UNIVERSITY OF BRISTOL
DEPARTMENT OF COMPUTER SCIENCE
http://www.cs.bris.ac.uk

Applied Cryptology (COMS30048)

Assessed coursework assignment

AttackSW

Note that:

1. The deadline for this assignment is 06/05/22, with standard regulations enforced wrt. late submission.
2. This assignment represents 100 percent of marks available for COMS30048, and is assessed on an individual basis. Before you start work, make sure you are aware of and adhere to the various regulations[a] which govern this mode of assessment.
3. There are numerous support resources available, for example:
   - via the unit forum, where you get help and feedback via n-to-m, collective discussion,
   - via the lab. slot(s), where you get help and feedback via 1-to-1, personal discussion, or
   - via the lecturer(s) responsible for this assignment: although the above are preferable, you can make contact in-person or online (e.g., via email).

[a] http://www.bristol.ac.uk/academic-quality/assessment/codeonline.html

1 Introduction

There are two main categories of cryptanalytic attack (which can overlap to some extent): they either focus on the underlying design (or theory), or on the properties of a resulting implementation. This assignment is concerned with the second category. The overarching goal is to research then implement some *real* attacks, against *real* cryptosystems deployed in *real* applications.

2 Terms and conditions

- The assignment description may refer to `question.txt` or "the marksheet". Download this ASCII text file from

  `http://tinyurl.com/kua9nz2r/csdsp/cw/AttackSW/question.txt`

  then complete and include it in your submission: this is important, and failure to do so may result in a loss of marks. Keep in mind that *if* separate *assessment* units exist, they may have different assessment criteria and so marking scheme.

- Certain aspects of the assignment have a (potentially large) design space of possible approaches. Where there is some debate about the correct or "best" approach, the assignment demands *you* make an informed decision *yourself*: it is therefore not (purely) a programming exercise st. blindly implementing *an* approach will be enough. Such decisions should ideally be based on a reasoned argument formed via your *own* background research (vs. reliance on the teaching material alone), and clearly documented (e.g., using the marksheet).

- Where a choice is possible, which is not always the case, *you* can select the programming language used to implement a given aspect of the assignment. Viable examples include C and Python. Use of (correctly cited) third-party libraries *is* allowed for cases that do not conflict with the assessment ILOs. Viable examples include OpenSSL for C, and the `pycrypto` package for Python.

- Include a set of instructions that clearly describe how to compile and execute your solution. The ideal approach would be to a) submit (or alter) a `Makefile`, and/or b) use the marksheet to provide written instructions.

- To make the marking process easier, your solution should only write error messages to `stderr` (or equivalent). In addition, the only input read from `stdin` (resp. output written to `stdout`, or equivalents) should be that specified by the assignment description.

- You should submit your solution, into the correct component, via

  `http://www.ole.bris.ac.uk`

  Include any a) source code files, b) text or PDF files, (e.g., documentation) and c) auxiliary files (e.g., example output), either as required or that *you* feel are relevant. Keep in mind that *if* separate *teaching* and *assessment* units exist, you should submit via the latter *not* the former.

- To make the submission process easier, the recommended approach is to develop your solution within the *same* directory structure as the material provided. This will allow you to first create then submit a *single* archive (e.g., `solution.zip` using `zip`, or `solution.tar.gz` using `tar` and `gzip`) of your entire solution, rather than *multiple* separate files.

- Implementations produced as part of the assignment will be assessed using a platform equivalent to the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11). As such, they *must* compile, execute, and be thoroughly tested using both the operating system and development tool-chain versions available by default.

- Although you can *definitely* expect to receive partial marks for a partial solution, it will be marked *as is*. This means a) there will be no effort to enable either optional or commented functionality (e.g., by uncommenting it, or via specification of compile-time or run-time parameters), and b) submitting multiple variant solutions is strongly discouraged, but would be dealt with by considering the variant which yields the highest single mark.

3 Description

3.1 Material

Selected material, personalised on a per student basis, is provided for you to use. Assuming `${USER}` is used to represent your 7-digit UoB student number, download[1] and unarchive the file

`http://tinyurl.com/kua9nz2r/csdsp/cw/AttackSW/${USER}.tar.gz`

somewhere secure in your file system (e.g., in the `Private` sub-directory in your home directory): from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content.

3.2 Overview

The overarching goal of this assignment is that you *understand* selected implementation attack techniques. It approaches this goal by focusing on practical tasks, i.e., *implementing* (or mounting) and *analysing* said attacks. In more detail, it is structured as multiple stages, each of which involves a) the implementation of an attack against a simulated attack target set within a specific (somewhat contrived) context, and b) analysis of the attack, and general, wider context, via a set of written exam-style questions.

- 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[2] platform, and, even then, only once the file has appropriate permissions set using `chmod`.
- Interaction with a simulated attack target requires understanding the representation and conversion of integers and octet strings (which are essentially human-readable sequences of 8-bit bytes). Appendix A includes a detailed discussion of this issue.
- Although the functional correctness of your attack implementation is obviously crucial wrt. marks, various additional criteria also have an impact. The marksheet offers a high-level idea of the marking scheme, but it remains *your* task to consider then address more specific criteria. For example, your attack implementation will ideally be

  1. self-contained, in the sense it requires no input from the user,
  2. robust, in the sense it produces the correct result *every* time (not just sometimes),
  3. generic, in the sense it produces the correct result for *any* material (not just your own), and
  4. efficient.

- In this assignment, efficiency can be judged using various different metrics:

  – duration or number of accesses to the target device,
  – high-level, algorithm-related efficiency (e.g., computational complexity), and
  – low-level, implementation-related efficiency (e.g., execution time, memory footprint)

  all represent examples.

  Although presented in decreasing order of importance, these metrics will collectively determine the execution time of each attack; they are collectively important, therefore, because a strict 15min wall-clock time limit will be imposed[3] during the marking process. Any attack exceeding the limit will be terminated, so deemed to have failed (and thus incur the associated marks penalty).

[1] If your 7-digit UoB student numberis 0123456, for example, the corresponding URL would be `http://tinyurl.com/kua9nz2r/csdsp/cw/AttackSW/0123456.tar.gz`. If you have a problem downloading or unarchiving this file (e.g., you find it is missing, which can occur if you register late for the unit for example), it is *vital* you contact the lecturer responsible for the assignment immediately.

[2] 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.

[3] Although this limit is *mainly* imposed to stress the importance of attack efficiency, it *also* plays a crucial role in ensuring the marking process is viable. Note that this specific limit has been carefully selected to provide a liberal, loose upper-bound. Put another way, any attack that does *not* meet it could be deemed abnormal in some sense; such cases inherently demand attention wrt. one or more of the metrics above.

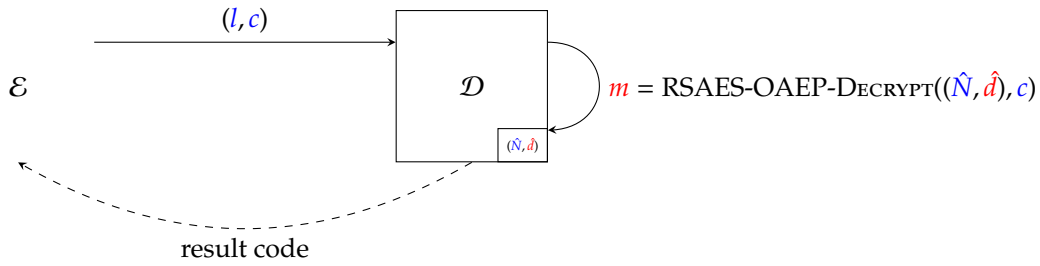
- A (subtle) difference exists between implementation and *genuine understanding* of the underlying theory; the written questions aim to reward advanced understanding which is difficult to demonstrate via source code alone. The requirement is simple: clearly and concisely answer as many of the questions as you can. Use the same, plain text file (*not* PDF, or similar) for your answers, inserting each one directly below the associated question.

  Note that although $Q.i$ denotes the i-th question, the questions may not be numbered sequentially: $Q.5$ may appear before $Q.3$, or perhaps $Q.4$ is missing for example. All questions are weighted equally: if there are n questions in total, each is worth $1/n$ of the associated marks.

3.3 Detail

3.3.1 Stage 1

3.3.1.1 Background Imagine you form part of a red team[4] asked to assess of a specific e-commerce server, denoted $\mathcal{D}$, which houses a 64-bit Intel Core2 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. In particular, it is able to compute RSAES-OAEP decryptions, as standardised by PKCS#1 v2.1 [6], by using an embedded RSA private key. By leveraging access to the network, an attacker $\mathcal{E}$ can interact with $\mathcal{D}$ as follows:



That is, in each interaction $\mathcal{E}$ can (adaptively) send a chosen RSAES-OAEP label and ciphertext to $\mathcal{D}$; the device will decrypt that ciphertext under the fixed, unknown RSA private key $(\hat{N}, \hat{d})$ and produce a result code based on validity of the underlying plaintext. Note that the associated plaintext is *not* produced explicitly as output.

RSAES-OAEP decryption makes use of Optimal Asymmetric Encryption Padding (OAEP) [2], per [6, Section 7.1]. 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 (that is, a given application should not reveal *which* error occurred, only that *some* error occurred). The implementation used by $\mathcal{D}$ does *not* adhere to this advice, meaning the error is exposed via the result code produced.

3.3.1.2 Material

`${ARCHIVE}/oaep/${USER}.D` This executable simulates the attack target $\mathcal{D}$. When executed it reads 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` and writes the following output

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

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

to `stdout`, in both cases with one field per line. Execution continues this way, i.e., by repeatedly reading input then writing output, until it is forcibly terminated (or crashes). The result code should be interpreted as follows:

- If the decryption was a success then the result code is 0.
- If error #1 occurred during decryption then the result code is 1.
- If error #2 occurred during decryption then the result code is 2.
- If there was some other internal error (e.g., due to malformed input) then the result code *attempts* to tell you why:
  - 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.

`${ARCHIVE}/oaep/${USER}.conf` This file represents a set of attack parameters, with everything (e.g., all public values) $\mathcal{E}$ has access to by default. 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), st. $\hat{e} \cdot \hat{d} \equiv 1 \pmod{\Phi(\hat{N})}$,
- $\hat{l}$, 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 $\hat{l}$),

with one field per line. More specifically, this represents the RSA public key $(\hat{N}, \hat{e})$ associated with the unknown RSA private key $(\hat{N}, \hat{d})$ embedded in $\mathcal{D}$, plus a ciphertext $\hat{c}$ whose decryption, i.e., the recovery of $\hat{m}$, 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.

3.3.1.3 Tasks

1. Write a program that simulates the adversary $\mathcal{E}$ by attacking the simulated target, or, more specifically, that recovers the target material $\hat{m}$. When executed using a command of the form

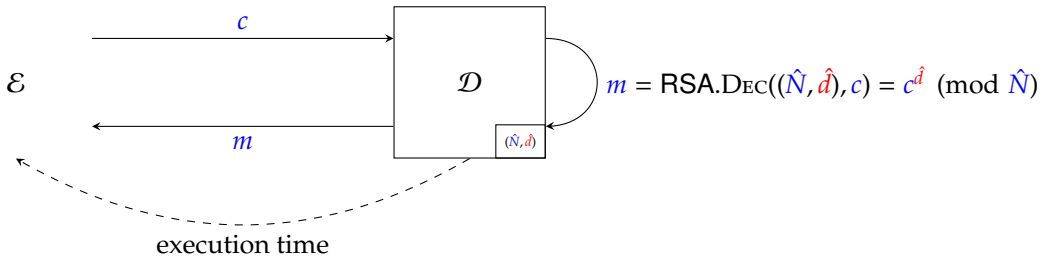
   ```
   ./attack ${USER}.D ${USER}.conf
   ```

   the attack should be invoked on the simulated target named (not some hard-coded alternative). Use `stdout` to print a) any intermediate output you deem relevant, followed finally by b) two lines which clearly detail the target material recovered plus the total number of interactions with the attack target.

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

3.3.2 Stage 2

3.3.2.1 Background Imagine you are tasked with attacking a server, denoted $\mathcal{D}$, which houses a 64-bit Intel Core2 processor. $\mathcal{D}$ is used by several front-line e-commerce servers, which offload computation relating to TLS handshakes. More specifically, $\mathcal{D}$ is used to compute an RSA decryption (in software, of the pre-master secret) whenever RSA-based key exchange is required. By leveraging access to the network, an attacker $\mathcal{E}$ can interact with $\mathcal{D}$ as follows:



That is, in each interaction $\mathcal{E}$ can (adaptively) send a chosen RSA ciphertext to $\mathcal{D}$; the device will decrypt that ciphertext under the fixed, unknown RSA private key $(\hat{N}, \hat{d})$ and produce the corresponding plaintext. In addition, $\mathcal{E}$ can measure the time $\mathcal{D}$ takes to execute each operation: it approximates this (keeping in mind there may be some experimental noise) by simply timing how long $\mathcal{D}$ takes to respond with m once provided with c.

3.3.2.2 Material

`${ARCHIVE}/time/${USER}.D` This executable simulates the attack target $\mathcal{D}$. When executed it reads the following input

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

from `stdin` and writes the following output

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

to `stdout`, in both cases with one field per line. Execution continues this way, i.e., by repeatedly reading input then writing output, until it is forcibly terminated (or crashes). Note that:

- $\mathcal{D}$ houses a 64-bit processor, so it uses 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, $\mathcal{D}$ uses a left-to-right binary exponentiation [5, Section 2.1] algorithm to compute $m = c^{\hat{d}} \pmod{\hat{N}}$. Montgomery multiplication [8] is harnessed to improve efficiency; following notation in [7], a CIOS-based [7, 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 st. $\hat{d}_l = 1$. However, it has been (artificially) selected st. $0 \leq \hat{d} < 2^{64}$ to limit the mount of time an attack requires: you should not rely on this fact, implying your attack should succeed for *any* $\hat{d}$ given enough time.

`${ARCHIVE}/time/${USER}.conf` This file represents a set of attack parameters, with everything (e.g., all public values) $\mathcal{E}$ has access to by default. 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), st. $\hat{e} \cdot \hat{d} \equiv 1 \pmod{\Phi(\hat{N})}$,

with one field per line. More specifically, this represents the RSA public key $(\hat{N}, \hat{e})$ associated with the unknown RSA private key $(\hat{N}, \hat{d})$ embedded in $\mathcal{D}$.

`${ARCHIVE}/time/${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 $\mathcal{D}$, bar accepting input of the form

- $\bar{c}$, 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),

on `stdin` (again with one field per line): in contrast to $\mathcal{D}$, this means $\mathcal{R}$ uses 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. $\gcd(\bar{N}, b) = 1$.

[5] MonPro [7, 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 $\mathcal{D}$ realises them concretely using the integrated CIOS algorithm.

3.3.2.3 Tasks

1. Write a program that simulates the adversary $\mathcal{E}$ by attacking the simulated target, or, more specifically, that recovers the target material $\hat{d}$. When executed using a command of the form

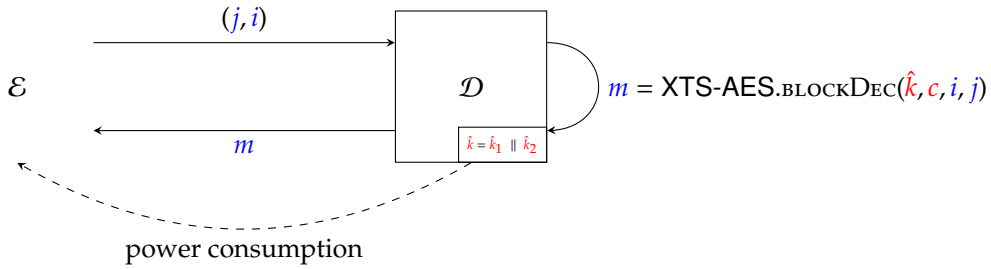
   ```
   ./attack ${USER}.D ${USER}.conf
   ```

   the attack should be invoked on the simulated target named (not some hard-coded alternative). Use `stdout` to print a) any intermediate output you deem relevant, followed finally by b) two lines which clearly detail the target material recovered plus the total number of interactions with the attack target.

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

3.3.3 Stage 3

3.3.3.1 Background Imagine you are tasked with attacking a device, denoted $\mathcal{D}$. 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 [3]. The decryption of data uses an XTS-AES key embedded in $\mathcal{D}$, 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, the password is *re*-entered as p' and a check whether $\hat{k} \stackrel{?}{=} f(p')$ then enforced; if the check fails, the SED refuses all interaction. $\mathcal{E}$ is lucky, and gets one-time access to $\mathcal{D}$ for a limited period of time while it is *already* powered-on (meaning the password check has already been performed). Put another way, $\mathcal{E}$ can interact with $\mathcal{D}$ as follows:



That is, in each interaction $\mathcal{E}$ can (adaptively) send a block and sector address (the latter of which doubles as the XTS-AES tweak) to $\mathcal{D}$; the device will read then decrypt the selected but unknown XTS-AES ciphertext under the fixed, unknown XTS-AES key $\hat{k}$ (per [3, Section 5.4.1]), and produce the corresponding plaintext. Note that $\mathcal{E}$ supplies the power and clock signals to $\mathcal{D}$, and can therefore measure the power consumed during each interaction: this yields at least one sample per instruction executed due to the high sample rate of the oscilloscope used (relative to the clock frequency demanded by and supplied to $\mathcal{D}$).

3.3.3.2 Material

`${ARCHIVE}/power/${USER}.D` This executable simulates the attack target $\mathcal{D}$. When executed it reads 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 tweak (represented as a length-prefixed, hexadecimal octet string),

from `stdin` and writes the following output

- Λ, a power consumption trace (whose representation is explained below), and
- m, a 1-block XTS-AES plaintext (represented as a length-prefixed, hexadecimal octet string),

to `stdout`, in both cases with one field per line. Execution continues this way, i.e., by repeatedly reading input then writing output, until it is forcibly terminated (or crashes). Note that:

- $\mathcal{D}$ 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 will be 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, $\mathcal{D}$ 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 $\mathcal{D}$ 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 ciphertext: valid use of $\mathcal{D}$ therefore demands

  - the block address satisfies $0 \leq j < 256$, and
  - the sector address satisfies $0 \leq i < 16777216$, which, since this implies only $\text{LSB}_{24}(i)$ is relevant, is equivalent to saying $\text{MSB}_{104}(i) = 0$.

  $\mathcal{E}$ can use *any* j and i as input to $\mathcal{D}$: if either is invalid, it uses a null ciphertext (i.e., 16 zero bytes, vs. a ciphertext read from the disk itself) but otherwise functions as normal.

- $\mathcal{D}$ 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 invocations dominate the computation it will perform for each interaction.

- Each power consumption trace Λ is a comma-separated line of the form

  $$l, s_0, s_1, \ldots, s_{l-1}$$

  where

  - l (represented as a decimal integer string), specifies the trace length, and
  - a given s_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 $0 \leq s_i < 256$ for $0 \leq i < l$.

`${ARCHIVE}/power/${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. In this case, $\mathcal{R}$ is a prototype version of $\mathcal{D}$ used to develop and debug the encryption mechanism: although identical in terms of power consumption, it accepts input of the form

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

on `stdin` (again with one field per line): in contrast to $\mathcal{D}$, this means $\mathcal{R}$ uses the *chosen* key $\bar{k}$ to decrypt the *chosen* ciphertext $\bar{c}$.

3.3.3.3 Tasks

1. Write a program that simulates the adversary $\mathcal{E}$ by attacking the simulated target, or, more specifically, that recovers the target material $\hat{k}$. When executed using a command of the form

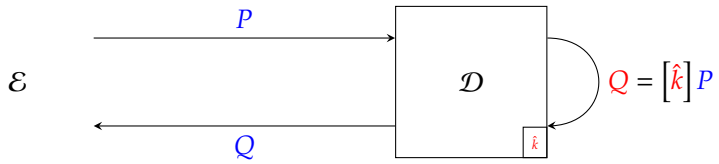
   ```
   ./attack ${USER}.D
   ```

   the attack should be invoked on the simulated target named (not some hard-coded alternative). Use `stdout` to print a) any intermediate output you deem relevant, followed finally by b) two lines which clearly detail the target material recovered plus the total number of interactions with the attack target.

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

3.3.4 Stage 4

3.3.4.1 Background Imagine you own a cryptographic accelerator, denoted $\mathcal{D}$: 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). $\mathcal{D}$ is primarily used to perform elliptic curve scalar multiplication, meaning you can interact with it as follows:



That is, in each interaction you can (adaptively) send a point to $\mathcal{D}$; the accelerator will multiply that point by the fixed, unknown scalar $\hat{k}$, and produce the corresponding point. Although it currently functions correctly, an electrical fault demands that $\mathcal{D}$ 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 various existing data sets will be lost, *unless* $\hat{k}$ itself can be recovered by some other means.

3.3.4.2 Material

`${ARCHIVE}/invalid/${USER}.D` This executable simulates the attack target $\mathcal{D}$. When executed it reads 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` and writes the following output

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

to `stdout`, in both cases with one field per line. Execution continues this way, i.e., by repeatedly reading input then writing output, until it is forcibly terminated (or crashes). Note that:

- $\mathcal{D}$ uses an affine, left-to-right binary scalar multiplication (i.e., an additive analogue of [5, Section 2.1]) to compute $Q = [\hat{k}]P$: 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, $\mathcal{D}$ 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, $\mathcal{D}$ 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).

3.3.4.3 Tasks

1. Write a program that simulates the adversary $\mathcal{E}$ by attacking the simulated target, or, more specifically, that recovers the target material $\hat{k}$. When executed using a command of the form

   `./attack ${USER}.D`

   the attack should be invoked on the simulated target named (not some hard-coded alternative). Use `stdout` to print a) any intermediate output you deem relevant, followed finally by b) two lines which clearly detail the target material recovered plus the total number of interactions with the attack target.
2. Answer the exam-style questions in `${ARCHIVE}/invalid/${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 p. 7).

[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. 7).

[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. 9).

[5] D.M. Gordon. "A Survey of Fast Exponentiation Methods". In: *Journal of Algorithms* 27 (1998), pp. 129–146 (see pp. 6, 9).

[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] 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 p. 6).

[8] P.L. Montgomery. "Modular multiplication without trial division". In: *Mathematics of Computation* 44.170 (1985), pp. 519–521 (see p. 6).

A Representation and conversion

A.1 Integers

A.1.0.1 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

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

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

$$\begin{aligned} \hat{x} \mapsto & \hat{x}_{19} \cdot 10^{19} + \hat{x}_{18} \cdot 10^{18} + \hat{x}_{17} \cdot 10^{17} + \hat{x}_{16} \cdot 10^{16} + \hat{x}_{15} \cdot 10^{15} + \\ & \hat{x}_{14} \cdot 10^{14} + \hat{x}_{13} \cdot 10^{13} + \hat{x}_{12} \cdot 10^{12} + \hat{x}_{11} \cdot 10^{11} + \hat{x}_{10} \cdot 10^{10} + \\ & \hat{x}_{9} \cdot 10^{9} + \hat{x}_{8} \cdot 10^{8} + \hat{x}_{7} \cdot 10^{7} + \hat{x}_{6} \cdot 10^{6} + \hat{x}_{5} \cdot 10^{5} + \\ & \hat{x}_{4} \cdot 10^{4} + \hat{x}_{3} \cdot 10^{3} + \hat{x}_{2} \cdot 10^{2} + \hat{x}_{1} \cdot 10^{1} + \hat{x}_{0} \cdot 10^{0} \\ \mapsto & 0_{(10)} \cdot 10^{19} + 9_{(10)} \cdot 10^{18} + 0_{(10)} \cdot 10^{17} + 8_{(10)} \cdot 10^{16} + 0_{(10)} \cdot 10^{15} + \\ & 7_{(10)} \cdot 10^{14} + 0_{(10)} \cdot 10^{13} + 6_{(10)} \cdot 10^{12} + 0_{(10)} \cdot 10^{11} + 5_{(10)} \cdot 10^{10} + \\ & 0_{(10)} \cdot 10^{9} + 4_{(10)} \cdot 10^{8} + 0_{(10)} \cdot 10^{7} + 3_{(10)} \cdot 10^{6} + 0_{(10)} \cdot 10^{5} + \\ & 2_{(10)} \cdot 10^{4} + 0_{(10)} \cdot 10^{3} + 1_{(10)} \cdot 10^{2} + 0_{(10)} \cdot 10^{1} + 0_{(10)} \cdot 10^{0} \\ \mapsto & 90807060504030201 00_{(10)} \end{aligned}$$

  whereas

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

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

A.1.0.2 Example. Consider the following Python program

```
a = "09080706050403020100"

b = long( a, 10 )
c = long( a, 16 )

d = ( "%d" % ( c ) )
e = ( "%X" % ( c ) )

f = ( "%X" % ( c ) ).zfill( 20 )

print "type( a ) = %-13s a = %s" % ( type( a ), str( a ) )
print "type( b ) = %-13s b = %s" % ( type( b ), str( b ) )
print "type( c ) = %-13s c = %s" % ( type( c ), str( c ) )
print "type( d ) = %-13s d = %s" % ( type( d ), str( d ) )
print "type( e ) = %-13s e = %s" % ( type( e ), str( e ) )
print "type( f ) = %-13s f = %s" % ( type( f ), str( f ) )
```

which, when executed, produces

```
type( a ) = <type 'str'>  a = 09080706050403020100
type( b ) = <type 'long'> b = 9080706050403020100
type( c ) = <type 'long'> c = 42649378395939397566720
type( d ) = <type 'str'>  d = 42649378395939397566720
type( e ) = <type 'str'>  e = 9080706050403020100
type( f ) = <type 'str'>  f = 09080706050403020100
```

The idea is that

- a is an integer string (i.e., a sequence of characters),
- b and c are conversions of a into integers (actually a Python `long`, which is the multi-precision integer type used), using decimal and hexadecimal respectively, and
- d and e are conversions of c into strings (i.e., a sequence of characters), using decimal and hexadecimal respectively.

Note that a and e do not match: the conversion has left out the left-most zero character, since this is not significant wrt. the integer value. To resolve this issue where it is problematic, the `zfill` function can be used to left-fill the string with zero characters until it is of the required length (here 20 characters in total, forming f which does then match).

A.2 Octet strings

A.2.0.1 Concept. The term octet[6] 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*[7] differ. A string is a sequence of characters, and so, by analogy, an octet string[8] 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[9] 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[10] to mean n, the length of the source byte sequence, is prepended to the octet string as a single 8-bit[11] length or "header" octet, and c) hexadecimal[12] 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 $x = \langle 00_{(16)}, 01_{(16)}, \ldots, 0F_{(16)} \rangle = \langle 0_{(10)}, 1_{(10)}, \ldots, 15_{(10)} \rangle \equiv \mathbf{x}$.

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

[7] 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.

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

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

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

[11] 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.

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

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.

A.2.0.2 Example. Consider the following Python program

```
import binascii

def str2seq( x ) :
  return          [ ord( t ) for t in x ]

def seq2str( x ) :
  return ''.join( [ chr( t ) for t in x ] )

def octetstr2str( 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 str2octetstr( x ) :
  return ( '%02X' % ( len( x ) ) ) + ':' + ( binascii.b2a_hex( x ) )

t_0 = '\xDE\xAD\xBE\xEF' ; t_1 = [ 0xDE, 0xAD, 0xBE, 0xEF ]

t_2 = str2octetstr(          t_0   )
t_3 = str2octetstr( seq2str( t_1 ) )

print "type( t_0 ) = %-18s t_0 = %s" % ( type( t_0 ), repr( t_0 ) )
print "type( t_1 ) = %-18s t_1 = %s" % ( type( t_1 ), repr( t_1 ) )
print "type( t_2 ) = %-18s t_2 = %s" % ( type( t_2 ), repr( t_2 ) )
print "type( t_3 ) = %-18s t_3 = %s" % ( type( t_3 ), repr( t_3 ) )

t_4 = '04:8BADF00D'

t_5 =          octetstr2str( t_4 )
t_6 = str2seq( octetstr2str( t_4 ) )

print "type( t_4 ) = %-18s t_4 = %s" % ( type( t_4 ), repr( t_4 ) )
print "type( t_5 ) = %-18s t_5 = %s" % ( type( t_5 ), repr( t_5 ) )
print "type( t_6 ) = %-18s t_6 = %s" % ( type( t_6 ), repr( t_6 ) )
```

which, when executed, produces

```
type( t_0 ) = <type 'str'>       t_0 = '\xde\xad\xbe\xef'
type( t_1 ) = <type 'list'>      t_1 = [222, 173, 190, 239]
type( t_2 ) = <type 'str'>       t_2 = '04:deadbeef'
type( t_3 ) = <type 'str'>       t_3 = '04:deadbeef'
type( t_4 ) = <type 'str'>       t_4 = '04:8BADF00D'
type( t_5 ) = <type 'str'>       t_5 = '\x8b\xad\xf0\r'
type( t_6 ) = <type 'list'>      t_6 = [139, 173, 240, 13]
```

as output. This is intended to illustrate that

- If you start with byte string (or array) `t_0`, then `str2octetstr` convert this into the octet string, you expect that `t_2 = 04:DEADBEEF` implies $n = 04_{(16)} = 4_{(10)}$ and $x = \langle DE_{(16)}, AD_{(16)}, BE_{(16)}, EF_{(16)} \rangle$ st. `t_0[ 0 ]` $= DE_{(16)} = x_0$.
- If you start with byte sequence `t_1`, then `seq2str` and `str2octetstr` convert this into the octet string, you expect that `t_3 = 04:DEADBEEF` implies $n = 04_{(16)} = 4_{(10)}$ and $x = \langle DE_{(16)}, AD_{(16)}, BE_{(16)}, EF_{(16)} \rangle$ st. `t_1[ 1 ]` $= AD_{(16)} = x_1$.
- If you start with octet string `t_4`, then `octetstr2str` convert this into the byte array, you expect that `t_4 = 04:8BADF00D` implies $n = 04_{(16)} = 04_{(10)}$ and $x = \langle 8B_{(16)}, AD_{(16)}, F0_{(16)}, 0D_{(16)} \rangle$ st. `t_5[ 2 ]` $= F0_{(16)} = x_2$.
- If you start with octet string `t_4`, then `octetstr2str` and `str2seq` convert this into the byte sequence, you expect that `t_4 = 04:8BADF00D` implies $n = 04_{(16)} = 04_{(10)}$ and $x = \langle 8B_{(16)}, AD_{(16)}, F0_{(16)}, 0D_{(16)} \rangle$ st. `t_6[ 3 ]` $= 0D_{(16)} = x_3$.