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

Assessed coursework
Applied Security (COMS30901)

Att
(Assignment description)

Note that:

1. **The deadline for this assignment is** 21/03/14 **(for Att1, stages** 1 **and** 2**) then** 09/05/14 **(for Att2, stages** 3**,** 4 **and** 5**).**

2. **This assignment represents** 30 **(for Att1, stages** 1 **and** 2**) plus** 45 **(for Att2, stages** 3**,** 4 **and** 5**) percent of the total marks available for COMS30901. It has been deemed an individual assignment: make sure you are aware of and adhere to the University and Faculty regulations which govern this type of work.**

3. **There are numerous support resources available, for example:**

   - **the online unit forum, which is hosted at**

     https://www.cs.bris.ac.uk/forum/index.jsp?title=COMS30901

     and on which lecturers, lab. demonstrators and students post questions and answers,

   - **the responsible lecturer, namely Dan Page, who is available in designated office hours, by appointment, or via email at**

     page@cs.bris.ac.uk

1 Introduction

Cryptographic attacks can be placed into at least two categories: those which focus on the underlying theory or design (including Mathematics), and those which focus on the behaviour of a resulting implementation. This assignment is concerned with the second category. Specifically, the goal is to research and then implement *real* attacks against *real* cryptosystems that are deployed in *real* applications.

2 Submission and marking

- This assignment is intended to help you learn something; where there is some debate about the right approach, the assignment demands *you* make an informed decision *yourself* and back it up with a reasonable argument based on *your own* background research. There are
  - multiple initial stages relating to implementation of various attacks (which are independent from each other), plus
  - one final stage relating to analysis of the initial stages (and therefore ideally attempted after completing each of the attack implementations).
- The assignment description refers to `marksheet.txt`. Download this ASCII text file from

  `www.cs.bris.ac.uk/Teaching/Resources/COMS30901/cw/Att/question/marksheet.txt`

  then complete and include it in your submission; this is not optional, and failure to do so may result in a loss of marks.
- *You* may select the programming language used to implement each of the attacks: viable examples include C, C++, Java and Python. However, since your solution will be marked using a platform equivalent to those in the CS lab. (MVB-2.11), it *must* compile, execute and be tested using the default operating system and development tool-chain versions available.

  Use of correctly cited third-party libraries *is* allowed if they satisfy the same criteria: examples include GMP or OpenSSL for C and C++, the Java Cryptography Architecture (JCA) for Java, and the `hashlib` module for Python.
- The functional correctness of your attack implementations is of course crucial wrt. marks, but some additional criteria apply. For example, each attack should ideally be
  1. self-contained, in the sense it requires no input from the user,
  2. robust and reproducible, in the sense the correct result is produced *every* time (not just sometimes), and
  3. generic, in the sense that the marking process will exercise it using input you have not seen.

  A partial solution (e.g., an attack which is partially successful) is certainly eligible for partial marks.
- Generally speaking, a more efficient solution will be viewed as better. Note however that efficiency can be judged using various metrics: in particular,
  1. duration of or number of accesses to the target device,
  2. higher-level algorithmic efficiency, or
  3. lower-level optimisation of attack implementation

  represent examples, roughly in decreasing order of importance. In significant cases `marksheet.txt` is explicit about the requirements, but you should assume optimising your implementation is an implicit goal throughout.
- Where appropriate, include instructions that carefully describe how to compile and execute your submission; the ideal solution would include a `Makefile` (or equivalent) where not already provided.

- To make the marking process easier, your solution should only write error messages to `stderr` (or equivalent). 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 work via the SAFE submission system at

  `wwwa.fen.bris.ac.uk/COMS30901/`

  including all source code, written solutions and any auxiliary files you think are important (e.g., example input and output). In the unlikely event that the submission system fails, email your work to the lecturer responsible for the assignment: the same rules wrt. deadlines and late work will still apply.

  Note that the staggered, 2-part submission[1] is more complicated than normal. Make sure you are aware of the deadlines, and carefully submit into the correct component in SAFE: there are two, namely `Att1` (for Att1, stages 1 and 2), and `Att2` (for Att2, stages 3, 4 and 5).

3 Material

Personalised, per-student material relating to each stage is provided for you to use. Given that `${USER}` represents your user name for the CS lab. (e.g., `wj7805`), download and unarchive the file

`www.cs.bris.ac.uk/Teaching/Resources/COMS30901/cw/Att/question/${USER}.tar.gz`

somewhere secure within your file system[2]; we assume `${ATTACK}` refers to said location from here on. As such, you should find the following sub-directories

- `${ATTACK}/${USER}/oaep/`,
- `${ATTACK}/${USER}/time/`,
- `${ATTACK}/${USER}/fault/`,
- `${ATTACK}/${USER}/power/`, and
- `${ATTACK}/${USER}/analysis/`,

relating to associated stages of the assignment. Note that:

- To make the submission process 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 `${USER}-solution.tar.gz`, using `tar` and `gzip` to archive *everything* into one file rather than deal with multiple files.
- The initial stages involve attacking targets whose behaviour is simulated by a set of executable programs. Said programs were originally compiled and linked on a platform equivalent to those available in the CS lab: they are *only* guaranteed to work on the same platform, and *only* after ensuring 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.

[1] It is important to stress that this approach has been adopted in direct response to feedback from previous cohorts. We previously set one deadline at the end of TB2 (allowing the maximum duration possible); students very vocally claimed this made time management hard(er), and that staggered deadlines to focus their effort were preferable.

[2] If your user name is `jh6970` for example, then the corresponding URL would be `www.cs.bris.ac.uk/Teaching/Resources/COMS30901/cw/Att/question/jh6970.tar.gz`. If there is a problem downloading or unarchiving this file, it is *vital* you contact the lecturer responsible for the assignment *immediately* so you have enough time to complete it.

4 Stage 1: an attack based on error messages

4.1 Background

Imagine you are tasked with attacking a given e-commerce server, denoted $\mathcal{D}$, which houses an Intel Core2 processor. The server forms an important part of the back-office infrastructure supporting various web-sites; it is hard-coded with a fixed program that performs RSAES-OAEP decryption as specified by PKCS#1 v2.1 [2].

RSAES-OAEP decryption uses a version of Optimal Asymmetric Encryption Padding (OAEP) [3], and is described in [2, Section 7.1]. Notice that during decryption, various error conditions may occur:

Error #1 : A decryption error occurs in Step 3.g of RSAES-OAEP-Decrypt if an octet string passed into the decoding phase does not have $00_{(16)}$ as the most-significant octet.

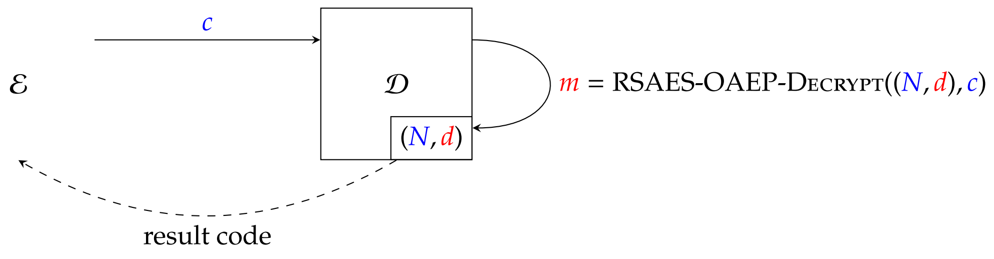
Another way to say the same thing is the error occurs because the output of RSA decryption is too large to fit into one fewer octets than the modulus.

Error #2 : A decryption error occurs in Step 3.g of RSAES-OAEP-Decrypt if an octet string passed into the decoding phase does not

1. produce a hashed label that matches, or
2. include a separating $01_{(16)}$ octet between the (possibly zero length) padding and the message.

Another way to say the same thing is that the error occurs because the plaintext validity checking mechanism is triggered.

A footnote [2, Page 21] motivates the fact that errors (these two in particular) should be indistinguishable from each other: a given application should not tell a user *which* error occurred, only that *some* error occurred. The implementation used by $\mathcal{D}$ does not adhere to this advice, meaning interaction with it can be described as follows:



That is, an attacker $\mathcal{E}$ can (adaptively) send ciphertexts to the attack target $\mathcal{D}$, which will compute their decryption under the fixed, unknown private key (N, d) and produce a result code (i.e., *no* corresponding plaintexts).

4.2 Materials

4.2.1 `${ATTACK}/${USER}/oaep/${USER}.D`

This is an executable program that simulates $\mathcal{D}$. When executed, it reads the following input (one field per-line) from `stdin`

- c, a ciphertext (represented as an octet string),

and writes the following output (again, one field per-line) to `stdout`

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

The program operates in this way, repeatedly reading input and writing output, until it is forcibly terminated (or crashes). In each case, the result code λ 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 (usually 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 $N - 1$.
  - 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 $N - 1$.
  - 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 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 `${ATTACK}/${USER}/oaep/${USER}.public`

This file represents a set of public parameters for the attack: it contains (one field per-line)

- N, a modulus (represented as a hexadecimal integer string),
- e, a public exponent, (represented as a hexadecimal integer string) st. $e \cdot d \equiv 1 \pmod{\Phi N}$, and
- c', a ciphertext (represented as an octet string) corresponding to an encryption of some unknown plaintext m' (represented as an octet string) under the public key (N, e).

More specifically, it contains the public key (N, e) associated with the unknown private key (N, d), plus the challenge ciphertext. You can assume

- the RSAES-OAEP encryption that produced c' from m' used SHA1 as the hash function with a null label (i.e., an octet string of length zero), and
- m' is random, except for the least-significant 4 octets: these are a little-endian, 32-bit, unsigned representation of the integer UNIX User IDentifier (UID)[3] for `${USER}` wrt. the CS lab.

4.3 Goal

Write a program that simulates $\mathcal{E}$ by mounting an attack capable of recovering m', the unknown challenge plaintext associated with the material provided. When executed from a BASH shell using a command of the form

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

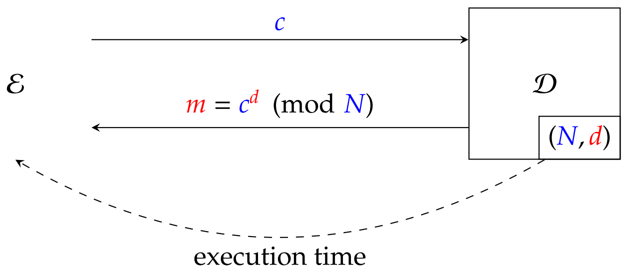
the attack should be invoked on the simulated attack target and public parameters named on the command line (rather than being hard-coded to use a specific attack target or a specific directory, for example). Print both the final result and any relevant intermediate computation to `stdout` (e.g., to keep track of progress); submit *all* source code relating to your attack, and ensure you paste the recovered value of m' (represented as an octet string) into `marksheet.txt`.

[3] http://en.wikipedia.org/wiki/User_identifier

5 Stage 2: an attack based on execution time

5.1 Background

Imagine you are tasked with attacking a given server, denoted $\mathcal{D}$, which houses an Intel Core2 processor. The server is used to accelerate TLS handshakes: it is hard-coded with a fixed program that performs RSA decryption, and used by front-line e-commerce servers whenever RSA-based key exchange is selected. Interaction with $\mathcal{D}$ can be described as follows:



That is, an attacker $\mathcal{E}$ can (adaptively) send ciphertexts to the attack target $\mathcal{D}$, which will compute their decryption under the fixed, unknown private key (N, d) and produce the corresponding plaintexts. Given the context, $\mathcal{E}$ is additionally able to measure the time required by $\mathcal{D}$ to execute each decryption operation.

5.2 Materials

5.2.1 `${ATTACK}/${USER}/time/${USER}.D`

This is an executable program that simulates $\mathcal{D}$. When executed, it reads the following input (one field per-line) from `stdin`

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

and writes the following output (again, one field per-line) to `stdout`

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

The program operates in this way, repeatedly reading input and writing output, until it is forcibly terminated (or crashes). Note that:

- $\mathcal{D}$ does not use the CRT, but does use Coarsely Integrated Operand Scanning (CIOS) Montgomery multiplication [6, Section 5] within a left-to-right binary exponentiation [4, Section 2.1] algorithm. Implementation of both components *exactly* follows the algorithmic description; using the same notation as the latter, this means d has $l + 1$ bits and that $d_l = 1$.
- Since the target houses a 64-bit processor, the implementation will use a base-2^{64} representation of multi-precision integers: since $w = 64$ therefore, the implementation sets $b = 2^w = 2^{64}$.
- To restrict the time required to mount the attack d has been artificially selected so $0 \leq d < b$, but your attack will ideally be valid for values without this restriction as well.

5.2.2 `${ATTACK}/${USER}/time/${USER}.public`

This file represents a set of public parameters for the attack: it contains (one field per-line)

- N, a modulus (represented as a hexadecimal integer string), and
- e, a public exponent (represented as a hexadecimal integer string) st. $e \cdot d \equiv 1 \pmod{\Phi N}$.

More specifically, it contains the public key (N, e) associated with the unknown private key (N, d).

5.2.3 `${ATTACK}/${USER}/time/${USER}.R`

In many side-channel and fault attacks, the attacker is able to perform an initial profiling or calibration phase: one rationale is to select or fine-tune various parameters for the specific attack target. With this in mind, a simulated replica $\mathcal{R}$ of the attack target is also provided. It behaves exactly the same way as $\mathcal{D}$, but accepts input (one field per-line) of the form

- c', a ciphertext (represented as a hexadecimal integer string),
- N', a modulus (represented as a hexadecimal integer string), and
- d', a private exponent (represented as a hexadecimal integer string).

It therefore uses the known private key (N', d') for decryption of c', rather than the unknown private key (N, d). Keep in mind that exponentiation is based on Montgomery multiplication, so will only work correctly *if* you provide an N' st. $\gcd(N', b) = 1$.

5.3 Goal

Write a program that simulates $\mathcal{E}$ by mounting an attack capable of recovering d, the unknown private exponent associated with the material provided. When executed from a BASH shell using a command of the form

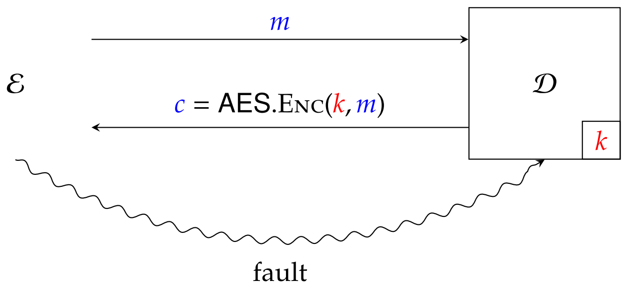
```
bash$ ./attack ${USER}.D ${USER}.public
```

the attack should be invoked on the simulated attack target and public parameters named on the command line (rather than being hard-coded to use a specific attack target or a specific directory, for example). Print both the final result and any relevant intermediate computation to `stdout` (e.g., to keep track of progress); submit *all* source code relating to your attack, and ensure you paste the recovered value of d (represented as a hexadecimal integer string) into `marksheet.txt`.

6 Stage 3: an attack based on an injected fault

6.1 Background

Imagine you are tasked with attacking some device $\mathcal{D}$: the device is a smart-card housing an 8-bit Intel 8051 processor, hard-coded with a fixed program that performs AES encryption. Interaction with $\mathcal{D}$ can be described as follows:



That is, an attacker $\mathcal{E}$ can (adaptively) send plaintexts to the attack target $\mathcal{D}$, which will compute their encryption under the fixed, unknown cipher key k and produce the corresponding ciphertexts. Given the context, $\mathcal{E}$ is additionally able to induce faults during encryption: one fault, which acts to randomises one element of the AES state matrix at a chosen point during each encryption operation performed by $\mathcal{D}$, can be induced.

6.2 Material

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

This is an executable program that simulates $\mathcal{D}$. When executed, it reads the following input (one field per-line) from `stdin`

- λ, a fault specification (represented as a 5-element tuple),
- m, a plaintext (represented as an octet string),

and writes the following output (again, one field per-line) to `stdout`

- c, a 128-bit ciphertext (represented as an octet string).

The program operates in this way, repeatedly reading input and writing output, until it is forcibly terminated (or crashes). Note that:

- $\mathcal{D}$ uses an 8-bit, low-resource implementation of AES-128 with the S-box held as a 256 B look-up table in memory; use of AES-128 clearly implies 128-bit block and cipher key sizes. The implementation matches FIPS-197 [1, Figure 5] exactly: setting $Nb = 4$ and $Nr = 10$ in the notation of [1], it uses a (4×4)-element state matrix and 11 rounds in total (numbered 0 to 10).
- Again with reference to FIPS-197 [1, Figure 5],
  - 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.
- The fault specification λ should be supplied on a single, 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 $0 \leq i, j < 4$.

It is possible to avoid inducing a fault at all by replacing the fault specification with a blank line; this of course yields the correctly encrypted ciphertext for a given plaintext.

- Some *different* fault specifications lead to the *same* outcome. For example, a fault after `AddRoundKey` in the 0-th round leads to the same outcome as a fault before `SubBytes` in the 1-st round.

6.3 Goal

Write a program that simulates $\mathcal{E}$ by mounting an attack capable of recovering k, the unknown cipher key associated with the material provided. When executed from a BASH shell using a command of the form

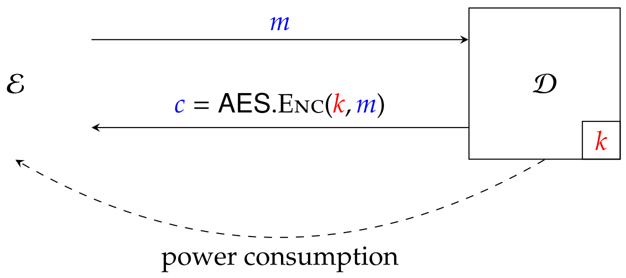
```
bash$ ./attack ${USER}.D
```

the attack should be invoked on the simulated attack target named on the command line (rather than being hard-coded to use a specific attack target or a specific directory, for example). Print both the final result and any relevant intermediate computation to `stdout` (e.g., to keep track of progress); submit *all* source code relating to your attack, and ensure you paste the recovered value of k (represented as an octet string) into `marksheet.txt`.

7 Stage 4: an attack based on power consumption

7.1 Background

Imagine you are tasked with attacking some device $\mathcal{D}$: the device is a smart-card housing an 8-bit Intel 8051 processor, hard-coded with a fixed program that performs AES encryption. Interaction with $\mathcal{D}$ can be described as follows:



That is, an attacker $\mathcal{E}$ can (adaptively) send plaintexts to the attack target $\mathcal{D}$, which will compute their encryption under the fixed, unknown cipher key k and produce the corresponding ciphertexts. Given the context, $\mathcal{E}$ is additionally able to measure the power consumed during each encryption operation: this produces *at least* one sample per instruction executed, due to the high sample rate of an oscilloscope relative to the clock frequency of $\mathcal{D}$.

7.2 Materials

7.2.1 `${ATTACK}/${USER}/power/${USER}.D`

This is an executable program that simulates $\mathcal{D}$. When executed, it reads the following input (one field per-line) from `stdin`

- m, a plaintext (represented as an octet string),

and writes the following output (again, one field per-line) to `stdout`

- λ, a power consumption trace (represented as an l-element vector), and
- c, a ciphertext (represented as an octet string).

The program operates in this way, repeatedly reading input and writing output, until it is forcibly terminated (or crashes). Note that:

- $\mathcal{D}$ uses an 8-bit, low-resource implementation of AES-128 with the S-box held as a 256 B look-up table in memory; use of AES-128 clearly implies 128-bit block and cipher key sizes. The implementation matches FIPS-197 [1, Figure 5] exactly: setting $Nb = 4$ and $Nr = 10$ in the notation of [1], it uses a (4×4)-element state matrix and 11 rounds in total (numbered 0 to 10).
- Each power consumption trace λ is a single, 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 of which represented as an 8-bit, unsigned decimal integer string): in short, this means $0 \leq s_i < 256$ for $0 \leq i < l$.

7.2.2 `${ATTACK}/${USER}/power/${USER}.R`

In many side-channel and fault attacks, the attacker is able to perform an initial profiling or calibration phase: one rationale is to select or fine-tune various parameters for the specific attack target. With this in mind, a simulated replica $\mathcal{R}$ of the attack target is also provided. It behaves exactly the same way as $\mathcal{D}$, but accepts input (one field per-line) of the form

- m', a plaintext (represented as an octet string), and
- k', a cipher key (represented as an octet string).

It therefore uses the known cipher key k' for encryption of m', rather than the unknown cipher key k.

7.3 Goal

Write a program that simulates $\mathcal{E}$ by mounting an attack capable of recovering k, the unknown cipher key associated with the material provided. When executed from a BASH shell using a command of the form

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

the attack should be invoked on the simulated attack target named on the command line (rather than being hard-coded to use a specific attack target or a specific directory, for example). Print both the final result and any relevant intermediate computation to `stdout` (e.g., to keep track of progress); submit *all* source code relating to your attack, and ensure you paste the recovered value of k (represented as an octet string) into `marksheet.txt`.

8 Stage 5: analysis

For each attack in the previous stages, a subtle difference exists between simple implementation and genuine understanding of the underlying theory. The goal of this stage is to assess the latter, and, in particular, reward more advanced understanding which is difficult to demonstrate via source code alone.

`${ATTACK}/${USER}/analysis/${USER}.txt` is an ASCII text file containing questions that relate to the previous stages. This stage has a single, simple requirement: clearly and concisely answer as many of the questions as you can. To do so,

1. provide your answers within the same file (i.e., use plain text, rather than PDF or similar), and
2. insert the answer text directly below the associated question.

Although $Q.i.j$ denotes the j-th question relating to the i-th stage, the questions may *not* be numbered sequentially: $Q.2.2$ might appear before $Q.2.1$, or perhaps $Q.2.2$ is missing for example. This fact is irrelevant, provided you follow the guidelines above. All questions are weighted equally: if there are n questions in total, each one is worth $1/n$ of the marks for this stage.

References

[1] Federal Information Processing Standards (FIPS). Specification for the Advanced Encryption Standard (AES), FIPS-197, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[2] RSA Laboratories. PKCS#1 v2.1 : RSA Cryptography Standard, 3-rd Draft, 2002. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1d3.pdf

[3] M. Bellare and P. Rogaway. Optimal Asymmetric Encryption. In *Advances in Cryptology (EUROCRYPT)*, Springer-Verlag LNCS 950, 92–111, 1994.

[4] D.M. Gordon. A Survey of Fast Exponentiation Methods. In *Journal of Algorithms*, **27** (1), 129–146, 1998.

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

[6] Ç.K. Koç, T. Acar and B.S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. In *IEEE Micro* **16** (3), 26–33, 1996.

[7] B. Schneier. Applied Cryptography, 2nd ed. John Wiley & Sons, 2006. http://www.schneier.com/book-applied.html

A Representation and conversion

Somewhat bizarrely, one of the most confusing aspects of the assignment is arguably understanding how to correctly provide input and interpret output. This is crucial because values which *look* similar as human-readable strings may be interpreted differently depending on their machine-readable representation. To demonstrate the conversion to and from the pertinent data types, we use examples written in Python. Although they can be translated to other languages, use of Python is motivated by two features:

1. programs are typically executed interactively via an interpreter, allowing quick (re)evaluation and experimentation, and
2. such interpreters are widely available.

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

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

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

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

  whereas

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

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

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

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

  and

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

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

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 $(|b| - 1 - i)$-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

$$\begin{array}{lllllllllllllllll} \hat{x} & \mapsto & 09_{(16)} \cdot 16^9 & + & 08_{(16)} \cdot 16^8 & + & 07_{(16)} \cdot 16^7 & + & 06_{(16)} \cdot 16^6 & + & 05_{(16)} \cdot 16^5 & + \\ & & 04_{(16)} \cdot 16^4 & + & 03_{(16)} \cdot 16^3 & + & 02_{(16)} \cdot 16^2 & + & 01_{(16)} \cdot 16^1 & + & 00_{(16)} \cdot 16^0 \\ & \mapsto & \multicolumn{11}{l}{42649378395939397566720_{(10)}} \end{array}$$

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

$$\begin{array}{lllllllllllllllll} \hat{y} & \mapsto & 09_{(16)} \cdot 16^9 & + & 08_{(16)} \cdot 16^8 & + & 07_{(16)} \cdot 16^7 & + & 06_{(16)} \cdot 16^6 & + & 05_{(16)} \cdot 16^5 & + \\ & & 04_{(16)} \cdot 16^4 & + & 03_{(16)} \cdot 16^3 & + & 02_{(16)} \cdot 16^2 & + & 01_{(16)} \cdot 16^1 & + & 0_{(16)} \cdot 16^0 \\ & \mapsto & \multicolumn{11}{l}{42649378395939397566720_{(10)}} \end{array}$$

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

$$\begin{array}{lllllllllllllllll} \hat{z} & \mapsto & 00_{(16)} \cdot 16^9 & + & 90_{(16)} \cdot 16^8 & + & 80_{(16)} \cdot 16^7 & + & 70_{(16)} \cdot 16^6 & + & 60_{(16)} \cdot 16^5 & + \\ & & 50_{(16)} \cdot 16^4 & + & 40_{(16)} \cdot 16^3 & + & 30_{(16)} \cdot 16^2 & + & 20_{(16)} \cdot 16^1 & + & 10_{(16)} \cdot 16^0 \\ & \mapsto & \multicolumn{11}{l}{2665586149746212347920_{(10)}} \end{array}$$