/or ciphertext length.
- The PKCS#7 v1.5 [7, Section 10.3] padding scheme is used: given a block size of 16 bytes, define $p = 16 - (|m \parallel \tau| \bmod 16)$ which implies that $1 \leq p \leq 16$. The padding can then be described as

  $$\rho = \underbrace{\langle p, p, \ldots, p \rangle}_{p \text{ octets}},$$

  i.e., a sequence of p octets each of whose value is p.

`${ARCHIVE}/aes_padding/${USER}.conf` This file represents a set of attack parameters, including everything (e.g., all public values) that $\mathcal{E}$ has access to by default. More specifically, it contains

- $\hat{c}$, a 1-block AES-128-CBC ciphertext (represented as a length-prefixed, hexadecimal octet string), corresponding to an encryption of some unknown plaintext $\hat{m}$ (using $\hat{iv}$), and
- $\hat{iv}$, a 1-block AES-128-CBC initialisation vector (represented as a length-prefixed, hexadecimal octet string),

using one field per line. More specifically, this represents the previously captured network traffic whose decryption, i.e., recovery of the underlying plaintext $\hat{m}$, is the task at hand. Keep in mind that $\hat{m}$ will include the SHA-1 hash of `${USER}` as the least-significant octets: this allows candidate decryptions to be checked for validity.

3.1.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{m}$. When executed using a command of the form

   ```
   ./attack ${USER}.T ${USER}.conf
   ```

   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.
2. Answer the written, exam-style questions in `${ARCHIVE}/aes_padding/${USER}.exam`.

3.2. Challenge #2: an attack based on error messages

3.2.1. Background

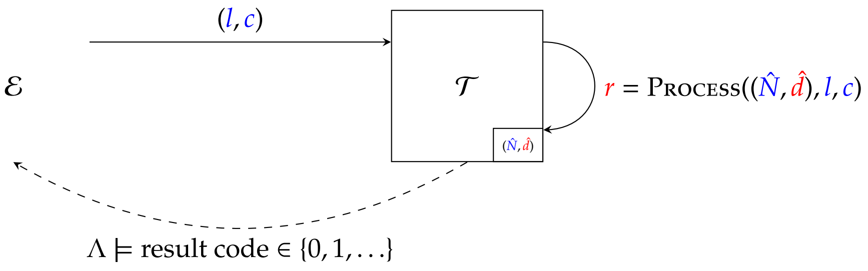
PKCS#1 v2.1 [6] specifies the RSAES-OAEP decryption [6, Section 7.1] scheme, which, in turn, makes use of Optimal Asymmetric Encryption Padding (OAEP) [2]. Crucially, the decryption process can encounter various error conditions:

Error #1: A decryption error occurs in Step 3.*g* of RSAES-OAEP-Decrypt if the octet string passed to the decoding phase does *not* have a most-significant $00_{(16)}$ octet. Put another way, the error occurs because the output produced by RSA decryption is too large to fit into one fewer octets than the modulus.

Error #2: A decryption error occurs in Step 3.*g* of RSAES-OAEP-Decrypt if the octet string passed into the decoding phase does *not* a) produce a hashed label that matches, or b) use a $01_{(16)}$ octet between any padding and the message. Put another way, the error occurs because the plaintext validity checking mechanism fails.

A footnote [6, Section 7.1.2] explains why all errors, these two in particular, should be indistinguishable from each other; a given application should not reveal *which* error occurred, even if it reveals *some* error occurred.

Imagine you form part of a red team[4] asked to assess a specific e-commerce server, denoted $\mathcal{T}$, which houses a 64-bit Intel Core2 micro-processor. In reality, the server is a HSM-like device representing the back-office infrastructure that supports various secure web-sites: the server offers secure key generation and storage, plus off-load for some cryptographic operations. $\mathcal{T}$ is able to compute RSAES-OAEP decryption, but, crucially, does *not* adhere to the advice re. error indistinguishability outlined above. An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (a private key), as follows:


(l, c)
$\mathcal{E}$
$\mathcal{T}$
$(\hat{N}, \hat{d})$
$r = \text{Process}((\hat{N}, \hat{d}), l, c)$
$\Lambda \models$ result code $\in \{0, 1, \ldots\}$

Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (a ciphertext) to $\mathcal{T}$, then 2) $\mathcal{T}$ performs the computation detailed (i.e., it processes the ciphertext, as detailed further below); note that *no* output is produced by $\mathcal{T}$, implying, e.g., that it uses the provided input internally. During such an interaction, $\mathcal{E}$ is *also* able to observe (i.e., measure) the computation performed by $\mathcal{T}$, so attempts to exploit this fact. In more detail, the algorithm Process which captures the computation performed by $\mathcal{T}$ can be described by the following steps:

[4] https://en.wikipedia.org/wiki/Red_team

1. decrypt c, i.e., compute
$$m = \text{RSAES-OAEP.}\textsc{Decrypt}((\hat{N}, \hat{d}), c),$$
and, in doing so, deal with various error conditions:
    - if error #1 occurred during decryption, abort immediately and produce result code 1,
    - if error #2 occurred during decryption, abort immediately and produce result code 2,
    - if there was some other internal error (e.g., due to malformed input), abort immediately and produce a result code that attempts to diagnose the cause:
        - if the result code is 3 then RSAEP failed because the operand was out of range (section 5.1.1, step 1, page 11), i.e., the plaintext is not between 0 and $\hat{N} - 1$,
        - if the result code is 4 then RSADP failed because the operand was out of range (section 5.1.2, step 1, page 11), i.e., the ciphertext is not between 0 and $\hat{N} - 1$,
        - if the result code is 5 then RSAES-OAEP-Encrypt failed because a length check failed (section 7.1.1, step 1.*b*, page 18), i.e., the message is too long,
        - if the result code is 6 then RSAES-OAEP-Decrypt failed because a length check failed (section 7.1.2, step 1.*b*, page 20), i.e., the ciphertext does not match the length of $\hat{N}$,
        - if the result code is 7 then RSAES-OAEP-Decrypt failed because a length check failed (section 7.1.2, step 1.*c*, page 20), i.e., the ciphertext does not match the length of the hash function output.
        - any other result code (of 8 upward) implies an abnormal error whose cause cannot be directly associated with any of the above.

2. process m somehow, and produce result code 0.

Since there is no (external) output from $\mathcal{T}$ (e.g., relating to r), one might question how $\mathcal{E}$ obtains the (internal) result code. Doing so is plausible, because $\mathcal{E}$ can use execution latency as a proxy: since computation by $\mathcal{T}$ early-aborts depending on the error condition, observation of the execution latency is (modulo any experimental noise) suggestive of whether, when, and what error has occured.

3.2.2. Material

`${ARCHIVE}/rsa_padding/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- l, an RSAES-OAEP label (represented as a length-prefixed, hexadecimal octet string), and
- c, an RSAES-OAEP ciphertext (represented as a length-prefixed, hexadecimal octet string),

from `stdin` using one field per line, and then write the following output (one field per line)

- Λ, a result code (represented as a decimal integer string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error.

`${ARCHIVE}/rsa_padding/${USER}.conf` This file represents a set of attack parameters, including everything (e.g., all public values) that $\mathcal{E}$ has access to by default. More specifically, it contains

- $\hat{N}$, an RSA modulus (represented as a hexadecimal integer string),
- $\hat{e}$, an RSA public exponent (represented as a hexadecimal integer string), such that $\hat{e} \cdot \hat{d} \equiv 1 \pmod{\Phi(\hat{N})}$,
- $\hat{l}$, an RSAES-OAEP label (represented as a length-prefixed, hexadecimal octet string), and
- $\hat{c}$, an RSAES-OAEP ciphertext (represented as a length-prefixed, hexadecimal octet string), corresponding to an encryption of some unknown plaintext $\hat{m}$ (using $\hat{l}$),

using one field per line. Note that $(\hat{N}, \hat{e})$ is a (known) RSA public key associated with $(\hat{N}, \hat{d})$, i.e., the (unknown) RSA private key, and that $\hat{c}$ is a ciphertext whose decryption, i.e., recovery of the underlying plaintext $\hat{m}$, is the task at hand. You can assume the RSAES-OAEP encryption of $\hat{m}$ that produced $\hat{c}$ used the MGF1 mask generation function with SHA-1 as the underlying hash function. Additionally, $\hat{m}$ will include the SHA-1 hash of `${USER}` as the least-significant octets: this allows candidate decryptions to be checked for validity.

3.2.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{m}$. When executed using a command of the form

   ```
   ./attack ${USER}.T ${USER}.conf
   ```

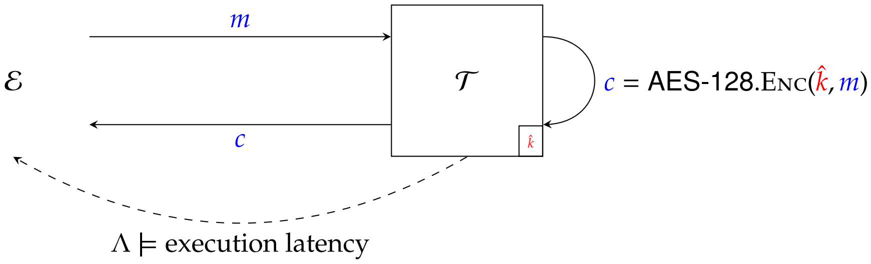
   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.

2. Answer the written, exam-style questions in `${ARCHIVE}/rsa_padding/${USER}.exam`.

3.3. Challenge #3: an attack based on execution time

3.3.1. Background

Imagine you are tasked with attacking a device, denoted $\mathcal{T}$, which houses an 8-bit Intel 8051 micro-processor. More specifically, $\mathcal{T}$ represents an ISO/IEC 7816 compliant contact-based smart-card used within larger devices as a cryptographic co-processor module: using a standardised protocol (or API), it supports secure key generation and storage plus off-load of some cryptographic operations. An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (a cipher key), as follows:



$\mathcal{E}$
m
$\mathcal{T}$
$c = \text{AES-128.Enc}(\hat{k}, m)$
c
$\hat{k}$
$\Lambda \models$ execution latency

Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (a plaintext) to $\mathcal{T}$, 2) $\mathcal{T}$ performs the computation detailed (i.e., it encrypts the plaintext using the cipher key), then 3) $\mathcal{T}$ provides some output (a ciphertext) to $\mathcal{E}$. During such an interaction, $\mathcal{E}$ is *also* able to observe (i.e., measure) the computation performed by $\mathcal{T}$, so attempts to exploit this fact. Note that the ability to observe execution latency is plausible, because $\mathcal{E}$ can simply record how long $\mathcal{T}$ takes to respond with the output associated with a given input (noting the potential for experimental noise while doing so).

3.3.2. Material

`${ARCHIVE}/aes_timing/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- m, a 1-block AES-128 plaintext (represented as a length-prefixed, hexadecimal octet string),

from `stdin` using one field per line, and then write the following output (one field per line)

- Λ, an execution latency measured in clock cycles (represented as a decimal integer string), and
- c, a 1-block AES-128 ciphertext (represented as a length-prefixed, hexadecimal octet string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error. Note that:

- $\mathcal{T}$ uses an AES-128 implementation, which implies 128-bit block and cipher key lengths; following the notation in [1, Figure 5], note that $Nb = 4$ and $Nr = 10$ means a (4×4)-element state matrix is used in a total of 11 rounds.
- More concretely, it is an 8-bit, memory-constrained implementation with the S-box held as a 256 B look-up table in memory. In line with the goal of minimising the memory footprint, the round keys are not pre-computed: each encryption takes the cipher key and evolves it forward, step-by-step, to form successive round keys for use during key addition.

  Crucially, the implementation realises `xtime` (i.e., multiplication-by-`x`, as used in the `MixColumns` round function) by computing it via

```
uint8_t xtime( uint8_t x ) {
  if( x & 0x80 ) {
    return 0x1B ^ ( x << 1 );
  }
  else {
    return        ( x << 1 );
  }
}
```

rather than, e.g., pre-computed and then realised via accesses into a look-up table.

`${ARCHIVE}/aes_timing/${USER}.R` In many side-channel and fault attacks, the attacker is able to perform an initial profiling or calibration phase: a common rationale for doing so is to support selection or fine-tuning of parameters for a subsequent attack. With this in mind, a simulated replica $\mathcal{R}$ of the attack target is also provided. $\mathcal{R}$ is identical to $\mathcal{T}$, bar the fact it reads the following input (one field per line)

- $\bar{m}$, a 1-block AES-128 plaintext (represented as a length-prefixed, hexadecimal octet string), and
- $\bar{k}$, an AES-128 cipher key (represented as a length-prefixed, hexadecimal octet string),

from `stdin`. In contrast to $\mathcal{T}$, this means $\mathcal{R}$ will use the *chosen* cipher key $\bar{k}$ to encrypt the chosen plaintext $\bar{m}$.

3.3.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{k}$. When executed using a command of the form

   ```
   ./attack ${USER}.T
   ```

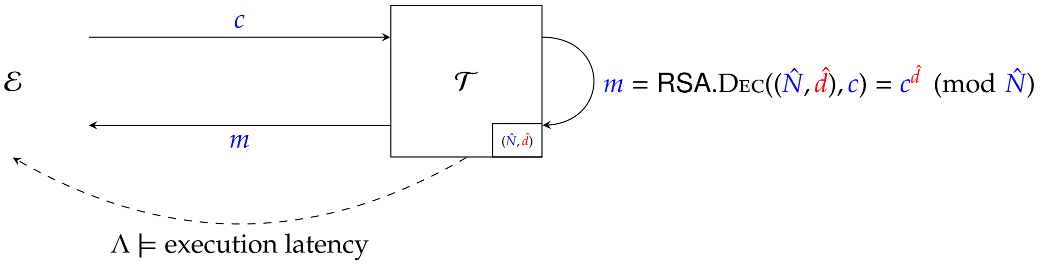
   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.

2. Answer the written, exam-style questions in `${ARCHIVE}/aes_timing/${USER}.exam`.

3.4. Challenge #4: an attack based on execution time

3.4.1. Background

Imagine you are tasked with attacking a server, denoted $\mathcal{T}$, which houses a 64-bit Intel Core2 micro-processor. $\mathcal{T}$ is used by several front-line e-commerce servers, which offload computation relating to TLS handshakes. More specifically, $\mathcal{T}$ is used to compute the RSA decryption required by RSA-based key exchange. An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (a private key), as follows:


c
$\mathcal{E}$
$\mathcal{T}$
m
$(\hat{N}, \hat{d})$
$m = \text{RSA.Dec}((\hat{N}, \hat{d}), c) = c^{\hat{d}} \pmod{\hat{N}}$
$\Lambda \models$ execution latency

Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (a ciphertext) to $\mathcal{T}$, 2) $\mathcal{T}$ performs the computation detailed (i.e., it decrypts the ciphertext using the private key), then 3) $\mathcal{T}$ provides some output (a plaintext) to $\mathcal{E}$. During such an interaction, $\mathcal{E}$ is *also* able to observe (i.e., measure) the computation performed by $\mathcal{T}$, so attempts to exploit this fact. Note that the ability to observe execution latency is plausible, because $\mathcal{E}$ can simply record how long $\mathcal{T}$ takes to respond with the output associated with a given input (noting the potential for experimental noise while doing so).

3.4.2. Material

`${ARCHIVE}/rsa_timing/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- c, an RSA ciphertext (represented as a hexadecimal integer string),

from `stdin` using one field per line, and then write the following output (one field per line)

- Λ, an execution latency measured in clock cycles (represented as a decimal integer string), and
- m, an RSA plaintext (represented as a hexadecimal integer string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error. Note that:

- Because $\mathcal{T}$ houses a 64-bit micro-processor, it employs a base-2^{64} representation of multi-precision integers throughout. Put another way, since $w = 64$, it selects $b = 2^w = 2^{64}$.
- Rather than a CRT-based approach, $\mathcal{T}$ uses a left-to-right binary exponentiation [5, Section 2.1] algorithm to compute $m = c^{\hat{d}} \pmod{\hat{N}}$. Montgomery multiplication [10] is harnessed to improve efficiency; following notation in [9], a CIOS-based [9, Section 5] approach is used to integrate[5] an integer multiplication with a subsequent Montgomery reduction.
- Following the notation in [5], $\hat{d}$ is assumed to have $l + 1$ bits such that $\hat{d}_l = 1$ for some l. However, it has been (artificially) selected so $0 \leq \hat{d} < 2^{64}$ to limit the duration of an attack: you should not rely on this fact, implying your attack could succeed for *any* $\hat{d}$ if afforded enough time.

`${ARCHIVE}/rsa_timing/${USER}.conf` This file represents a set of attack parameters, including everything (e.g., all public values) that $\mathcal{E}$ has access to by default. More specifically, it contains

- $\hat{N}$, an RSA modulus (represented as a hexadecimal integer string), and
- $\hat{e}$, an RSA public exponent (represented as a hexadecimal integer string), such that $\hat{e} \cdot \hat{d} \equiv 1 \pmod{\Phi(\hat{N})}$,

using one field per line. Note that $(\hat{N}, \hat{e})$ is a (known) RSA public key associated with $(\hat{N}, \hat{d})$, i.e., the (unknown) RSA private key.

`${ARCHIVE}/rsa_timing/${USER}.R` In many side-channel and fault attacks, the attacker is able to perform an initial profiling or calibration phase: a common rationale for doing so is to support selection or fine-tuning of parameters for a subsequent attack. With this in mind, a simulated replica $\mathcal{R}$ of the attack target is also provided. $\mathcal{R}$ is identical to $\mathcal{T}$, bar the fact it reads the following input (one field per line)

- $\bar{c}$, an RSA ciphertext (represented as a hexadecimal integer string),
- $\bar{N}$, an RSA modulus (represented as a hexadecimal integer string), and
- $\bar{d}$, an RSA private exponent (represented as a hexadecimal integer string),

from `stdin`. In contrast to $\mathcal{T}$, this means $\mathcal{R}$ will use the *chosen* RSA private key $(\bar{N}, \bar{d})$ to decrypt the chosen RSA ciphertext $\bar{c}$. Keep in mind that $\mathcal{R}$ uses Montgomery multiplication, so functions correctly iff. $\gcd(\bar{N}, b) = 1$.

3.4.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{d}$. When executed using a command of the form

   ```
   ./attack ${USER}.T ${USER}.conf
   ```

   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.

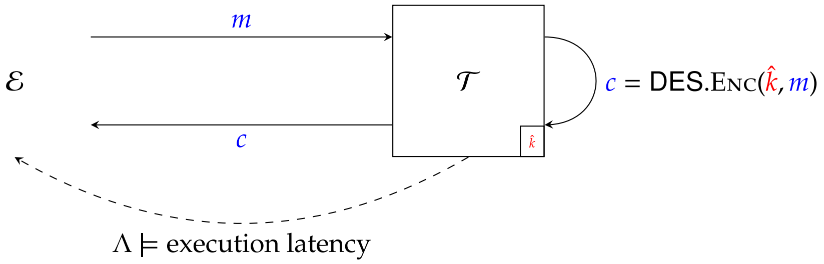
2. Answer the written, exam-style questions in `${ARCHIVE}/rsa_timing/${USER}.exam`.

[5] MonPro [9, Section 2] perhaps offers a more direct way to explain this: steps 1 and 2 are the integer multiplication and Montgomery reduction written both separately and abstractly, whereas $\mathcal{T}$ realises them concretely using the integrated CIOS algorithm.

3.5. Challenge #5: an attack based on execution time

3.5.1. Background

Imagine you encounter a server, denoted $\mathcal{T}$, which houses an ageing but still operational 32-bit Katmai model Intel Pentium III micro-processor. A datasheet for this micro-processor shows that it has a 16 kilobyte, 4-way set-associative L1 data cache with 32 byte cache lines. Since around 1999, the server has performed a mission-critical back-office role in a banking application: you discover it acts as a primitive form of HSM, performing DES encryption on behalf of front-end clients. An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (a cipher key), as follows:


m
$\mathcal{E}$
$\mathcal{T}$
$c = \text{DES.ENC}(\hat{k}, m)$
c
$\hat{k}$
$\Lambda \models$ execution latency

Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (a plaintext) to $\mathcal{T}$, 2) $\mathcal{T}$ performs the computation detailed (i.e., it encrypts the plaintext using the cipher key), then 3) $\mathcal{T}$ provides some output (a ciphertext) to $\mathcal{E}$. During such an interaction, $\mathcal{E}$ is *also* able to observe (i.e., measure) the computation performed by $\mathcal{T}$, so attempts to exploit this fact. Note that the ability to observe execution latency is plausible, because $\mathcal{E}$ can simply record how long $\mathcal{T}$ takes to respond with the output associated with a given input (noting the potential for experimental noise while doing so).

3.5.2. Material

`${ARCHIVE}/des_cache/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- m, a 1-block DES plaintext (represented as a length-prefixed, hexadecimal octet string),

from `stdin` using one field per line, and then write the following output (one field per line)

- Λ, an execution latency measured in clock cycles (represented as a decimal integer string), and
- c, a 1-block DES ciphertext (represented as a length-prefixed, hexadecimal octet string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error. Note that:

- $\mathcal{T}$ uses a software implementation of DES, based on that written by Richard Outerbridge [12, Part V].
- DES has a 64-bit block length, and, although it notionally accepts a 64-bit cipher key, the *effective* cipher key length is in fact 56 bits: the remaining 8 bits support error detection via a parity code. You can ignore the parity bits, meaning *many* 64-bit candidate cipher keys will be valid when used in place of $\hat{k}$ (since only the 56 bits actually used by DES are relevant).
- The cited source code uses a set of eight separate S-box look-up tables. Each look-up table has 64 entries of type **`unsigned long`** (i.e., a 64-bit unsigned integer), so will consume 512 bytes of memory. Doing so improves efficiency: a look-up table captures the function of an associated S-box *plus* the (inefficient) P- and E-permutations. However, the implementation actually used by $\mathcal{T}$ differs slightly. Given each of the look-up table entries holds 32 bits of content, the type of entries has been altered to **`unsigned int`**. Once loaded, an entry is simply cast into **`unsigned long`**, roughly halving the memory footprint with no impact on efficiency. Given the 32-byte L1 cache line size, and that each look-up table is aligned to a 32-byte boundary, each cache line can therefore accommodate eight full 32-bit entries.
- To perform the role described, the implementation
  1. invokes the `deskey` function to pre-compute the round keys from a given cipher key, then
  2. invokes the `des` function to perform an encryption operation.

Within the `des` function, `scrunch` and `unscrun` are used to convert a sequence of eight 8-bit bytes to and from a 64-bit integer (held as two 32-bit halves) before and after `desfunc` performs the encryption operation itself.

- You can assume that when an encryption operation is performed, i.e., when the `des` function is invoked, the cache contains *no* S-box content: any cache-hits or -misses during encryption therefore depend only on $\hat{k}$ and m, as does the execution latency.

`${ARCHIVE}/des_cache/${USER}.R` In many side-channel and fault attacks, the attacker is able to perform an initial profiling or calibration phase: a common rationale for doing so is to support selection or fine-tuning of parameters for a subsequent attack. With this in mind, a simulated replica $\mathcal{R}$ of the attack target is also provided. $\mathcal{R}$ is identical to $\mathcal{T}$, bar the fact it reads the following input (one field per line)

- $\bar{m}$, a 1-block DES plaintext (represented as a length-prefixed, hexadecimal octet string), and
- $\bar{k}$, a DES cipher key (represented as a length-prefixed, hexadecimal octet string),

from `stdin`. In contrast to $\mathcal{T}$, this means $\mathcal{R}$ will use the *chosen* DES cipher key $\bar{k}$ to encrypt the chosen DES plaintext $\bar{m}$.

3.5.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{k}$. When executed using a command of the form

   ```
   ./attack ${USER}.T
   ```

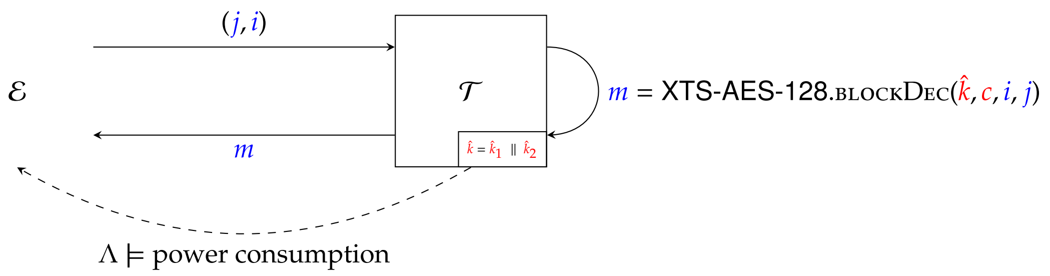
   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.

2. Answer the written, exam-style questions in `${ARCHIVE}/des_cache/${USER}.exam`.

3.6. Challenge #6: an attack based on power consumption

3.6.1. Background

Imagine you are tasked with attacking a device, denoted $\mathcal{T}$. The device is a Self Encrypting Disk (SED): the underlying disk only ever stores encrypted data, which is encrypted (before writing) and decrypted (after reading) using XTS-AES-128 [3]. The decryption of data uses an XTS-AES-128 key embedded in $\mathcal{T}$, which is initially derived from a user-selected password p, i.e., $\hat{k} = f(p)$ for a derivation function f. Each time the SED is powered-on, a password attempt/guess g is entered and a check whether $\hat{k} \stackrel{?}{=} f(g)$ then enforced; if the check fails, then the SED will refuses all further attempts to interact with it. $\mathcal{E}$ is lucky, and gets one-time access to $\mathcal{T}$ for a limited period of time while it is already powered-on (implying the the password check has already been performed). An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (some key material), as follows:


(j, i)
$\mathcal{E}$
$\mathcal{T}$
$m = \text{XTS-AES-128.BlockDec}(\hat{k}, c, i, j)$
m
$\hat{k} = \hat{k}_1 \parallel \hat{k}_2$
$\Lambda \models$ power consumption

Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (a block and sector address) to $\mathcal{T}$, 2) $\mathcal{T}$ performs the computation detailed (i.e., it reads and decrypts the addressed ciphertext block), then 3) $\mathcal{T}$ provides some output (a plaintext) to $\mathcal{E}$. During such an interaction, $\mathcal{E}$ is *also* able to observe (i.e., measure) the computation performed by $\mathcal{T}$, so attempts to exploit this fact. Note that $\mathcal{E}$ provides a power supply and clock signal to $\mathcal{T}$; direct access to the power supply means it is plausible for the former to observe power consumption of the later. Doing so will yield at least one sample per instruction executed, due to the sample rate of the oscilloscope used relative to the clock frequency demanded by and supplied to $\mathcal{T}$.

3.6.2. Material

`${ARCHIVE}/aes_power/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- j, a block address (represented as a decimal integer string), and
- i, a sector address, i.e., a 1-block XTS-AES-128 tweak (represented as a length-prefixed, hexadecimal octet string),

from `stdin` using one field per line, and then write the following output (one field per line)

- Λ, a power consumption trace (the format and semantics of which are explained further below), and
- m, a 1-block XTS-AES-128 plaintext (represented as a length-prefixed, hexadecimal octet string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error. Note that:

- $\mathcal{T}$ uses an AES-128 implementation, which implies 128-bit block and cipher key lengths; following the notation in [1, Figure 5], note that $Nb = 4$ and $Nr = 10$ means a (4×4)-element state matrix is used in a total of 11 rounds.
- More concretely, it is an 8-bit, memory-constrained implementation with the S-box held as a 256 B look-up table in memory. In line with the goal of minimising the memory footprint, the round keys are not pre-computed: each encryption takes the cipher key and evolves it forward, step-by-step, to form successive round keys for use during key addition. However, decryption is more complicated because the round keys must be used in the reverse order. One possibility would be to store the last round key as well as the cipher key, and evolve this backward through each round key. To avoid the increased demand on (secure, non-volatile) storage, $\mathcal{T}$ instead opts to first evolve the cipher key forward into the last round key, *then* evolve this backward through each round key.
- The underlying disk used by $\mathcal{T}$ has a 64 GiB capacity and uses Advanced Format (AF) 4 KiB sectors. As such, each of the 16777216 sectors contains 256 blocks of AES-128 ciphertext: valid use of $\mathcal{T}$ demands the block address satisfies $0 \leq j < 256$, and the sector address satisfies $0 \leq i < 16777216$ (which, since this implies only $\mathsf{LSB}_{24}(i)$ is relevant, is equivalent to saying $\mathsf{MSB}_{104}(i) = 0$). $\mathcal{E}$ can use *any* j and i as input to $\mathcal{T}$: if either is invalid, it uses a null ciphertext (i.e., 16 zero bytes, versus a ciphertext read from the disk itself) but otherwise functions as normal.
- $\mathcal{T}$ pre-computes the 256 possibilities of α^j, which are stored in a 4096 B look-up table. As a result, it is reasonable to assume AES-128 invocations dominate the computation it will perform for each interaction.
- Each power consumption trace Λ is a comma-separated line of the form

  $$l, \lambda_0, \lambda_1, \ldots, \lambda_{l-1}$$

  where

  - l (represented as a decimal integer string), specifies the trace length, and
  - a given λ_i (represented as a decimal integer string), specifies the i-th of l power consumption samples, each constituting an 8-bit, unsigned decimal integer: in short, this means $0 \leq \lambda_i < 256$ for $0 \leq i < l$.

`${ARCHIVE}/aes_power/${USER}.R` In many side-channel and fault attacks, the attacker is able to perform an initial profiling or calibration phase: a common rationale for doing so is to support selection or fine-tuning of parameters for a subsequent attack. With this in mind, a simulated replica $\mathcal{R}$ of the attack target is also provided. $\mathcal{R}$ is identical to $\mathcal{T}$, bar the fact it reads the following input (one field per line)

- $\bar{j}$, a block address (represented as a decimal integer string),
- $\bar{i}$, a sector address, i.e., a 1-block XTS-AES-128 tweak (represented as a length-prefixed, hexadecimal octet string),
- $\bar{c}$, a 1-block XTS-AES-128 ciphertext (represented as a length-prefixed, hexadecimal octet string), and
- $\bar{k}$, an XTS-AES-128 key (represented as a length-prefixed, hexadecimal octet string),

from `stdin`. In contrast to $\mathcal{T}$, this means $\mathcal{R}$ will use the *chosen* key $\bar{k}$ to decrypt the *chosen* ciphertext $\bar{c}$.

3.6.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{k}$. When executed using a command of the form

   ```
   ./attack ${USER}.T
   ```

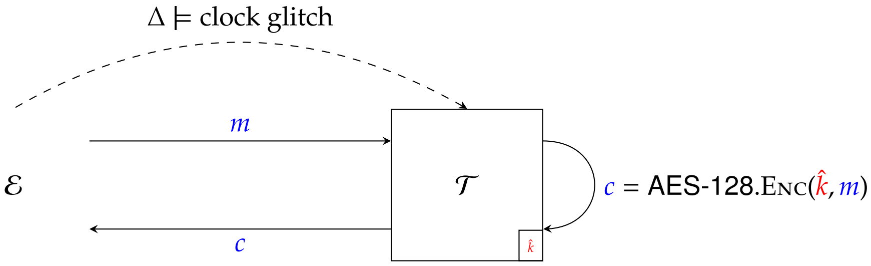
   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.

2. Answer the written, exam-style questions in `${ARCHIVE}/aes_power/${USER}.exam`.

3.7. Challenge #7: an attack based on an injected fault

3.7.1. Background

Imagine you are tasked with attacking a device, denoted $\mathcal{T}$, which houses an 8-bit Intel 8051 micro-processor. More specifically, $\mathcal{T}$ represents an ISO/IEC 7816 compliant contact-based smart-card used within larger devices as a cryptographic co-processor module: using a standardised protocol (or API), it supports secure key generation and storage plus off-load of some cryptographic operations. An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (a cipher key), as follows:



$\Delta \models$ clock glitch
m
$\mathcal{E}$
$\mathcal{T}$
$c = \text{AES-128.Enc}(\hat{k}, m)$
c
$\hat{k}$

Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (a plaintext) to $\mathcal{T}$, 2) $\mathcal{T}$ performs the computation detailed (i.e., it encrypts the plaintext using the cipher key), then 3) $\mathcal{T}$ provides some output (a ciphertext) to $\mathcal{E}$. During such an interaction, $\mathcal{E}$ is *also* able to induce faults during the computation performed by $\mathcal{T}$, so attempts to exploit this fact. Note that $\mathcal{E}$ provides a power supply and clock signal to $\mathcal{T}$; direct access to the clock signal means it is plausible for the former to influence the latter, e.g., causing a malfunction by supplying an irregular clock signal (or 'glitch'). One malfunction, or fault, can be induced per interaction with $\mathcal{T}$: it will act to randomise one element of the state matrix used by AES-128, at a chosen point during the associated encryption.

3.7.2. Material

`${ARCHIVE}/aes_fault/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- Δ, a fault specification (the format and semantics of which are explained further below), and
- m, a 1-block AES-128 plaintext (represented as a length-prefixed, hexadecimal octet string),

from `stdin` using one field per line, and then write the following output (one field per line)

- c, a 1-block AES-128 ciphertext (represented as a length-prefixed, hexadecimal octet string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error. Note that:

- $\mathcal{T}$ uses an AES-128 implementation, which clearly implies 128-bit block and cipher key lengths; following the notation in [1, Figure 5], the fact that $Nb = 4$ and $Nr = 10$ means a (4×4)-element state matrix is used in a total of 11 rounds where
  - the 0-th round consists of the `AddRoundKey` round function alone,
  - the 1-st to 9-th rounds consist of the `SubBytes`, `ShiftRows`, `MixColumns`, then `AddRoundKey` round functions, and

  - the 10-th round consists of the `SubBytes`, `ShiftRows`, then `AddRoundKey` round functions.

- More concretely, it is an 8-bit, memory-constrained implementation with the S-box held as a 256 B look-up table in memory. In line with the goal of minimising the memory footprint, the round keys are not pre-computed: each encryption takes the cipher key and evolves it forward, step-by-step, to form successive round keys for use during key addition.

- The fault specification is *either* a comma-separated line of the form

$$r, f, p, i, j$$

  where

  1. r (represented as a decimal integer string) specifies the round in which the fault occurs, implying $0 \leq r < 11$,
  2. f (represented as a decimal integer string) specifies the round function in which the fault occurs via

$$f = \begin{cases} 0 & \text{for a fault in the } \texttt{AddRoundKey} \text{ round function} \\ 1 & \text{for a fault in the } \texttt{SubBytes} \text{ round function} \\ 2 & \text{for a fault in the } \texttt{ShiftRows} \text{ round function} \\ 3 & \text{for a fault in the } \texttt{MixColumns} \text{ round function} \end{cases}$$

  3. p (represented as a decimal integer string) specifies whether the fault occurs before or after execution of the round function via

$$p = \begin{cases} 0 & \text{for a fault before the round function} \\ 1 & \text{for a fault after the round function} \end{cases}$$

     and

  4. i and j (both represented as decimal integer strings), specify the row and column of the state matrix which the fault occurs in, implying $0 \leq i, j < 4$

  or a blank line. In the latter case, *no* fault will be induced; this will obviously yield the correctly encrypted ciphertext for a given plaintext.

- The fault randomises one element of the state matrix: this can be captured by writing the resulting value as $s_{i,j} \oplus \delta$ where δ is a (random) difference introduced by the fault. Keep in mind that faults such that $\delta = 0$ *are* possible, and, since $s_{i,j} \oplus \delta = s_{i,j} \oplus 0 = s_{i,j}$, this effectively means no fault will be induced in such cases.

3.7.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{k}$. When executed using a command of the form

```
./attack ${USER}.T
```

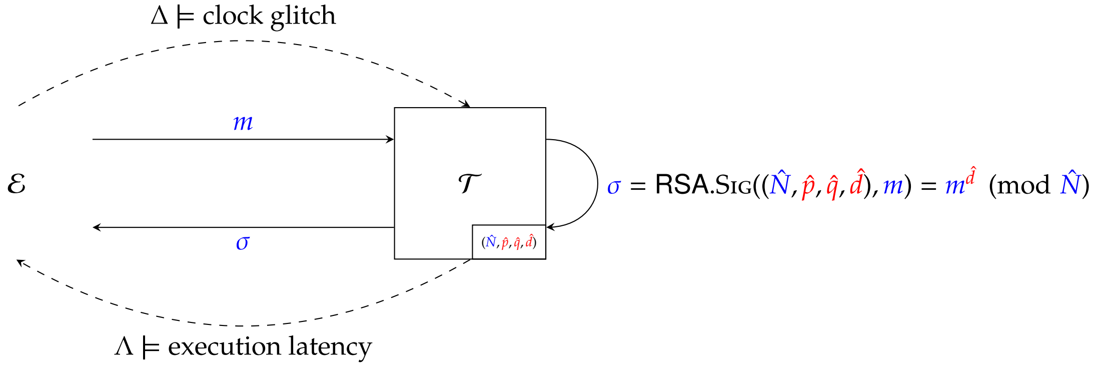
   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.

2. Answer the written, exam-style questions in `${ARCHIVE}/aes_fault/${USER}.exam`.

3.8. Challenge #8: an attack based on an injected fault

3.8.1. Background

Imagine you are tasked with attacking a device, denoted $\mathcal{T}$, which houses an 8-bit Intel 8051 micro-processor. More specifically, $\mathcal{T}$ represents an ISO/IEC 7816 compliant contact-based smart-card used within larger devices as a cryptographic co-processor module: using a standardised protocol (or API), it supports secure key generation and storage plus off-load of some cryptographic operations. An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (a private key), as follows:


$\Delta \models$ clock glitch
m
$\mathcal{E}$
$\mathcal{T}$
$\sigma = \text{RSA.Sig}((\hat{N}, \hat{p}, \hat{q}, \hat{d}), m) = m^{\hat{d}} \pmod{\hat{N}}$
$(\hat{N}, \hat{p}, \hat{q}, \hat{d})$
σ
$\Lambda \models$ execution latency

Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (a message) to $\mathcal{T}$, 2) $\mathcal{T}$ performs the computation detailed (i.e. it signs the message using the private key), then 3) $\mathcal{T}$ provides some output (a signature) to $\mathcal{E}$. During such an interaction, $\mathcal{E}$ is *also* able to observe (i.e., measure) and influence the computation performed by $\mathcal{T}$, so attempts to exploit this fact. Note that $\mathcal{E}$ provides a power supply and clock signal to $\mathcal{T}$; direct access to the clock signal means it is plausible for the former to influence the latter, e.g., causing a malfunction by supplying an irregular clock signal (or 'glitch').

3.8.2. Material

`${ARCHIVE}/rsa_fault/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- Δ a fault specification (represented as a decimal integer string, the format and semantics of which are explained further below),
- m, a message (represented as a hexadecimal integer string),

from `stdin` using one field per line, and then write the following output (one field per line)

- Λ, an execution latency measured in clock cycles (represented as a decimal integer string), and
- σ, an RSA signature on m (represented as a hexadecimal integer string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error. Note that:

- Internally, $\mathcal{T}$ uses a co-processor to support multi-precision integer arithmetic: by storing operands in an internal, special-purpose register file, it is able to perform single-cycle computation of modular inversion (i.e., $1/x \pmod{n}$), negation (i.e., $-x \pmod{n}$), addition (i.e., $x + y \pmod{n}$), subtraction (i.e., $x - y \pmod{n}$), and multiplication (i.e., $x \cdot y \pmod{n}$).
- $\mathcal{T}$ uses the co-processor to realise a CRT-based [11] implementation of RSA decryption, using a Gauss-based recombination step.
- $\mathcal{E}$ can induce a fault, via a 'glitch' in the clock signal supplied to $\mathcal{T}$; the fault is controlled using Δ, in the sense that

  $$\begin{array}{lcl} \Delta < 0 & \Rightarrow & \text{no fault induced} \\ \Delta \geq 0 & \Rightarrow & \text{fault induced in clock cycle } \Delta \end{array}$$

  Through analysis of the fault model, it is known that inducing a fault will impact the co-processor ALU; if a fault is induced in clock cycle Δ, computation by (or output of) the ALU in that clock cycle is corrupted (or randomised).

`${ARCHIVE}/rsa_fault/${USER}.conf` This file represents a set of attack parameters, including everything (e.g., all public values) that $\mathcal{E}$ has access to by default. More specifically, it contains

- $\hat{N}$, an RSA modulus (represented as a hexadecimal integer string), and
- $\hat{e}$, an RSA public exponent (represented as a hexadecimal integer string), such that $\hat{e} \cdot \hat{d} \equiv 1 \pmod{\Phi(\hat{N})}$,

using one field per line. Note that $(\hat{N}, \hat{e})$ is a (known) RSA public key associated with $(\hat{N}, \hat{d})$, i.e., the (unknown) RSA private key.

3.8.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{d}$. When executed using a command of the form

   ```
   ./attack ${USER}.T ${USER}.conf
   ```

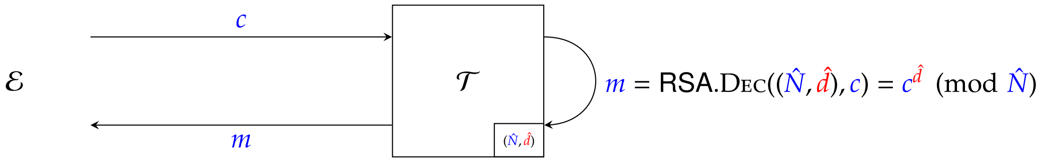
   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.

2. Answer the written, exam-style questions in `${ARCHIVE}/rsa_fault/${USER}.exam`.

3.9. Challenge #9: an attack based on an inherent fault

3.9.1. Background

Imagine you are tasked with attacking a server, denoted $\mathcal{T}$, which houses a 'cloned' 64-bit MIPS64 micro-processor: the reduced cost (versus a legitimate alternative) made it an attractive option for the owner, who was forced to economise when upgrading an old data center. $\mathcal{T}$ is used by several front-line e-commerce servers, which offload computation relating to the TLS handshake. More specifically, $\mathcal{T}$ is used to compute the RSA decryption required by RSA-based key exchange. An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (a private key), as follows:


c
$\mathcal{E}$
$\mathcal{T}$
$m = \text{RSA.Dec}((\hat{N}, \hat{d}), c) = c^{\hat{d}} \pmod{\hat{N}}$
m
$(\hat{N}, \hat{d})$

Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (a ciphertext) to $\mathcal{T}$, 2) $\mathcal{T}$ performs the computation detailed (i.e., it decrypts the ciphertext using the private key), then 3) $\mathcal{T}$ provides some output (a plaintext) to $\mathcal{E}$. Unfortunately, however, the micro-processor used by $\mathcal{T}$ has a defect: hardware within the ALU which supports (64×64)-bit integer multiplication has a (deterministic) bug, which means the output (i.e., the product $x \cdot y$) is incorrect for certain input (i.e., certain x and/or y).

3.9.2. Material

`${ARCHIVE}/rsa_bug/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- c, an RSA ciphertext (represented as a hexadecimal integer string),

from `stdin` using one field per line, and then write the following output (one field per line)

- m, an RSA plaintext (represented as a hexadecimal integer string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error. Note that:

- Because $\mathcal{T}$ houses a 64-bit micro-processor, it employs a base-2^{64} representation of multi-precision integers throughout. Put another way, since $w = 64$, it selects $b = 2^w = 2^{64}$.

- Rather than a CRT-based approach, $\mathcal{T}$ uses a left-to-right binary exponentiation [5, Section 2.1] algorithm to compute $m = c^{\hat{d}} \pmod{\hat{N}}$. Montgomery multiplication [10] is harnessed to improve efficiency; following notation in [9], a CIOS-based [9, Section 5] approach is used to integrate[6] an integer multiplication with a subsequent Montgomery reduction.

- Following the notation in [5], $\hat{d}$ is assumed to have $l + 1$ bits such that $\hat{d}_l = 1$ for some l. However, it has been (artificially) selected so $0 \leq \hat{d} < 2^{64}$ to limit the duration of an attack: you should not rely on this fact, implying your attack could succeed for *any* $\hat{d}$ if afforded enough time.

[6] MonPro [9, Section 2] perhaps offers a more direct way to explain this: steps 1 and 2 are the integer multiplication and Montgomery reduction written both separately and abstractly, whereas $\mathcal{T}$ realises them concretely using the integrated CIOS algorithm.

- Consider the base-2^{16} expansion of an integer t, i.e., $t = \langle t_0, t_1, \ldots, t_{n-1} \rangle_{(2^{16})}$. We say that t is γ-poisoned iff. there exists an i such that $0 \leq i < n$ and $t_i = \gamma$, i.e., at least one of the limbs is equal to γ. Recall that the integer multiplier used by $\mathcal{T}$ is defective: for

  $$\begin{array}{rcl} \alpha & = & DEAD_{(16)} \\ \beta & = & BEEF_{(16)} \end{array}$$

  it will incorrectly compute the product $x \cdot y$ iff. x is α-poisoned and y is β-poisoned.

`${ARCHIVE}/rsa_bug/${USER}.conf` This file represents a set of attack parameters, including everything (e.g., all public values) that $\mathcal{E}$ has access to by default. More specifically, it contains

- $\hat{N}$, an RSA modulus (represented as a hexadecimal integer string), and
- $\hat{e}$, an RSA public exponent (represented as a hexadecimal integer string), such that $\hat{e} \cdot \hat{d} \equiv 1 \pmod{\Phi(\hat{N})}$,

using one field per line. Note that $(\hat{N}, \hat{e})$ is a (known) RSA public key associated with $(\hat{N}, \hat{d})$, i.e., the (unknown) RSA private key.

3.9.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{d}$. When executed using a command of the form

   ```
   ./attack ${USER}.T ${USER}.conf
   ```

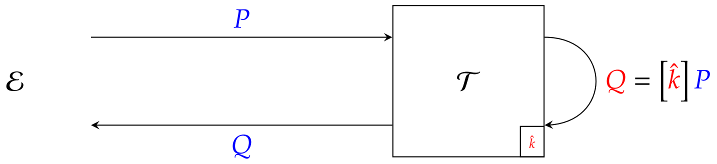
   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.

2. Answer the written, exam-style questions in `${ARCHIVE}/rsa_bug/${USER}.exam`.

3.10. Challenge #10: an attack based on a erroneous input

3.10.1. Background

Imagine you own a cryptographic accelerator, denoted $\mathcal{T}$: it is used as a low-cost HSM in your data center, supporting secure key generation and storage plus off-load of various cryptographic operations via a standardised protocol (or API). An attacker $\mathcal{E}$ interacts with $\mathcal{T}$, which stores the embedded, security-critical data (a scalar), as follows:



Each interaction can be described as a series of potentially adaptive steps, where 1) $\mathcal{E}$ provides some input (an input point) to $\mathcal{T}$, 2) $\mathcal{T}$ performs the computation detailed (i.e., multiplies the input point by the scalar), then 3) $\mathcal{T}$ provides some output (an output point) to $\mathcal{E}$. Although it currently functions correctly, an electrical fault demands that $\mathcal{T}$ is replaced. However, doing so presents a serious problem: the API cannot be used to export $\hat{k}$, so a replacement will have to be generated and used thereafter. This means compatibility with existing data sets will be lost, *unless* $\hat{k}$ itself can be recovered by some other means.

3.10.2. Material

`${ARCHIVE}/ecc_invalid/${USER}.T` This executable simulates interaction with the attack target $\mathcal{T}$. When executed it will read the following input

- P_x, the x-coordinate of elliptic curve point $P = (P_x, P_y)$ (represented as a hexadecimal integer string), and
- P_y, the y-coordinate of elliptic curve point $P = (P_x, P_y)$ (represented as a hexadecimal integer string),

from `stdin` using one field per line, and then write the following output (one field per line)

- Q_x, the x-coordinate of elliptic curve point $Q = (Q_x, Q_y) = \left[\hat{k}\right] P$ (represented as a hexadecimal integer string), and
- Q_y, the y-coordinate of elliptic curve point $Q = (Q_x, Q_y) = \left[\hat{k}\right] P$ (represented as a hexadecimal integer string),

to `stdout` using one field per line. Execution continues this way, i.e., repeatedly reading input then writing output, until terminated by the user or some form of internal error. Note that:

- $\mathcal{T}$ uses an left-to-right binary scalar multiplication (i.e., an additive analogue of [5, Section 2.1]) to compute $Q = \left[\hat{k}\right] P$. In doing so, it assumes use of the NIST-P-256 elliptic curve

  $$E(\mathbb{F}_p) : y^2 = x^3 + a_4 x + a_6$$

  for standardised [4] domain parameters p, a_4, and a_6.
- More concretely, $\mathcal{T}$ uses OpenSSL to support arithmetic on the elliptic curve. You can browse the associated source code at

  `http://github.com/openssl`

  noting that the choice outlined above implies use of `ec_GFp_simple_add` and `ec_GFp_simple_dbl` to realise the (additive) group operation, i.e., to compute either a point addition or doubling, within the scalar multiplication.
- Crucially, $\mathcal{T}$ applies *no* checks to either the input or output points (e.g., to check whether $P \in E(\mathbb{F}_p)$ and reject P if not).
- At least from a Mathematical point of view, there is no valid affine representation of the point at infinity. If computation of $\left[\hat{k}\right] P$ does yield the point at infinity, $\mathcal{T}$ therefore signals this fact by producing

  $$Q = (Q_x, Q_y) = (0, 0)$$

  as output (rather than some special-purpose mechanism, e.g., an error or an alternative output format).

3.10.3. Tasks

1. Write a program that simulates the attacker $\mathcal{E}$, by interacting with and attacking the simulated attack target, i.e., realising the goal of recovering the target material $\hat{k}$. When executed using a command of the form

   ```
   ./attack ${USER}.T
   ```

   the attack should be invoked on the named simulated attack target. Use `stdout` to record any intermediate output you deem relevant, followed finally by two lines which clearly detail a) the target material recovered, plus b) the total number of interactions with the simulated attack target that were required to do so.
2. Answer the written, exam-style questions in `${ARCHIVE}/ecc_invalid/${USER}.exam`.

References

[1] *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 197. 2001. URL: http://csrc.nist.gov (see pp. 6, 11, 12).

[2] M. Bellare and P. Rogaway. “Optimal asymmetric encryption”. In: *Advances in Cryptology (EUROCRYPT)*. LNCS 950. Springer-Verlag, 1994, pp. 92–111 (see p. 4).

[3] *Cryptographic Protection of Data on Block-Oriented Storage Devices*. Institute of Electrical and Electronics Engineers (IEEE) Standard 1619-2007. 2007. URL: http://standards.ieee.org (see p. 10).

[4] *Digital Signature Standard (DSS)*. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 186-2. 2000. URL: http://csrc.nist.gov (see p. 17).

[5] D.M. Gordon. “A Survey of Fast Exponentiation Methods”. In: *Journal of Algorithms* 27 (1998), pp. 129–146 (see pp. 8, 15, 17).

[6] J. Jonsson and B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specification Version 2.1*. Internet Engineering Task Force (IETF) Request for Comments (RFC) 3447. 2003. URL: http://tools.ietf.org/html/rfc3447 (see p. 4).

[7] B. Kaliski. *PKCS #7: Cryptographic Message Syntax Version* 1.5. Internet Engineering Task Force (IETF) Request for Comments (RFC) 2315. 1998. URL: http://tools.ietf.org/html/rfc2315 (see p. 3).

[8] E. Käsper and P. Schwabe. "Faster and Timing-attack Resistant AES-GCM". In: *Cryptographic Hardware and Embedded Systems (CHES)*. LNCS 5747. Springer-Verlag, 2009, pp. 1–17 (see p. 3).

[9] Ç.K. Koç, T. Acar, and B.S. Kaliski. "Analyzing and comparing Montgomery multiplication algorithms". In: *IEEE Micro* 16.3 (1996), pp. 26–33 (see pp. 8, 15).

[10] P.L. Montgomery. "Modular multiplication without trial division". In: *Mathematics of Computation* 44.170 (1985), pp. 519–521 (see pp. 8, 15).

[11] J-J. Quisquater and C. Couvreur. "Fast decipherment algorithm for RSA public-key cryptosystem". In: *IEE Electronics Letters* 18.21 (1982), pp. 905–907 (see p. 14).

[12] B. Schneier. *Applied Cryptography*. 2nd. John Wiley & Sons, 2006. URL: http://www.schneier.com/book-applied.html (see p. 9).

A Representation and conversion

1.1. Integers

Concept. An integer string (or literal) is written as a string of characters, each of which represents a digit; the set of possible digits, and the value being represented, depends on a base b. We read digits from right-to-left: the least-significant (resp. most-significant) digit is the right-most (resp. left-most) character within the string. As such, the 20-character string

$$\hat{x} = \texttt{09080706050403020100}$$

represents the integer value

$$\hat{x} \mapsto \begin{array}{lllllllllll}
\hat{x}_{19} \cdot b^{19} & + & \hat{x}_{18} \cdot b^{18} & + & \hat{x}_{17} \cdot b^{17} & + & \hat{x}_{16} \cdot b^{16} & + & \hat{x}_{15} \cdot b^{15} & + \\
\hat{x}_{14} \cdot b^{14} & + & \hat{x}_{13} \cdot b^{13} & + & \hat{x}_{12} \cdot b^{12} & + & \hat{x}_{11} \cdot b^{11} & + & \hat{x}_{10} \cdot b^{10} & + \\
\hat{x}_{9} \cdot b^{9} & + & \hat{x}_{8} \cdot b^{8} & + & \hat{x}_{7} \cdot b^{7} & + & \hat{x}_{6} \cdot b^{6} & + & \hat{x}_{5} \cdot b^{5} & + \\
\hat{x}_{4} \cdot b^{4} & + & \hat{x}_{3} \cdot b^{3} & + & \hat{x}_{2} \cdot b^{2} & + & \hat{x}_{1} \cdot b^{1} & + & \hat{x}_{0} \cdot b^{0} &
\end{array}$$

Of course the actual value depends on the base b in which we interpret the representation $\hat{x}$:

- for a decimal integer string $b = 10$, meaning

$$\begin{array}{lllllllllll}
\hat{x} & \mapsto & \hat{x}_{19} \cdot 10^{19} + \hat{x}_{18} \cdot 10^{18} + \hat{x}_{17} \cdot 10^{17} + \hat{x}_{16} \cdot 10^{16} + \hat{x}_{15} \cdot 10^{15} + \\
& & \hat{x}_{14} \cdot 10^{14} + \hat{x}_{13} \cdot 10^{13} + \hat{x}_{12} \cdot 10^{12} + \hat{x}_{11} \cdot 10^{11} + \hat{x}_{10} \cdot 10^{10} + \\
& & \hat{x}_{9} \cdot 10^{9} + \hat{x}_{8} \cdot 10^{8} + \hat{x}_{7} \cdot 10^{7} + \hat{x}_{6} \cdot 10^{6} + \hat{x}_{5} \cdot 10^{5} + \\
& & \hat{x}_{4} \cdot 10^{4} + \hat{x}_{3} \cdot 10^{3} + \hat{x}_{2} \cdot 10^{2} + \hat{x}_{1} \cdot 10^{1} + \hat{x}_{0} \cdot 10^{0} \\[1em]
& \mapsto & 0_{(10)} \cdot 10^{19} + 9_{(10)} \cdot 10^{18} + 0_{(10)} \cdot 10^{17} + 8_{(10)} \cdot 10^{16} + 0_{(10)} \cdot 10^{15} + \\
& & 7_{(10)} \cdot 10^{14} + 0_{(10)} \cdot 10^{13} + 6_{(10)} \cdot 10^{12} + 0_{(10)} \cdot 10^{11} + 5_{(10)} \cdot 10^{10} + \\
& & 0_{(10)} \cdot 10^{9} + 4_{(10)} \cdot 10^{8} + 0_{(10)} \cdot 10^{7} + 3_{(10)} \cdot 10^{6} + 0_{(10)} \cdot 10^{5} + \\
& & 2_{(10)} \cdot 10^{4} + 0_{(10)} \cdot 10^{3} + 1_{(10)} \cdot 10^{2} + 0_{(10)} \cdot 10^{1} + 0_{(10)} \cdot 10^{0} \\[1em]
& \mapsto & 9080706050403020100_{(10)}
\end{array}$$

  whereas

- for a hexadecimal integer string $b = 16$, meaning

$$\begin{array}{lllllllllll}
\hat{x} & \mapsto & \hat{x}_{19} \cdot 16^{19} + \hat{x}_{18} \cdot 16^{18} + \hat{x}_{17} \cdot 16^{17} + \hat{x}_{16} \cdot 16^{16} + \hat{x}_{15} \cdot 16^{15} + \\
& & \hat{x}_{14} \cdot 16^{14} + \hat{x}_{13} \cdot 16^{13} + \hat{x}_{12} \cdot 16^{12} + \hat{x}_{11} \cdot 16^{11} + \hat{x}_{10} \cdot 16^{10} + \\
& & \hat{x}_{9} \cdot 16^{9} + \hat{x}_{8} \cdot 16^{8} + \hat{x}_{7} \cdot 16^{7} + \hat{x}_{6} \cdot 16^{6} + \hat{x}_{5} \cdot 16^{5} + \\
& & \hat{x}_{4} \cdot 16^{4} + \hat{x}_{3} \cdot 16^{3} + \hat{x}_{2} \cdot 16^{2} + \hat{x}_{1} \cdot 16^{1} + \hat{x}_{0} \cdot 16^{0} \\[1em]
& \mapsto & 0_{(16)} \cdot 16^{19} + 9_{(16)} \cdot 16^{18} + 0_{(16)} \cdot 16^{17} + 8_{(16)} \cdot 16^{16} + 0_{(16)} \cdot 16^{15} + \\
& & 7_{(16)} \cdot 16^{14} + 0_{(16)} \cdot 16^{13} + 6_{(16)} \cdot 16^{12} + 0_{(16)} \cdot 16^{11} + 5_{(16)} \cdot 16^{10} + \\
& & 0_{(16)} \cdot 16^{9} + 4_{(16)} \cdot 16^{8} + 0_{(16)} \cdot 16^{7} + 3_{(16)} \cdot 16^{6} + 0_{(16)} \cdot 16^{5} + \\
& & 2_{(16)} \cdot 16^{4} + 0_{(16)} \cdot 16^{3} + 1_{(16)} \cdot 16^{2} + 0_{(16)} \cdot 16^{1} + 0_{(16)} \cdot 16^{0} \\[1em]
& \mapsto & 42649378395939397566720_{(10)}
\end{array}$$

Example. Consider the following Python 3.x program

```
t_0 = '09080706050403020100'

t_1 = int( t_0, 10 )
t_2 = int( t_0, 16 )

t_3 = ( '{0:d}'.format( t_2 ) )
t_4 = ( '{0:X}'.format( t_2 ) )

t_5 = ( '{0:X}'.format( t_2 ) ).zfill( 20 )

print( 'type( t_0 ) = {0!s:14s} t_0 = {1!s}'.format( type( t_0 ), t_0 ) )
print( 'type( t_1 ) = {0!s:14s} t_1 = {1!s}'.format( type( t_1 ), t_1 ) )
print( 'type( t_2 ) = {0!s:14s} t_2 = {1!s}'.format( type( t_2 ), t_2 ) )
print( 'type( t_3 ) = {0!s:14s} t_3 = {1!s}'.format( type( t_3 ), t_3 ) )
print( 'type( t_4 ) = {0!s:14s} t_4 = {1!s}'.format( type( t_4 ), t_4 ) )
print( 'type( t_5 ) = {0!s:14s} t_5 = {1!s}'.format( type( t_5 ), t_5 ) )
```

which, when executed, produces

```
type( t_0 ) = <class 'str'>  t_0 = 09080706050403020100
type( t_1 ) = <class 'int'>  t_1 = 9080706050403020100
type( t_2 ) = <class 'int'>  t_2 = 42649378395939397566720
type( t_3 ) = <class 'str'>  t_3 = 42649378395939397566720
type( t_4 ) = <class 'str'>  t_4 = 9080706050403020100
type( t_5 ) = <class 'str'>  t_5 = 09080706050403020100
```

as output. This is intended to illustrate that

- `t_0` is an integer string (i.e., a sequence of characters),
- `t_1` and `t_2` are conversions of `t_0` into integers, using decimal and hexadecimal respectively, and
- `t_3` and `t_4` are conversions of `t_2` into strings (i.e., a sequence of characters), using decimal and hexadecimal respectively.

Note that `t_0` and `t_4` do not match: the conversion ignored the left-most zero character, because it is not significant wrt. the associated integer value. If/when this issue is problematic, it can be resolved by using the `zfill` function to left-fill the string (with zero characters) so it is of the required length. `t_0` and `t_5` do match, for example, because the latter has been filled to ensure an overall length of 20 characters.

1.2. Octet strings

Concept. The term octet[7] is normally used as a synonym for byte, most often within the context of communication (and computer networks). Using octet is arguably more precise than byte, in that the former is *always* 8 bits whereas the latter *can*[8] differ. A string is a sequence of characters, and so, by analogy, an octet string[9] is a sequence of octets: ignoring some corner cases, it is reasonable to use the term "octet string" as a synonym for "byte sequence".

To represent a given byte sequence, we use what can be formally termed a (little-endian) length-prefixed, hexadecimal octet string. However, doing so requires some explanation: each element of that term relates to a property of the representation, where we define a) little-endian[10] to mean, if read left-to-right, the first octet represents the 0-th element of the source byte sequence and the last octet represents the $(n-1)$-st element of the source byte sequence, b) length-prefixed[11] to mean n, the length of the source byte sequence, is prepended to the octet string as a single 8-bit[12] length or "header" octet, and c) hexadecimal[13] to mean each octet is represented by using 2 hexadecimal digits. Note that, confusingly, hexadecimal digits within each pair will be big-endian: if read left-to-right, the most-significant is first. For convenience, we assume the term octet string is a catch-all implying all such properties from here on.

An example likely makes all of the above *much* clearer: certainly there is nothing complex involved. Concretely, consider a 16-element byte sequence

```
uint8_t x[ 16 ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 }
```

defined using C. This would be represented as

$$\hat{x} \;=\; \texttt{10:000102030405060708090A0B0C0D0E0F}$$

using a colon to separate the length and value fields:

- the length (LHS of the colon) is the integer $n = 10_{(16)} = 16_{(10)}$, and
- the value (RHS of the colon) is the byte sequence $x = \langle 00_{(16)}, 01_{(16)}, \ldots, 0F_{(16)} \rangle = \langle 0_{(10)}, 1_{(10)}, \ldots, 15_{(10)} \rangle \equiv \texttt{x}$.

Note that the special-case of an empty byte sequence *is* valid: now starting with the 0-element byte sequence

```
uint8_t x[ 0 ] = { }
```

defined using C, setting n to 0 and x to an empty byte sequence yields the representation

$$\hat{x} \;=\; \texttt{00:}$$

vs. say an empty or null string, which, in contrast, is an invalid octet string.

[7] http://en.wikipedia.org/wiki/Octet_(computing)

[8] http://en.wikipedia.org/wiki/Byte, for example, details the fact that the term "byte" can be and has been interpreted to mean a) a group of n bits for $n < w$ (i.e., smaller than the word size), b) the data type used to represent characters, or c) the (smallest) unit of addressable data in memory: although POSIX mandates 8-bit bytes, for example, each of these cases permits an alternative definition.

[9] Note the octet string terminology stems from ASN.1 encoding; see, e.g., http://en.wikipedia.org/wiki/Abstract_Syntax_Notation_One.

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

[11] http://en.wikipedia.org/wiki/String_(computer_science)

[12] Although it simplifies the challenge associated with parsing such a representation, note that use of an 8-bit length implies an upper limit of 255 elements in the associated byte sequence.

[13] http://en.wikipedia.org/wiki/Hexadecimal

Example. Consider the following Python 3.*x* program

```
import binascii

def octetstr2bytes( x ) :
  t = x.split( ':' ) ; n = int( t[ 0 ], 16 ) ; x = binascii.a2b_hex( t[ 1 ] )

  if ( n != len( x ) ) :
    raise ValueError
  else :
    return x

def bytes2octetstr( x ) :
  n = '{0:02X}'.format( len( x ) ) ; x = binascii.b2a_hex( x ).decode( 'ascii' ).upper()

  return n + ':' + x

def bytes2seq( x ) :
  return            [ int( t ) for t in x ]

def seq2bytes( x ) :
  return bytearray( [ int( t ) for t in x ] )

t_0 = [ 0xDE, 0xAD, 0xBE, 0xEF ]
t_1 = bytes2octetstr( seq2bytes( t_0 ) )

print( 'type( t_0 ) = {0!s:14s} t_0 = {1!s}'.format( type( t_0 ), t_0 ) )
print( 'type( t_1 ) = {0!s:14s} t_1 = {1!s}'.format( type( t_1 ), t_1 ) )

t_2 = '04:DEADBEEF'
t_3 = bytes2seq( octetstr2bytes( t_2 ) )

print( 'type( t_2 ) = {0!s:14s} t_2 = {1!s}'.format( type( t_2 ), t_2 ) )
print( 'type( t_3 ) = {0!s:14s} t_3 = {1!s}'.format( type( t_3 ), t_3 ) )
```

which, when executed, produces

```
type( t_0 ) = <class 'list'>   t_0 = [222, 173, 190, 239]
type( t_1 ) = <class 'str'>    t_1 = 04:DEADBEEF
type( t_2 ) = <class 'str'>    t_2 = 04:DEADBEEF
type( t_3 ) = <class 'list'>   t_3 = [222, 173, 190, 239]
```

as output. This is intended to illustrate that

- If you start with a list (or sequence) `t_0`, then `seq2bytes` and `bytes2octetstr` will convert it first into a byte sequence then an octet string: the fact that `t_1` = `04:DEADBEEF` implies $n = 04_{(16)} = 04_{(10)}$ and $x = \langle DE_{(16)}, AD_{(16)}, BE_{(16)}, EF_{(16)} \rangle$, i.e., n = `len( t_0 )` and x = `t_0`, so, for example, x_1 = `t_0[ 1 ]` = $AD_{(16)}$.
- If you start with an octet string `t_2`, then `octetstr2bytes` and `bytes2seq` will convert it first into a byte sequence then a list (or sequence): the fact that `t_2` = `04:DEADBEEF` implies $n = 04_{(16)} = 04_{(10)}$ and $x = \langle DE_{(16)}, AD_{(16)}, BE_{(16)}, EF_{(16)} \rangle$, i.e., n = `len( t_3 )` and x = `t_3`, so, for example, x_0 = `t_3[ 0 ]` = $DE_{(16)}$.

Note that the conversions are self-consistent in the sense that `t_0` = `t_3`, or rather

`t_0` = `bytes2seq( octetstr2bytes( bytes2octetstr( seq2bytes( t_0 ) ) ) )`.