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

Applied Cryptology (COMS30048)

Assessed coursework assignment

AttackHW

Note that:

1. This coursework assignment has a 100 percent weighting, i.e., it represents 100 percent of Credit Points (CPs) associated with COMS30048, and is assessed on an individual basis. The submission deadline is 02/05/24.
2. Before you start work, ensure you are aware of and/or adhere to various regulations[a] which govern coursework assessments: pertinant examples include those related to academic integrity (see, e.g., Sec. 3) and submission (see, e.g., Sec. 12).
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 coursework 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 gain a deeper understanding of a) implementation challenges for given cryptographic primitives, b) attacks against said implementations, and c) countermeasures against said attacks, all through applied research and development tasks set within a motivating, example scenario.

2 Terms and conditions

- The assignment description may refer to the ASCII text file `question.txt`, or more generally "the marksheet": download this file from

  http://tinyurl.com/5h3tux8s/csdsp/cw/AttackHW/question.txt

  then complete and include it in your submission. This is important, in the sense that 1) it offers *you* clarity with respect to the marking process, e.g., via a marking scheme, and 2) it offers *us* useful (meta-)information about your solution. 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 (versus relying exclusively on the teaching material provided), and clearly documented (e.g., using the marksheet).

- The assignment design includes some heavily supported, closed initial stages which reflect a lower mark, and some mostly unsupported, open later stages which reflects a higher mark. This suggests the marking scale is non-linear: it is clearly easier to obtain X marks in the initial stages than in the final stage. The term open (resp. closed) should be understood as meaning flexibility with respect to options for work, *not* non-specificity with respect to workload: each stage has a clear success criteria that limit the functionality you implement, meaning you can (and should) stop work once they have been satisfied.

- As was outlined in the lab. worksheets, the SCALE kits are *only* available in the lab. slots. Our rationale is that doing so acts to control the amount of time you invest in the assignment, and so your overall workload. *Please* keep this in mind: this approach may not suit everyone, but is carefully considered and well intended.

- 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 associated ILOs. Viable examples include OpenSSL for C, and the `pycryptodome` 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.

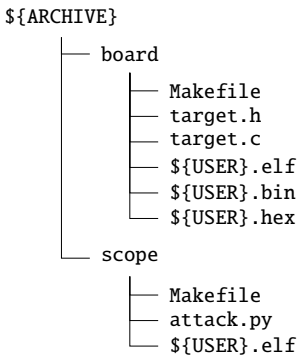


Figure 1: *A diagrammatic description of the material in* `${USER}.tar.gz`.

- Implementations produced as part of the assignment will be marked 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

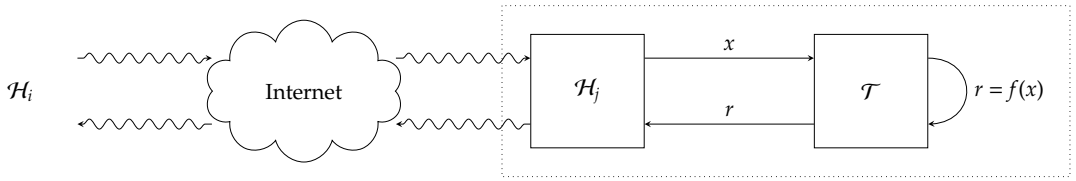
Selected material, personalised on a per student basis, is provided for you to use. Assuming `${USER}` is used to represent your 7-digit UoB student number, download[1] and unarchive the file

`http://tinyurl.com/5h3tux8s/csdsp/cw/AttackHW/${USER}.tar.gz`

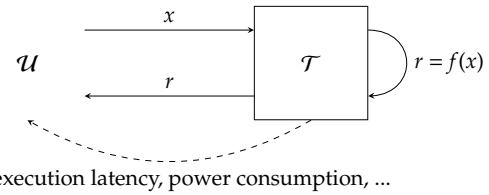
somewhere secure in your file system: from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content illustrated by Figure 1.

- `${ARCHIVE}/board/target.[ch]` provides a skeleton attack target implementation (i.e., some source code, written in C), which relates to stage 1.
- `${ARCHIVE}/scope/attack.py`; provides a skeleton attack implementation (i.e., some source code, written in Python), which relates to stage 2.
- `${ARCHIVE}/board/${USER}.elf` (plus `${ARCHIVE}/board/${USER}.bin` and `${ARCHIVE}/board/${USER}.hex`, which are derived from it) provides a compiled attack target implementation (i.e., an executable for a SCALE development board), which relates to stage 2; it is expanded upon in Appendix A.
- `${ARCHIVE}/scope/${USER}.elf` provides a compiled attack implementation (i.e., an executable for a control workstation), which relates to stages 2 and 3; it is expanded upon in Appendix B. The 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 platform, and, even then, only once the file has appropriate permissions set using `chmod`.
- `${ARCHIVE}/board/Makefile` and `${ARCHIVE}/scope/Makefile` are fairly self explanatory, acting as analogues to the files encountered in lab. worksheet #1.1 and #1.2. The former, for example, includes a build system for the skeleton attack target implementation.

[1] If your 7-digit UoB student numberis 0123456, for example, the corresponding URL would be `http://tinyurl.com/5h3tux8s/csdsp/cw/AttackHW/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.



(a) *A more abstract description of the scenario.*



(b) *A less abstract description of the scenario.*

Figure 2: *A diagrammatic description of the scenario considered.*

3.2 Overview

Consider an example scenario, where you join the development team for a device $\mathcal{T}$; this scenario is described diagrammatically in Figure 2a. The device is design to act as a co-processor which supports a range of security-related functionality. For example, using a standard protocol it includes support for 1) secure key generation and storage, and 2) off-load of cryptographic operations. This allows it to support some host system $\mathcal{H}_j$, which engages with TLS-based communication across the Internet, with a remote system $\mathcal{H}_i$.

To minimise cost, no bespoke hardware is used: all the functionality is realised in software, or rather firmware, as executed by the integrated micro-controller. For convenience, we use $\mathcal{T}$ to denote the *whole* device from here on, i.e., the hardware plus software, including firmware, executed on it. The development of an initial prototype has just begun, with a focus on delivery of functionality related to AES-128. This means the prototype $\mathcal{T}$ 1) has non-volatile, secure storage for an AES-128 cipher key $\hat{k}$, and 2) can compute AES-128 encryption operations using it. That is, in each interaction with it, a given user $\mathcal{U}$ can send an AES-128 plaintext to $\mathcal{T}$; the device encrypts that plaintext under $\hat{k}$, and sends back the corresponding ciphertext. Due to the context in which $\mathcal{T}$ will be deployed, there is some concern about the threat of implementation attacks against it; DPA-like attacks have been specifically highlighted as a threat. You have been tasked with ensuring the AES-128 implementation used is secure against such attacks, while *also* being efficient enough.

This assignment models aspects of the scenario outlined above, and, in particular, a case described by Figure 2b in which $\mathcal{U}$ has physical access to $\mathcal{T}$. Replicating broader challenges with respect to cryptographic engineering, and using the SCALE kit (per lab. worksheets #1.1 and #1.2) as a vehicle to do so, it tasks you with developing a) an attack target implementation (i.e., an AES-128 implementation) modelling the firmware for $\mathcal{T}$, executed on a SCALE development board, and b) an attack implementation modelling a malicious, or adversarial $\mathcal{U}$, executed on a control workstation (using a PicoScope 2206B to acquire traces of power consumption from a SCALE development board). The protocol used to communication between the modelled $\mathcal{U}$ and $\mathcal{T}$ is detailed in Appendix C.

3.3 Detail

Stage 1. This stage involves development of an attack target implementation, modelling $\mathcal{T}$ per Section 3.2. As a starting point you *must* use the skeleton attack target implementation provided. *Do not* alter the `main`[2] function: instead, produce a solution by implementing a) the placeholder functions `octetstr_rd` and `octetstr_wr` (per lab. worksheet #1.1), and b) the placeholder functions `aes` and `aes_init` (per lab. worksheet #2).

Success criteria. Demonstrate that the attack target implementation is

(a) *correct*, i.e., given the cipher key $\hat{k}$ hard-coded in the skeleton, it correctly computes the ciphertext $c = \mathsf{AES.Enc}(\hat{k}, m)$ for a given plaintext m,

(b) *efficient*, i.e., yields said ciphertext within a 1ms "efficiency budget" (or time limit).

[2]The `main` function within the skeleton implements the a) communication protocol and b) trigger signal management, required to support interaction with the implementation (e.g., within later stages): if you alter it, said interaction may fail. Likewise, it includes a hard-coded AES-128 key `k` that differs per student: if you alter it, verifying your implementation works correctly is more difficult.

Advice. A sensible approach would be to first develop, e.g., an AES-128 implementation, independently and then, only once you are confident that it works as intended, port it to the development board: doing so will likely render development, debugging in particular, *significantly* easier and quicker.

Advice. As the documentation (i.e., comments in the source code) states, the `aes_init` function allows initialisation before an associated encryption operation is then performed via the `aes` function: examples include expansion of a cipher key into round keys. If your solution requires no initialisation, `aes_init` can be ignored.

Advice. At this stage, your solution can and should ignore the argument `r` to `aes_init` and `aes`: this only becomes relevant if/when you attempt stage 3.

Stage 2. This stage involves development of an attack implementation, modelling $\mathcal{U}$ per Section 3.2: your solution should demonstrate the recovery of $\hat{k}$ by $\mathcal{U}$, via analysis of power consumption traces acquired during execution of `aes` by $\mathcal{T}$. As a starting point you *could* use the skeleton attack implementation provided, although a) you are free not to, and b) can use a programming language of your choice. There are two classes of valid solution that *you* can select between; since the former (resp. latter) is easier (resp. harder), it is weighted less (resp. more) with respect to the mark scheme.

(a) In an assisted (or dependent) solution, the idea is to a) use the compiled attack implementation provided to acquire and store a trace data set, then b) load and use said trace data set as input to your solution.

Success criteria. Demonstrate a successful key recovery attack, executing it using a command similar to

```
./attack ${FILE}
```

where the mandatory (file name) command-line argument `${FILE}` specifies the trace data set to use as input.

(b) In an unassisted (or independent) solution, the idea is your solution is standalone, meaning that it can acquire then use a trace data set itself.

Success criteria. Demonstrate a successful key recovery attack, executing it using a command similar to

```
./attack
```

For either class of solution, ensure the attack output *clearly* reports both the a) recovered key (represented as an octet string), and b) relevant metrics (e.g., the number of traces used).

Advice. A sensible approach would be to work step-by-step, first developing an assisted solution and then extending it to form an unassisted solution: doing so a) employs best-practice with respet to incremental development, plus b) decouples development of the attack itself from the challenge of using a 2206B, plus dependency on the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11); having acquired a trace data set *in* the lab. one could develop the attack itself *anywhere*, for example.

To support this approach and *further* decouple work on the attack from the 2206B, you can download an example (compressed) trace data set from

```
http://tinyurl.com/5h3tux8s/csdsp/cw/AttackHW/stage2.dat.gz
```

An attack target implementation equivalent to the one provided (i.e., `${USER}.elf`) was used to produce the data set; if your attack implementation works using it as input, you can be confident it will work more generally (i.e., against `${USER}.elf` itself, noting of course that the cipher key will differ).

Advice. In *theory*, your attack implementation will be applicable to *both* a) the compiled attack target implementation provided, *and* b) your solution to stage 1. In *practice*, however, it suffices to demonstrate your attack implementation against the former. Fixing this remit produces a more uniform challenge (i.e., it is the same for all students, irrespective of their approach in stage 1), and also allows independent progress with stage 2 (i.e., it removes the need to complete stage 1 first).

Advice. Although the functional correctness of your attack implementation is obviously crucial, various additional criteria 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

(a) self-contained, in the sense it requires no input from the user,

(b) robust, in the sense it produces the correct result *every* time (not just sometimes),

(c) generic, in the sense it produces the correct result for *any* material (not just your own), and

(d) efficient.

Stage 3. This stage involves development of a countermeasure within the previous attack target implementation (from stage 1), intended to secure (or a least "harden") it against the previous attack implementation (from stage 2).

Success criteria. Demonstrate that the attack target implementation is

(a) *correct*, i.e., given the cipher key $\hat{k}$ hard-coded in the skeleton, it correctly computes the ciphertext $c = \mathsf{AES}.\textsc{Enc}(\hat{k}, m)$ for a given plaintext m,

(b) *efficient*, i.e., yields said ciphertext within a 100ms "efficiency budget" (or time limit),

(c) *secure*, i.e., it is able to prevent the attack implementation recovering $\hat{k}$.

Produce a document which precisely specifies plus clearly explains and justifies the countermeasure you used, paying particular attention to the following points:

(a) the underlying design

(b) the implementation of said design, and

(c) an analysis of the trade-off between expected improvement in security and other metrics (e.g., latency, area), and a set of assumptions this depends on.

The document should be a PDF, of at most 2 pages (excluding the bibliography and any figures), named `stage3.pdf`; You may include extra content in appendices beyond the 2-page limit, e.g., technical detail which is useful but not crucial with respect to addressing the points above, but keep in mind this may not be read nor assessed.

Advice. Note that the success criteria for this stage subsume those for stage 1. That is, there is no *requirement* to retain the (insecure) solution for stage 1 *if* you submit a (secure) solution for stage 3.

Advice. Although analysis and identification of suitable countermeasures forms part of the assessment, most options will require a source of randomness: per Appendix C, the argument `r` to `aes_init` and `aes` provides such a source. Ensure you change `SIZEOF_RND` (in `target.h`) to suit the requirements of your solution, so the skeleton attack target implementation can then a) advertise the correct requirement to $\mathcal{U}$, then both b) allocate enough space for, and read `r` correctly in `main`.

Advice. It is reasonable to interpret "is able to prevent" as "could *plausibly* prevent" rather than "does *actually* prevent". One reason for making this distinction is that various practical limitations may prevent you *actually* executing the attack: a central example is the amount of physical memory available on the lab. workstations, which limits the number of traces and/or samples per trace and so efficacy of the attack implementation.

Stage 4. This stage involves development of a written specification. Although the initial prototype $\mathcal{T}$ focused on support for AES-128 alone, the *final* $\mathcal{T}$ is intended to support a much broader suite of functionality. In particular, imagine the use-case for $\mathcal{T}$ is support of TLS-based communication (between the host system and some remote server); the context is the same, in that implementation (e.g., side-channel and fault) attacks are pertinent in a broad sense.

Success criteria. Produce a document which (re)designs the final $\mathcal{T}$, i.e., precisely specifies plus clearly explains and justifies

(a) the functionality supported,

(b) the implementation strategy which should be adopted for said functionality, and

(c) how the host system accesses said functionality via the API, e.g., via a suitable analogue of Section C,

aligning each point with requirements of the use-case and context.

The document should be a PDF, of at most 4 pages (excluding the bibliography and any figures), named `stage4.pdf`; You may include extra content in appendices beyond the 4-page limit, e.g., technical detail which is useful but not crucial with respect to addressing the points above, but keep in mind this may not be read nor assessed.

Advice. Keep in mind that although it is important to *consider* the implementation of your design, there is no need to implement it: you could think of this stage as a thought experiment[3] therefore, the result of which is the specification alone.

[3] https://en.wikipedia.org/wiki/Thought_experiment

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. 9).

[2] E. Brier, C. Clavier, and F. Olivier. "Correlation Power Analysis with a Leakage Model". In: *Cryptographic Hardware and Embedded Systems (CHES)*. LNCS 3156. Springer-Verlag, 2004, pp. 16–29 (see p. 10).

[3] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer, 2002 (see p. 9).

A An attack target implementation: `${ARCHIVE}/board/${USER}.elf`

A.1 Remit

The compiled attack target implementation provided offers a minimal exemplar that, by design, is 1) correct, but 2) inefficient with respect to time, and insecure with respect to side-channel attack. Although it is common to assume[4] an attacker has *full* access to the design (e.g., to the source code) of their target, doing so conflicts with the assignment because stage 1 tasks you with developing it yourself! Instead, sufficient *partial* access is captured by the following details:

- Given that it constitutes an implementation of AES-128, it uses 128-bit block and cipher key lengths (meaning a 16-element byte sequence, resp. octet string, representation).
- Following the notation in [1, Figure 5], the fact that $Nb = 4$ and $Nr = 10$ means a (4×4)-element state matrix will be manipulated in a total of 11 rounds.
- [1, Section 6.4] refers to [3] with respect to implementation strategy. Given the platform used to execute it, the implementation follows [3, Section 4.1] more or less verbatim: doing so a) uses an 8-bit data-path (i.e., it performs operations on 8-bit bytes used to represent elements of both the state and round key matrices), and b) makes a trade-off that favours low memory footprint over low latency (resp. high throughput). More specifically:
  1. a 256 B look-up table in memory is used to store pre-computed values of the S-box (as used, e.g., in the `SubBytes` round function),
  2. a 256 B look-up table in memory is used to store pre-computed values of `xtime` (as used, e.g., in the `MixColumns` round function),
  3. the round keys required are not pre-computed: each encryption operation takes the cipher key and evolves it forward, step-by-step, to form successive round keys.

A.2 Usage

You can use the compiled attack target implementation (i.e, program a SCALE development board with it) as follows:

- Fix the working directory:

  ```
  cd ${ARCHIVE}/board
  ```

- Initiate the programming process using

  ```
  make PROJECT="${USER}" program
  ```

  to perform the first step, and then perform each of the subsequent (manual) steps.

[4] Per Kerckhoffs's principle, in contrast to "security through obscurity"; see, e.g., http://en.wikipedia.org/wiki/Kerckhoffs's_principle

B An attack implementation: `${ARCHIVE}/scope/${USER}.elf`

B.1 Remit

The compiled attack implementation provided offers a minimal exemplar that, by design, is a) limited with respect to the constituent techniques used, and thus b) limited with respect to guarantees of performance and effectiveness. The attack makes use of Correlation Power Analysis (CPA) [2]: it targets the `SubBytes` round function in the first AES-128 round (i.e., the operation S-box($m_i \oplus k_i$), for some known plaintext m and unknown cipher key k), and assumes a Hamming weight leakage model.

B.2 Usage

You can use the compiled attack implementation (i.e., mount said attack against an attack target implementation) as follows:

- Fix the working directory:

  ```
  cd ${ARCHIVE}/scope
  ```

- Execute

  ```
  ./${USER}.elf --help
  ```

  to assess the set of command-line options available, which act to control *what* the attack does and *how* it does so.

- Execute the attack itself, either

  1. manually

     ```
     ./${USER}.elf
     ```

     or
  2. automatically (via `Makefile`)

     ```
     make attack
     ```

  noting that to prevent it being overly agressive, it will abort if various limits (e.g., target implementation execution time, number of samples, or memory footprint) are exceeded.

As a benchmark, keep in mind that when used against the attack target implementation provided (using the default parameters per the above), the attack implementation will a) take roughly 5min to complete, including both acquisition and analysis phases, and b) produce an uncompressed trace data set of roughly 160MB in size.

B.3 Trace data set format

Consider a data set which captures n traces each of l samples; let $\Lambda_{i,j}$ denote the j-th sample of the i-th such trace, for $0 \leq i < n$ and $0 \leq j < l$. The traces will have been acquired with respect to AES-128 encryption operations, each of which uses a (known) plaintext as input and produces a (known) ciphertext as output; let $M_{i,j}$ (resp. $C_{i,j}$) denote the j-th byte of the i-th such plaintext (resp. ciphertext) for $0 \leq i < n$ and $0 \leq j < 16$. With this in mind, the (binary) trace data set format is illustrated by Figure 3:

- If M, C, and Λ are consider as matrices, their elements are stored in row-major (or trace-major) order.
- The data types involved stem, in part, from the PicoScope API; as expressed using C, they are as follows

  $$\begin{array}{lcl} n & \leadsto & \texttt{uint32\_t} \\ l & \leadsto & \texttt{uint32\_t} \\ M_{i,j} & \leadsto & \texttt{uint8\_t} \\ C_{i,j} & \leadsto & \texttt{uint8\_t} \\ \Lambda_{i,j} & \leadsto & \texttt{int16\_t} \end{array}$$

  Keep in mind that use of `int16_t` for each sample, i.e., each $\Lambda_{i,j}$, matches `ps2000aSetDataBuffer` and `ps2000aGetValues` from the C binding and `getDataRaw` from the Python binding; as a result, they also match lab. worksheet #1.2.

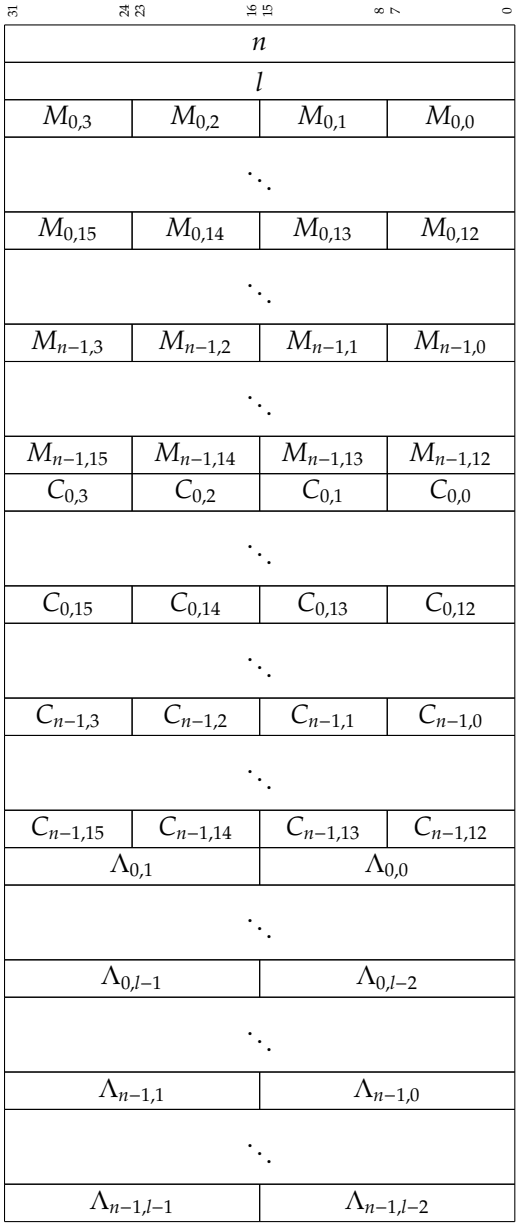


Figure 3: *A diagrammatic description of the (binary) trace data set format used by the attack implementation* `${ARCHIVE}/scope/${USER}.elf`.

C Communication between $\mathcal{U}$ and $\mathcal{T}$

C.1 Low-level representation

A textual (vs. binary) representation is used for *all* communication between $\mathcal{U}$ and $\mathcal{T}$. Although a simplification[5] intended to limit the assignment scope, doing so allows human users to easily interact with and so test an implementation of $\mathcal{T}$. A short explanation is that the representation is based on octet strings per lab. worksheet #1.1, with an assumption that the same EOL semantics are adhered to; a long explanation follows, repeated for completeness.

C.1.1 Representation of byte sequences using (hexadecimal) octet strings

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 $x = \langle 00_{(16)}, 01_{(16)}, \ldots, 0F_{(16)} \rangle = \langle 0_{(10)}, 1_{(10)}, \ldots, 15_{(10)} \rangle \equiv \mathtt{x}$.

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

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

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

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

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

[5] In reality, a binary representation can be more compact *and* easier to parse; it is more efficient in space *and* time. There are also disadvantages, but these properties would often make it more attractive in scenarios such as the example, cf. a real protocol using a smart-card Application Protocol Data Unit (APDU) standard; see, e.g., http://en.wikipedia.org/wiki/Smart_card_application_protocol_data_unit.

[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

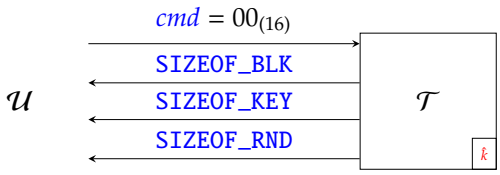


Figure 4: *Communication protocol for the inspect command.*

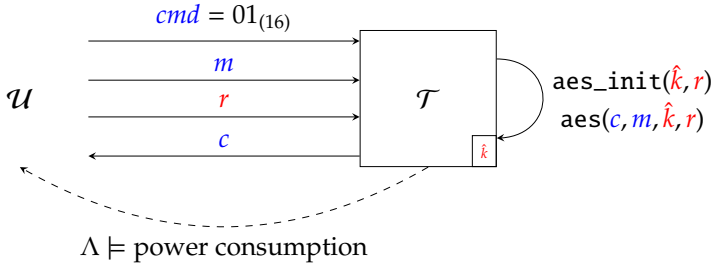


Figure 5: *Communication protocol for the encrypt command.*

C.1.2 UART communication and EOL semantics

The concept of the End Of Line (EOL) character (aka. newline[13]) *seems* trivial, and, in theory, is: in essence it is a control character we expect to be associated with pressing a return (or enter) key. In practice, however, the control character, or characters, used will differ based on various factors. The most obvious example is the use of Carriage Return (CR), i.e., the byte $0D_{(16)}$ (or C escape character `'\r'`), and/or Line Feed (LF), i.e., the byte $0A_{(16)}$ (or C escape character `'\n'`), characters. Note that much of the terminology[14] stems from (electronic) typewriters. For example, CR moves the type element (or cursor) to the start of the *same* line, whereas LF moves the type element to the same position on the *next* line; in combination (i.e., CR+LF) realises what we normally consider to be a new line (or express verbally as "start a new line").

As such, different EOL semantics are possible: for example a Linux will typically use LF, whereas Windows will typically use CR+LF. You may have already *observed* this difference, when extra control characters appear in a text file (e.g., C source code) first written on a Windows-based platform then transferred to a Linux-based alternative. The same difference is important when engaging in serial communication, e.g., with a given development board. Although not complicated, this *does* need some care. Arguably the easiest, and hence recommended approach is to use the same EOL semantics as PuTTY:

1. By default, PuTTY emulates a VT100 terminal[15]. This means pressing the return key will transmit CR.
2. Match those semantics in your implementation. For example, one might read a line of input by consuming characters until a CR is encountered; at this point, the CR is "eaten" (or discarded) and the line deemed complete.
3. When receiving, PuTTY can be configured so it injects an implicit LF and/or CR. This can be useful, since receiving CR without LF, for example, can induce (visually) odd behaviour in the terminal (per a typewriter, lack of LF produces "overwritten" text).
4. By default the `pyserial` function `readline` waits for a LF to mark the EOL, so a CR-based alternative means taking an alternative approach. Viable approaches include a) writing a bespoke `readline` replacement or b) using the `TextIOWrapper` wrapper, which allows an explicit selection of EOL semantics.

C.2 High-level communication

The high-level communication between $\mathcal{U}$ and $\mathcal{T}$ can be described using a general 4-step protocol, namely

[13] http://en.wikipedia.org/wiki/Newline
[14] See, e.g., http://en.wikipedia.org/wiki/Carriage_return
[15] http://en.wikipedia.org/wiki/VT100

1. $\mathcal{U}$ sends a command to $\mathcal{T}$,
2. $\mathcal{U}$ sends input (if any) to $\mathcal{T}$,
3. $\mathcal{T}$ performs some computation,
4. $\mathcal{T}$ sends output (if any) to $\mathcal{U}$.

where each step constitutes the communication of data represented as an octet string. The protocol is specialised depending on the command, of which there are two: the commands are described in detail below, noting that Figure 6 offers a concrete example.

- The inspect command exposes any parameters used by $\mathcal{T}$, notably those related to the AES-128 implementation it uses, to $\mathcal{U}$. The associated protocol is illustrated in Figure 4, and can be described as follows:
  - $\mathcal{U}$ sends a command $cmd = 00_{(16)}$ (a sequence of 1 byte) to $\mathcal{T}$,
  - $\mathcal{T}$ sends `SIZEOF_BLK` (a sequence of 1 byte) to $\mathcal{U}$,
  - $\mathcal{T}$ sends `SIZEOF_KEY` (a sequence of 1 byte) to $\mathcal{U}$,
  - $\mathcal{T}$ sends `SIZEOF_RND` (a sequence of 1 byte) to $\mathcal{U}$.

  Note that use of AES-128 means $\mathcal{U}$ will expect (and so can assume) `SIZEOF_BLK` = `SIZEOF_KEY` = 16, whereas the value of `SIZEOF_RND` is determined by $\mathcal{T}$: the default is `SIZEOF_RND` = 0.
- The encrypt command instructs $\mathcal{T}$ to execute an AES-128 encryption operation on behalf of $\mathcal{U}$. The associated protocol is illustrated in Figure 5, and can be described as follows:
  - $\mathcal{U}$ sends a command $cmd = 01_{(16)}$ (a sequence of 1 byte) to $\mathcal{T}$,
  - $\mathcal{U}$ sends an AES-128 plaintext m (a sequence of `SIZEOF_BLK` bytes) to $\mathcal{T}$,
  - $\mathcal{U}$ sends some randomness r (a sequence of `SIZEOF_RND` bytes) to $\mathcal{T}$,
  - $\mathcal{T}$ executes the AES implementation, i.e., invokes `aes_init`$(\hat{k}, r)$ then `aes`$(c, m, \hat{k}, r)$, using an AES-128 cipher key $\hat{k}$ to encrypt the plaintext m (and thereby compute the associated ciphertext c),
  - $\mathcal{T}$ sends an AES-128 ciphertext c (a sequence of `SIZEOF_BLK` bytes) to $\mathcal{U}$.

  Note that the provision of randomness by $\mathcal{U}$ to $\mathcal{T}$ demands some explanation. The purpose of r is essentially to support implementation of any (randomised) countermeasures; the reason $\mathcal{U}$ supplies it as input, vs. $\mathcal{T}$ generating it, is a simplification[16] to limit the assignment scope. Based on the above, it is *crucial* that $\mathcal{U}$ makes use of the inspect command to recover `SIZEOF_RND`: this determines the amount of randomness required by $\mathcal{T}$, and so the form required of r in each encrypt command.

[16] In reality, it makes no sense for $\mathcal{T}$ to trust $\mathcal{U}$: it could be, and in this example scenario *is*, an adversary! It could, for example, act adversarially by attempting to control (e.g., provide an *non*-random value for) or use (e.g., tailor an attack to capitalise on the known value of) r somehow. Any real $\mathcal{T}$ would therefore generate r internally somehow, typically via the use of a (T)RNG of some sort.

Direction	Content	Meaning
$\mathcal{U} \rightarrow \mathcal{T}_0$	`01:00`	$cmd = 00_{(16)} \mapsto$ inspect
$\mathcal{T}_0 \rightarrow \mathcal{U}$	`01:10`	SIZEOF_BLK = 16
$\mathcal{T}_0 \rightarrow \mathcal{U}$	`01:10`	SIZEOF_KEY = 16
$\mathcal{T}_0 \rightarrow \mathcal{U}$	`01:00`	SIZEOF_RND = 0
$\mathcal{U} \rightarrow \mathcal{T}_0$	`01:01`	$cmd = 01_{(16)} \mapsto$ encrypt
		$m = \left\langle \begin{array}{cccc} 00_{(16)}, & 01_{(16)}, & 02_{(16)}, & 03_{(16)}, \\ 04_{(16)}, & 05_{(16)}, & 06_{(16)}, & 07_{(16)}, \\ 08_{(16)}, & 09_{(16)}, & 0A_{(16)}, & 0B_{(16)}, \\ 0C_{(16)}, & 0D_{(16)}, & 0E_{(16)}, & 0F_{(16)} \end{array} \right\rangle$
$\mathcal{U} \rightarrow \mathcal{T}_0$	`10:000102030405060708090A0B0C0D0E0F`	
		$r = \langle\rangle$
$\mathcal{U} \rightarrow \mathcal{T}_0$	`00:`	
		$c = \left\langle \begin{array}{cccc} AC_{(16)}, & 91_{(16)}, & 18_{(16)}, & 64_{(16)}, \\ 1B_{(16)}, & C8_{(16)}, & 54_{(16)}, & C8_{(16)}, \\ 52_{(16)}, & 08_{(16)}, & 96_{(16)}, & 2A_{(16)}, \\ FC_{(16)}, & 0B_{(16)}, & EB_{(16)}, & 85_{(16)} \end{array} \right\rangle$
$\mathcal{T}_0 \rightarrow \mathcal{U}$	`10:AC9118641BC854C85208962AFC0BEB85`	
$\mathcal{U} \rightarrow \mathcal{T}_1$	`01:00`	$cmd = 00_{(16)} \mapsto$ inspect
$\mathcal{T}_1 \rightarrow \mathcal{U}$	`01:10`	SIZEOF_BLK = 16
$\mathcal{T}_1 \rightarrow \mathcal{U}$	`01:10`	SIZEOF_KEY = 16
$\mathcal{T}_1 \rightarrow \mathcal{U}$	`01:10`	SIZEOF_RND = 16
$\mathcal{U} \rightarrow \mathcal{T}_1$	`01:01`	$cmd = 01_{(16)} \mapsto$ encrypt
		$m = \left\langle \begin{array}{cccc} A7_{(16)}, & C6_{(16)}, & DD_{(16)}, & FC_{(16)}, \\ FA_{(16)}, & FA_{(16)}, & 4B_{(16)}, & A8_{(16)}, \\ F9_{(16)}, & 19_{(16)}, & A1_{(16)}, & B3_{(16)}, \\ B5_{(16)}, & DA_{(16)}, & 1C_{(16)}, & 2C_{(16)} \end{array} \right\rangle$
$\mathcal{U} \rightarrow \mathcal{T}_1$	`10:A7C6DDFCFAFA4BA8F919A1B3B5DA1C2C`	
		$r = \left\langle \begin{array}{cccc} E6_{(16)}, & 81_{(16)}, & 49_{(16)}, & 38_{(16)}, \\ 66_{(16)}, & 10_{(16)}, & 14_{(16)}, & A1_{(16)}, \\ 34_{(16)}, & E3_{(16)}, & 22_{(16)}, & 43_{(16)}, \\ 7E_{(16)}, & 00_{(16)}, & 89_{(16)}, & 31_{(16)} \end{array} \right\rangle$
$\mathcal{U} \rightarrow \mathcal{T}_1$	`10:E68149386610144134E322437E008931`	
		$c = \left\langle \begin{array}{cccc} EE_{(16)}, & 73_{(16)}, & A8_{(16)}, & 73_{(16)}, \\ 02_{(16)}, & F6_{(16)}, & 13_{(16)}, & FE_{(16)}, \\ 44_{(16)}, & AF_{(16)}, & B8_{(16)}, & 39_{(16)}, \\ 09_{(16)}, & F6_{(16)}, & 59_{(16)}, & AE_{(16)} \end{array} \right\rangle$
$\mathcal{T}_1 \rightarrow \mathcal{U}$	`10:EE73A87302F613FE44AFB83909F659AE`	

Figure 6: *An example transcript of communication between some $\mathcal{U}$ and two* different *devices $\mathcal{T}_0$ and $\mathcal{T}_1$ (which, for convenience of explanation only, use the* same *cipher key $\hat{k} = \langle D3_{(16)}, 85_{(16)}, 33_{(16)}, 46_{(16)}, 02_{(16)}, 8B_{(16)}, 6E_{(16)}, 24_{(16)}, 86_{(16)}, 62_{(16)}, E9_{(16)}, 95_{(16)}, AB_{(16)}, 68_{(16)}, 7E_{(16)}, 25_{(16)}\rangle$): note that $\mathcal{T}_0$ has SIZEOF_RND = 0 whereas $\mathcal{T}_1$ has SIZEOF_RND = 16, which is reflected in the r subsequently communicated in each case.*