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

Assessed coursework
Applied Security (COMS30901)

Att

Note that:

1. **The deadline for this courswork is** 17/03/17 **(for Att1, stages** 1 **and** 2**) then** 05/05/17 **(for Att2, stages** 3 **and** 4**), with standard regulations enforced wrt. late submission.**
2. **This courswork represents** 35 **(for Att1, stages** 1 **and** 2**) plus** 35 **(for Att2, stages** 3 **and** 4**) percent of marks available for COMS30901, 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:**
   - **the unit forum hosted via**

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

     where lecturers, lab. demonstrators and students all regularly post questions and answers,
   - **the lecturer responsible for this courswork, who is available within stated office hours, by appointment, or 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. More specifically, the goal is to research and then implement some *real* attacks against *real* cryptosystems that are deployed in *real* applications.

2 Terms and conditions

- This assignment is intended to help you learn something; where there is some debate about the right approach, the assignment demands that *you* make an informed decision *yourself*. Such decisions should ideally be based on a reasoned argument formed via your *own* background research (rather than reliance on the teaching material alone).
- *You* can select the programming language used to implement each attack: viable examples include C, Java, and Python. However, since it will be marked using a platform equivalent to the CS lab. (MVB-2.11), it *must* compile, execute and be thoroughly tested against default operating system and development tool-chain versions available.

  Use of (correctly cited) third-party libraries *is* allowed iff. they satisfy the same criteria: viable examples include GMP or OpenSSL for C, the Java Cryptography Architecture (JCA) for Java, and the `hashlib` module for Python.
- The assignment description may refer to `marksheet.txt`. Download this ASCII text file from

  `http://tinyurl.com/zetunur/csdsp/crypto/cw/Att/marksheet.txt`

  then complete and include it in your submission: this is important, and failure to do so may result in a loss of marks.
- Include a set of instructions that clearly describe how to compile and execute your submission: the ideal approach would be to either submit a `Makefile` or equivalent (if not provided), or use `marksheet.txt`.
- Even though you can *definitely* expect to receive partial marks for a partial solution, it will be marked as is: for example, there will be no effort to enable optional or commented functionality (by uncommenting it, or setting specific compile-time parameters).
- 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.
- A subtle difference exists between implementation and *genuine understanding* of the underlying theory. The written questions for each stage attempt to assess the latter, and, in particular, 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.
- You should submit your work via the SAFE submission system at

  `http://tinyurl.com/jqfggey`

  including all source code, written solutions and any auxiliary files you think are important (e.g., any example input or output).
- Keep in mind that the staggered, 2-part submission[1] is more complicated than normal. Ensure you are aware of both the deadline and stages associated with each part, and carefully submit into the correct component in SAFE.

[1] It is important to stress that although the marking process is much more labour intensive as a result, this approach has been adopted in direct response to feedback from previous cohorts. Previously, we set one deadline (at the end of TB2, allowing the maximum duration); students (very) vocally claimed this made time management hard(er), and that staggered deadlines to focus their effort were preferable. This approach also allows feedback from your first submission to inform your second, which is clearly good practice.

3 Material

Personalised, per student material relating to each stage is provided for you to use. Assuming `${USER}` represents your candidate number, download[2] and unarchive the file

`http://tinyurl.com/zetunur/csdsp/crypto/cw/Att/${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 that `${ARCHIVE}` refers to said location. You should find the following sub-directories

- `${ARCHIVE}/${USER}/oaep`
- `${ARCHIVE}/${USER}/time`
- `${ARCHIVE}/${USER}/fault`
- `${ARCHIVE}/${USER}/power`

relating to associated stages of the assignment. Note that:

- The behaviour of each attack target is simulated by an executable program, which was compiled and linked on a platform equivalent to those in CS lab. (MVB-2.11). As such, the attack targets are only guaranteed to work on the same or a compatible[3] platform, and, even then, only once appropriate permissions are 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. The lab. worksheet(s) offer an introduction to *and* prepared framework for pipe-based inter-process communication, so support the work required to interact between your attacks and associated, simulated targets.
- To make submission via SAFE easier, the recommended approach is to develop your solution within the *same* directory structure as the material provided. This allows you to create and submit an archive (`${USER}-solution.tar.gz`, for example, by using `tar` and `gzip`) of your *entire* solution rather than deal with numerous separate files.
- Although the functional correctness of your attack implementation is obviously crucial wrt. marks, various additional criteria have an impact. `marksheet.txt` offers a high-level idea of the marking scheme, but it remains *your* task to consider and then address specific criteria for each stage. For example, each 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,
  – higher-level algorithmic efficiency, and
  – lower-level optimisation of the implementation

  all represent examples, in roughly decreasing order of importance. A notional, wall-clock time limit is included for each attack: it is rough, and not enforced during the marking process, but can be used to gauge whether or not your solution can be deemed efficient versus the latter metrics above.

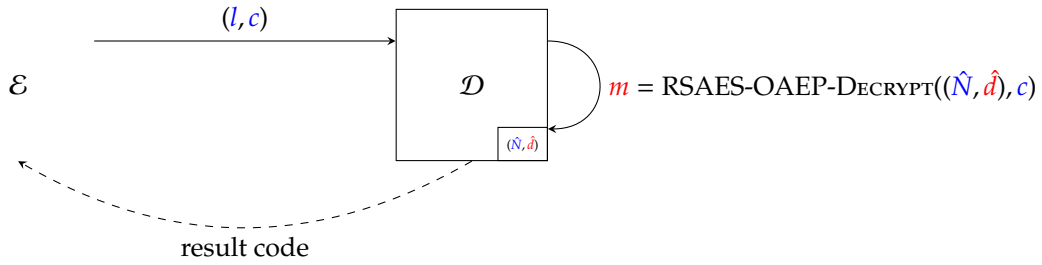
[2] If your candidate number were 46918, for example, the corresponding URL would be `http://tinyurl.com/zetunur/csdsp/crypto/cw/Att/46918.tar.gz`. If you have a problem downloading or unarchiving this file (e.g., you find it is missing, which can occur if you register late for the unit for example), it is *vital* you contact the lecturer responsible for the assignment immediately.

[3] For instance, one way to reduce your dependency on workstations in the CS lab. is to use a compatible operating system in a VM or as a LiveCD on your own workstation.

4 Stage 1

4.1 Background

As part of a penetration testing team, imagine you are asked to assess 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 [5], 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 [5, 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 [5, 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.

4.2 Materials

4.2.1 `${ARCHIVE}/${USER}/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 an octet string, which may be empty: a blank line represents a 0-octet string), and
- c, an RSAES-OAEP ciphertext (represented as an octet string)

from `stdin` and writes the following output

- Λ, a result code (represented as a decimal 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). 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.

4.2.2 `${ARCHIVE}/${USER}/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 an octet string, which may be empty, i.e., a blank link meaning a 0-octet string), and
- $\hat{c}$, an RSAES-OAEP ciphertext (represented as an 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.

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 $\hat{m}$. When executed using a command of the form

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

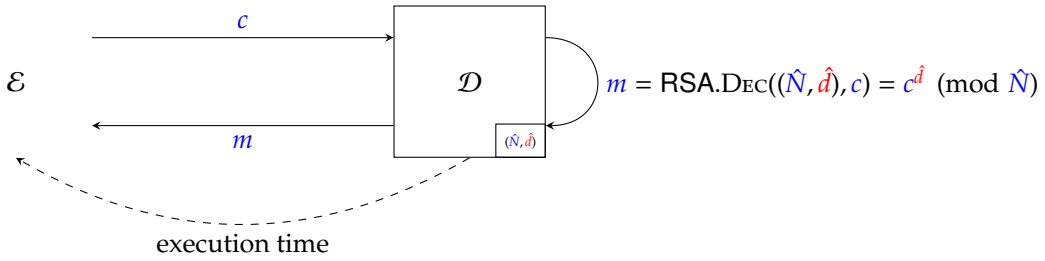
   the attack should be invoked on the simulated target named (not some hard-coded alternative). Use `stdout` to print a) any intermediate output you deem relevant, followed finally by b) two lines which clearly detail the target material recovered plus the total number of interactions with attack target. A notional, wall-clock time limit to aim for would be 2 min (although this is a *very* rough, *upper* bound only).

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

5 Stage 2

5.1 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.

5.2 Materials

5.2.1 `${ARCHIVE}/${USER}/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, $\mathcal{D}$ instead uses vanilla left-to-right binary exponentiation [4, Section 2.1] to compute $m = c^{\hat{d}} \pmod{\hat{N}}$. However, to optimise the implementation, it makes use of Montgomery multiplication [7] internally. Following Koç et al. [6], the Coarsely Integrated Operand Scanning (CIOS) method [6, Section 5] is used to realise steps 1 and 2 (i.e., integrated multiplication and subsequent reduction) of MonPro [6, Section 2].
- Following the same notation as [4], $\hat{d}$ is assumed to have $l + 1$ bits where $\hat{d}_l = 1$. 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.

5.2.2 `${ARCHIVE}/${USER}/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 $\mathcal{D}$.

5.2.3 `${ARCHIVE}/${USER}/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. $\gcd(\bar{N}, b) = 1$.

5.3 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

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

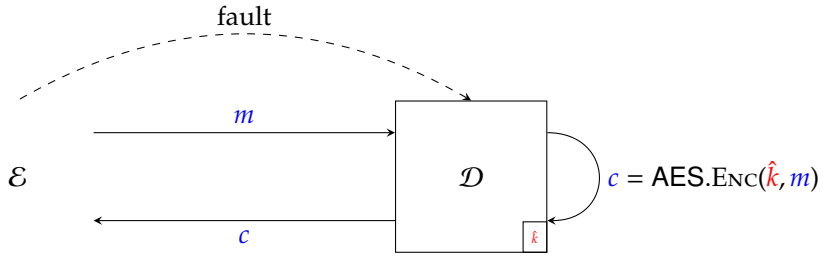
   the attack should be invoked on the simulated target named (not some hard-coded alternative). Use `stdout` to print a) any intermediate output you deem relevant, followed finally by b) two lines which clearly detail the target material recovered plus the total number of interactions with attack target. A notional, wall-clock time limit to aim for would be 5 min (although this is a *very* rough, *upper* bound only).

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

6 Stage 3

6.1 Background

Imagine you are tasked with attacking a device, denoted $\mathcal{D}$, which houses an 8-bit Intel 8051 processor. $\mathcal{D}$ represents an ISO/IEC 7816 compliant contact-based smart-card chip, used within larger devices as a co-processor module: it has support for secure key generation and storage, plus off-load of some cryptographic operations, such as AES, via a standardised protocol (or API). By leveraging physical access, an attacker $\mathcal{E}$ can interact with $\mathcal{D}$ as follows:



That is, in each interaction $\mathcal{E}$ can (adaptively) send a chosen AES plaintext to $\mathcal{D}$; the device will encrypt that plaintext under the fixed, unknown AES cipher key $\hat{k}$, and produce the corresponding ciphertext. Note that $\mathcal{E}$ supplies the power and clock signals to $\mathcal{D}$. Using an irregular clock signal (e.g., via some form of glitch to disrupt regular oscillation) causes $\mathcal{D}$ to malfunction. One malfunction, or fault, can be induced per interaction with $\mathcal{D}$: it will act to randomise one element of the state matrix used by AES, at a chosen point during the associated encryption operation.

6.2 Material

6.2.1 `${ARCHIVE}/${USER}/fault/${USER}.D`

This executable simulates the attack target $\mathcal{D}$. When executed it reads the following input

- Λ, a fault specification (whose representation is explained below), and
- m, a 1-block AES plaintext (represented as an octet string)

from `stdin` and writes the following output

- c, a 1-block AES ciphertext (represented as an 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. Per FIPS 197 [1, Figure 5], the fact $Nb = 4$ and $Nr = 10$ means a (4×4)-element state matrix will be 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 this latter case, no fault will induced (which then obviously yields 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 st. $\delta = 0$ *are* possible: 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.

6.3 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

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

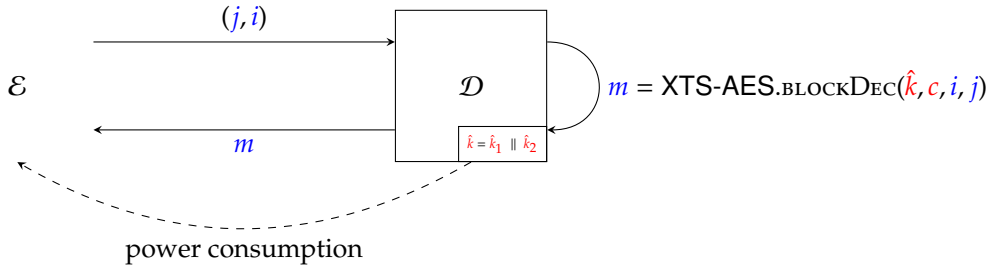
the attack should be invoked on the simulated target named (not some hard-coded alternative). Use `stdout` to print a) any intermediate output you deem relevant, followed finally by b) two lines which clearly detail the target material recovered plus the total number of interactions with attack target. A notional, wall-clock time limit to aim for would be 5 min (although this is a *very* rough, *upper* bound only).

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

7 Stage 4

7.1 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 $\mathcal{D}$).

7.2 Materials

7.2.1 `${ARCHIVE}/${USER}/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 an 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 an 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. Per FIPS 197 [1, Figure 5], the fact $Nb = 4$ 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 $0 \leq i < l$.

7.2.2 `${ARCHIVE}/${USER}/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 an octet string),
- $\bar{c}$, a 1-block XTS-AES ciphertext (represented as an octet string), and
- $\bar{k}$, an XTS-AES key (represented as an 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 $\bar{c}$.

7.3 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

```
bash$ ./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 attack target. A notional, wall-clock time limit to aim for would be 10 min (although this is a *very* rough, *upper* bound only).

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

References

[1] *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 197. http://csrc.nist.gov/publications/. 2001 (see pp. 8, 10).

[2] M. Bellare and P. Rogaway. "Optimal asymmetric encryption". In: *Advances in Cryptology (EUROCRYPT)*. Springer-Verlag LNCS 950, 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. http://standards.ieee.org/. 2007 (see p. 10).

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

[5] 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. http://tools.ietf.org/html/rfc3447. 2003 (see p. 4).

[6] 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).

[7] 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

Somewhat bizarrely, one of the most confusing aspects of this assignment can often be correctly formatting input and/or interpreting output! One reason is that similar *looking* human-readable strings *may* be (subtly) different depending on their machine-readable representation.

To illustrate common problems, and demonstrate the conversion to and from pertinent data types, the following Sections offer a suite of Python-based examples. Although each example can be ported into other languages, the choice of Python is motivated by the fact it allows a) concise, easy to read source code that can quickly be experimented with, and b) suitable interpreters are widely available, (typically) with no extra installation required.

A.1 Decimal and hexadecimal integer strings

A.1.1 Representation

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

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

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

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

  whereas

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

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

A.1.2 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

```
bash$ python integer.py
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.2.1 Representation

An octet string specifies a sequence of octets (i.e., 8-bit bytes), each written as two hexadecimal digits; this implies the length of an octet string is always an even number of digits. As such, the 20-character string

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

represents the 10-element octet sequence

$$\hat{x} \quad \mapsto \quad \langle 09_{(16)}, 08_{(16)}, 07_{(16)}, 06_{(16)}, 05_{(16)}, 04_{(16)}, 03_{(16)}, 02_{(16)}, 01_{(16)}, 00_{(16)} \rangle$$

which, in turn, is more or less the same as defining the C array

```
uint8_t x[] = { 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00 };
```

In a sense, this means we read left-to-right: the 0-th octet within the octet sequence is the left-most one (i.e., the left-most pair of characters) within the octet string.

A.2.2 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 ] )

a = "09080706050403020100"
```

```
b = str2seq( binascii.a2b_hex( a ) )
c = binascii.b2a_hex( seq2str( b ) )

d = sum( [ b[ i ] * 2 ** ( 8 *              i       ) for i in range( len( b ) ) ] )
e = sum( [ b[ i ] * 2 ** ( 8 * ( len( b ) - i - 1 ) ) for i in range( len( b ) ) ] )

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 ) )
```

which, when executed, produces

```
bash$ python octet.py
type( a ) = <type 'str'>  a = 09080706050403020100
type( b ) = <type 'list'> b = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
type( c ) = <type 'str'>  c = 09080706050403020100
type( d ) = <type 'long'> d = 18591708106338011145
type( e ) = <type 'long'> e = 42649378395939397566720
```

The idea is that

- a is an octet string (i.e., a sequence of characters),
- b is the conversion of a into an octet sequence (i.e., a sequence of 8-bit bytes), and
- c is the conversion of b into an octet string (i.e., a sequence of characters),

noting that a and c match. The conversion uses the `binascii` module to convert character strings to/from sequences of bytes; these are converted to/from sequences of usable integer using the two user-defined functions `str2seq` and `seq2str`. Then,

- d is a little-endian integer converted from b via

$$\mathsf{d} = \sum_{i=0}^{i<|\mathsf{b}|} \mathsf{b}_i \cdot 2^{8 \cdot i},$$

  and

- e is a big-endian integer converted from b via

$$\mathsf{e} = \sum_{i=0}^{i<|\mathsf{b}|} \mathsf{b}_i \cdot 2^{8 \cdot (|\mathsf{b}|-1-i)}.$$

The intuition here is that the former little-endian conversion weights each i-th digit normally (i.e., per the normal base-2^8 expansion), but the latter big-endian conversion reverses this weighting (so each i-th digit is instead weighted as if it were the $(|\mathsf{b}| - 1 - i)$-th digit).

A.2.3 Caveats

Per the above, a side-effect of big-endian conversion (e.g., via PKCS#1 I2OSP and OS2IP functions) is that an octet string has a natural meaning when considered as a (hexadecimal) integer. For example, the octet string

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

will be interpreted by OS2IP as the big-endian integer

$$\begin{aligned} \hat{x} \;\mapsto\; & 09_{(16)} \cdot 16^9 + 08_{(16)} \cdot 16^8 + 07_{(16)} \cdot 16^7 + 06_{(16)} \cdot 16^6 + 05_{(16)} \cdot 16^5 + \\ & 04_{(16)} \cdot 16^4 + 03_{(16)} \cdot 16^3 + 02_{(16)} \cdot 16^2 + 01_{(16)} \cdot 16^1 + 00_{(16)} \cdot 16^0 \\ \;\mapsto\; & 42649378395939397566720_{(10)} \end{aligned}$$

which is essentially the same as if we just interpreted $\hat{x}$ as an hexadecimal integer string.

However, there is a subtle caveat which needs care: an octet string of the *wrong* length is likely to be misinterpreted. For example, consider the 19-character string

$$\hat{y} = \texttt{0908070605040302010}$$

where versus the previous example $\hat{x}$, the right-most zero character has been removed. Formally, this is no longer a valid octet string: since it contains an odd number of characters, one octet is partially specified (using one character rather than two). If we were to interpret it as a hexadecimal integer string, the value would be

$$2665586149746212347920_{(10)}.$$

But, read from left-to-right (noting the partial octet) as before we actually end up with the octet sequence

$$\hat{y} \mapsto \langle 09_{(16)}, 08_{(16)}, 07_{(16)}, 06_{(16)}, 05_{(16)}, 04_{(16)}, 03_{(16)}, 02_{(16)}, 01_{(16)}, 0_{(16)} \rangle$$

and hence

$$\begin{aligned} \hat{y} \mapsto & \; 09_{(16)} \cdot 16^9 + 08_{(16)} \cdot 16^8 + 07_{(16)} \cdot 16^7 + 06_{(16)} \cdot 16^6 + 05_{(16)} \cdot 16^5 + \\ & \; 04_{(16)} \cdot 16^4 + 03_{(16)} \cdot 16^3 + 02_{(16)} \cdot 16^2 + 01_{(16)} \cdot 16^1 + 0_{(16)} \cdot 16^0 \\ \mapsto & \; 42649378395939397566720_{(10)} \end{aligned}$$

again! If we really meant the octet string to have one less digit in it, probably we meant to pad with a left-most zero to get the 20-character string

$$\hat{z} = \texttt{00908070605040302010}$$

and hence the octet sequence

$$\hat{z} \mapsto \langle 00_{(16)}, 90_{(16)}, 80_{(16)}, 70_{(16)}, 60_{(16)}, 50_{(16)}, 40_{(16)}, 30_{(16)}, 20_{(16)}, 10_{(16)} \rangle$$

which is now interpreted correctly as

$$\begin{aligned} \hat{z} \mapsto & \; 00_{(16)} \cdot 16^9 + 90_{(16)} \cdot 16^8 + 80_{(16)} \cdot 16^7 + 70_{(16)} \cdot 16^6 + 60_{(16)} \cdot 16^5 + \\ & \; 50_{(16)} \cdot 16^4 + 40_{(16)} \cdot 16^3 + 30_{(16)} \cdot 16^2 + 20_{(16)} \cdot 16^1 + 10_{(16)} \cdot 16^0 \\ \mapsto & \; 2665586149746212347920_{(10)} \end{aligned}$$