COMS30048 hand-out: exam-style revision questions

▷ **S1.** a i A good example of a symmetric block cipher is the Advanced Encryption Standard (AES), which implies $n_b = 128$ and a choice of $n_k = 128$, $n_k = 192$ or $n_k = 256$ depending on the required security level; clearly other examples exist (e.g., DES, PRESENT or IDEA) and are valid.

ii In this setting, the term symmetric relates to the fact that both parties use the same key k for encryption and decryption. In contrast, the use of asymmetric encryption would mean each of the parties has a public- *and* private key: the former is published openly and is used for encryption, the latter is kept secret by the owner and used for decryption.

Abstractly, imagine $\mathcal{B}$ has a public key k_{PUB} and a private key k_{PRI}. To send m, $\mathcal{A}$ first computes

$$c = \textsc{Enc}(k_{PUB}, m)$$

and communicates C. Then, $\mathcal{B}$ can compute

$$m = \textsc{Enc}^{-1}(k_{PRI}, c)$$

to recover m. A concrete example of this type of encryption scheme would be RSA, that utilises a trapdoor one-way function, i.e., one that is easy to compute in one direction (resp. perform encryption using the public key) but hard to invert (resp. perform decryption) without knowing the trapdoor (resp. the private key).

b Using a (very) simple diagram, use of the OSI model as suggested could be roughly described as:

```
         m                         m
         |                         ^
         v                         |
+-----------------+        +-----------------+
| 7. Application  |        | 7. Application  |
| 6. Presentation |        | 6. Presentation |
| 5. Session      |        | 5. Session      |
| 4. Transport    |        | 4. Transport    |
| 3. Network      |        | 3. Network      |
| 2. Link         |        | 2. Link         |
| 1. Physical     |        | 1. Physical     |
+-----------------+        +-----------------+
         |                         ^
         |                         |
         +----------- c -----------+
```

with the question being at what layer m is translated to and from c.

In a sense, encryption and decryption could be performed at a number of (and even multiple) layers. For example it is not uncommon for an encrypted SSL connection (managed at the application-layer) to run over an encrypted 802.11 interface (managed at a lower-layer). However, a reasonable solution would be to perform it as part of the presentation layer. This means that the network stack supports all applications transparently (i.e., it is not true that each application needs to deal with the task independently), and end-to-end security is preserved as far as possible (i.e., as far as possible, only the applications ever "see" m).

c In short, how does $\mathcal{A}$ know it communicates with $\mathcal{B}$ and not $\mathcal{E}$? Without authentication of the parties to give this confidence, $\mathcal{E}$ could masquerade as $\mathcal{B}$ or even act as a man-in-the-middle by relaying messages between $\mathcal{A}$ and $\mathcal{B}$ (while capturing and/or manipulating them). On a wired network, this of course needs some physical intervention, but is arguable more problematic over a wireless network where there is less control over physical connections.

d The confidentiality of m relates to the fact that no party other than $\mathcal{A}$ or $\mathcal{B}$ can recover it from C. That is, as a result of encrypting m using the block cipher, it remains secret: it can only be recovered from c by those who also know k (i.e., not a passive eavesdropper).

The integrity of m relates to the the idea that it might be manipulated or altered somehow. Ignoring any error correction used by the network stack, the idea is that if $\mathcal{B}$ receives some $c' \neq c$ (e.g., as the result of manipulation by an active man-in-the-middle for example) then $m' \neq m$ will be recovered. However, currently, $\mathcal{B}$ cannot determine if or when such an event occurs.

e There are various advantages and disadvantages (both theoretically and practically motivated), and hence numerous valid answers. Two examples are:

i When using MAC-then-encrypt, the receiver has to decrypt the ciphertext in order to recover the tag and then validate the plaintext; if this validation fails, i.e.,

$$\textsc{Ver}(k', m, \tau) = \textbf{false},$$

the effort of decryption is wasted in some sense. The same is not true of encrypt-then-MAC, where the validity of the ciphertext can be checked before the need for decryption.

ii Intuitively, encrypt-then-MAC allows an attacker to manipulate the tag and encrypted plaintext independently. Whether this is a problem or not depends on the context, but either way the same feature is not shared by MAC-then-encrypt: to manipulate the tag, an attacker must have first decrypted c' which suggests they broke the block cipher.

f i This mode of operation is called Electronic Code Book (ECB), the idea being that translation of a plaintext block into a ciphertext block amounts to look-up in some fixed code book (implied by the key). This description suggests the problem: if we have two plaintext blocks where $m_i = m_j$, their encryption will yield identical ciphertext blocks. This might not be ideal, since it gives an attacker information about the structure of m (which alone might be enough for their purpose).

ii There are numerous alternatives: one example is the Cipher Block Chaining (CBC) mode, with encryption described as

$$c_i = \textsc{Enc}(k, m_i) \oplus c_{i-1}$$

and decryption as

$$m_i = \textsc{Enc}(k, c_i) \oplus c_{i-1}$$

in both cases given $c_0 = iv$ for some random Initialisation Vector (IV). Using CBC mode, even if we have some $m_i = m_j$ the encryption of these blocks will differ since c_i and c_j also depend on all preceding ciphertext blocks as well as m_i or m_j.

g i Notice that $\mathcal{B}$ will receive c_{l-1}, and then decrypt it to recover

$$m_{l-1} = \textsc{Enc}^{-1}(k, c_{l-1})$$

In essence, the problem is simply that given c must be a whole number of blocks (if $\mathcal{A}$ sends only part of c_{l-1} then clearly *none* of c_{l-1} can be correctly decrypted), how can $\mathcal{B}$ know how much of the decrypted m_{l-1} is valid? If $\mathcal{B}$ ignores the problem, then in some cases M is not reliably communicated between the parties.

ii There are a variety of approaches to solve this problem, but perhaps the simplest is to prepend the message length. That is, $\mathcal{A}$ forms a new message

$$m' = n \parallel m$$

before encrypting it to produce and communicate c'. When $\mathcal{B}$ decrypts c', it knows n and so knows the original size: this permits it to discard any invalid content in the final block (which, if one includes then, imply a message of length $n' = \lceil n/b \rceil \cdot b > n$).

▷ **S2.** i The standard way to do this is via an iterative construction; the idea is to break m up into l blocks, each n_k bits in size, where m_i is i-th block for $0 < i \leq l$. Then we “chain together” invocations of the block cipher using

$$\begin{array}{rcl} d_0 & = & iv \\ d_i & = & \textsc{Enc}(m_i, d_{i-1}) \end{array}$$

for some initialisation vector iv, and setting $d = d_l$, i.e., taking the last element in the chain as the hash function output. Roughly, this is the Merkle-Damgård construction: typically the description is extended to pad the message so it is of the right length, and to add an extra block including the message length to the end of the message (acting to prevent certain types of attack).

ii In short, the following describes the purpose of and difference between MDCs and MACs:

i. A Modification Detection Code (MDC) can be characterised as an unkeyed hash function. The purpose of this primitive is to detect when an input has been modified: the MDC output acts as a short fingerprint which if the input changes will also (change with high probability).

ii. A Message Authentication Code (MAC) can be characterised as a keyed hash function. The purpose of this primitive is to ensure that the input is authentic, i.e., was sent by someone how knows the shared key.

Since $\textsc{Hash}$ is unkeyed, it corresponds to a MDC; for a MAC, one might use $\textsc{Enc}$ within a construction such as CBC-MAC, or $\textsc{Hash}$ within a construction such as HMAC.

a i A typical PRNG is a function

$$\text{PRNG} : \{0,1\}^{n_s} \rightarrow \{0,1\}^{n_s} \times \{0,1\}^{n_r}.$$

It takes an n_s-bit state as input, and produces a n_s-bit state and an n_r-bit pseudo-random number as output. The idea is that using some initial seed state, the function is applied iteratively to produce successive pseudo-random numbers. Each number is generated deterministically, and hence are not really random, but the sequence will have the similar statistical properties as (ideally will be indistinguishable from) real randomness.

One application for a PRNG is generation of key material, or ephemeral nonces within a scheme or protocol.

ii There are many possible approaches; one is to use Enc in CTR (or "counter") mode. In short, we take an n_b-bit counter c and initialise it to some constant, then select a random n_k-bit key k (or set it to a seed for the PRNG). To update the PRNG and produce some output, one performs something like

$$\begin{array}{rcl} r & \leftarrow & \textsc{Enc}(k, c) \\ c & \leftarrow & c + 1 \end{array}$$

which produces the output r and updates c. The n_b-bit value c represents the state (although one could argue that we also need k, so the total size is arguably $b + k$ bits), and the output is also n_b bits; this is attractive because traditional PRNGs often produce much less output per-update. There are only 2^{n_b} possible outputs from Enc, so the period is at most 2^{n_b}.

▷ **S3.** a Without further knowledge about Enc (i.e., the algorithm) we cannot apply any structural approaches to cryptanalysis. However, we can make an estimate about brute-force attacks: by measuring how long an invocation of Enc takes, we can estimate how long a brute-force attack would take to search the entire key space. Based on this estimate, we can assess whether n_k is too small for our needs:

i for government documents, we want long term security (assuming they are never declassified) so if the attack would take anything less than the lifetime of the universe, k is too small, but

ii an RFID tag attached to perishable goods has a relatively short life span: we might not care if the attack takes more than 10 years because the tag would be destroyed by that point anyway.

b i With $n_k = 16$, there are only $2^{16} = 65536$ possible keys; as such it is trivial to search the entire key space. To mount a chosen plaintext attack, for example, we get a ciphertext

$$c = \textsc{Enc}(k, m)$$

for our choice of m without knowing k. To recover k, we simply try every possible k' and compute

$$c' = \textsc{Enc}(k', m)$$

until we find a $c' = c$ and hence know $k' = k$.

ii With $n_b = 8$, there are only $2^8 = 256$ possible plaintext (and ciphertext) values. To mount a chosen plaintext attack, for example, we get a ciphertext

$$c_i = \textsc{Enc}(k, m_i)$$

for all possible plaintexts, i.e., all m_i. We do not need to recover k because now we have a dictionary which can map every plaintext to the corresponding ciphertext: if we want to decrypt some c', we can just look-up the corresponding m'.

▷ **S4.** a A known ciphertext attack is where the attacker has access to a set of ciphertexts and their corresponding plaintexts (under the key in question), but cannot select the content of that set.

b A ciphertext only attacks restricts the known ciphertext attack by removing knowledge of the corresponding plaintexts: the attacker is simply presented with a set of (valid) ciphertexts. Typically the attacker can only proceed by making statistical assumptions about the plaintexts, e.g., that although they are unknown, they are English text.

c A chosen ciphertext attack extends the attackers ability from the known ciphertext case: it allows the attacker to select the ciphertexts in the set he receives.

d Finally, an adaptive chosen ciphertext attack extends the chosen ciphertext case by allowing the attacker to build the set in an interactive manner. Instead of selecting the set as a "one off", the set of chosen ciphertexts is constructed step-by-step; at each step the attacker can base his decision on the previous steps.

▷ **S5.** a i Given $a = g^x$, compute x.

ii Given $a = g^x$ and $b = g^y$, compute $c = g^{x \cdot y \pmod q}$.

iii Given $a = g^x$, $b = g^y$ and $c = g^z$, decide whether or not $z = x \cdot y \pmod q$.

b Essentially this means there is a polynomial time algorithm which can reduce a problem instance of Y into an instance of X; effectively this allows us to "translate" between problems, or reason about the difficulty of one problem in relation to another. In this case for example, X is no harder than Y: if we can solve Y we can always solve X.

c Given an oracle to solve DLP, given $a = g^x$ we can recover x. We can use such an oracle to solve a DHP instance: simply feed the oracle a and b to recover x and y, then compute $g^{x \cdot y \pmod{q}}$. Given we can now solve DHP instances using the DLP oracle, we can say $DHP \leq_P DLP$. The next step is to solve a DDH instance: using the ability to solve DHP instances, we can compute $g^{x \cdot y}$ and simply compare this to g^z. If and only if the two are equal do we find that $z = x \cdot y \pmod{q}$. This means we can write $DDH \leq_P DHP$ and hence, by transitivity, $DDH \leq_P DLP$.

▷ **S6.** a i Since $x, y \in \mathbb{F}_p$ we know $0 \leq x, y < 7$. Therefore, probably the quickest way to list the rational points is to compute the LHS and RHS of the curve equation and see where the matches are. So given

x	y	$x^3 + x + 1 \pmod{7}$	$y^2 \pmod{7}$
0	0	1	0
1	1	3	1
2	2	4	4
3	3	3	2
4	4	6	2
5	5	5	4
6	6	6	1

we see that the rational points are those in the set

$$\{\mathcal{O}, (0,1), (0,6), (2,2), (2,5)\}$$

i.e., the pairs (x, y) where $y^2 \equiv x^3 + x + 1 \pmod{7}$.

ii To compress P, the sender computes the parity of P_y as

$$t = P_y \pmod{2}$$

and then communicates $P' = (P_x, t)$ which is $\lceil \log_2(p) \rceil + 1$ bits; this saves $\lceil \log_2(p) \rceil - 1$ bits since communicating P would have required $2 \cdot \log_2(p)$ bits. To decompress P', the receiver computes

$$P'_y = \sqrt{P_x^3 + P_x + 1} \pmod{p}.$$

If there is no solution, then the point is invalid; for valid points there are two possible solutions as a result of the symmetric nature of the curve. A choice between the solutions is made via

$$P_y = \begin{cases} P'_y & \text{if } P'_y \equiv t \pmod{2} \\ p - P'_y & \text{if } P'_y \not\equiv t \pmod{2} \end{cases}$$

to reconstruct $P = (P_x, P_y)$. As an example, note that for $P_x = 2$ there are two possible values of P_y, namely 2 and 5, that produce rational points:

i. Imagine we select $P = (P_x, P_y) = (2, 2) \in E(\mathbb{F}_p)$. The sender computes

$$t = P_y \pmod{2} = 0$$

and communicates $P' = (P_x, t) = (2, 0)$. The receiver computes

$$P'_y = \sqrt{P_x^3 + P_x + 1} \pmod{7} = 2$$

and since $P'_y \equiv t \pmod{2}$ sets $P_y = P'_y = 2$, reconstructing the point as $(P_x, P_y) = (2, 2)$.

ii. Imagine we select $P = (P_x, P_y) = (2, 5) \in E(\mathbb{F}_p)$. The sender computes

$$t = P_y \pmod{2} = 1$$

and communicates $P' = (P_x, t) = (2, 1)$. The receiver computes

$$P'_y = \sqrt{P_x^3 + P_x + 1} \pmod{7} = 2$$

and since $P'_y \not\equiv t \pmod{2}$ sets $P_y = p - P'_y = 5$, reconstructing the point as $(P_x, P_y) = (2, 5)$.

b i To generate the public key for RSA, we need to first generate $N = p \cdot q$ which demands generation of two 512-bit primes p and q; this phase is computationally expensive due to the requirement for primality testing. Next we generate a random $1 < e < \Phi(N)$ which is coprime to $\Phi(N)$; the random generation is inexpensive, but will potentially need to be repeated until $\gcd(e, \Phi(N) = 1)$. Of course, this step can be skipped if one uses a fixed e. The public key length is thus roughly $2 \cdot \log_2(N)$, or 2048 bits, for a random choice of e.

To generate the public key for EC-IES, we first generate the private key as a random integer $1 \leq d < n$; we then generate the public key by computing $P = [d]\,G$ for the public generator $\langle G \rangle = E(\mathbb{F}_p)$. So the public key length is roughly $2 \cdot \log_2(p)$, or 512 bits, if no point compression is used.

As such, EC-IES can be said to have both a smaller public key and more efficient key generation. Further, arguable, advantages include the fact that fewer operations are required; this may lead to a smaller footprint, greater maintainability etc. For example, EC-IES requires the same scalar multiplication (required to compute P) as within an associated key generation. In contrast, RSA uses primality testing in key generation alone.

ii The public key in EC-IES is a point $P \in E(\mathbb{F}_p)$. There is no notion of "small" points, which would mean points that afford more efficient scalar multiplication. Further, if one fixed P as one would fix e in RSA then this would imply a fixed private key d; RSA avoids this problem because N, the modulus, can change. As a result, in EC-IES there is no analogous concept to the "small" encryption exponents used in RSA.

iii This is quite an open ended question; there are numerous answers, including windowing techniques for example. However, the simplest approach might be to utilise the fact that one can compute $P - Q$ more or less as efficiently as $P + Q$ for some $P, Q \in E(\mathbb{F}_p)$: point negation is inexpensive.

Thus, we can recode k into a signed binary representation k' where each digit $k'_i \in \{-1, 0, +1\}$. An example of such a recoding technique is the so-called Non-Adjacent Form (NAF). The advantage of this approach is that the larger digit set allows k' to be shorter than k, and also to potentially be of lower-weight (i.e., have less non-zero digits) because of the NAF recoding strategy. Once k' is constructed, one can compute the result as follows:

Input: A group element $P \in \mathbb{G}$ of order n, an integer $0 \leq k' < n$ in NAF representation
Output: The group element $Q = [k]\,P \in \mathbb{G}$

```
1  $t \leftarrow \mathcal{O}$
2  for $i = |k'| - 1$ downto $0$ step $-1$ do
3      $t \leftarrow [2]\,t$
4      if $k'_i = +1$ then
5          $t \leftarrow t + P$
6      end
7      else if $k'_i = -1$ then
8          $t \leftarrow t - P$
9      end
10 end
11 return $t$
```

The shorter and lower-weight k' means less loop iterations than would be required for binary exponentiation with k (meaning less point doublings), and less point addition (or subtraction) operations. For top marks, note should be made that use of projective coordinates and a "mixed" addition operation may provide an additional efficiency improvement.

▷ **S7.** a Given an elliptic curve $E(\mathbb{F}_q)$ of order n and some generator $G \in E(\mathbb{F}_p)$, the EC-DLP is: given $Q = [k]\,G$, find k. The best known algorithm to solve the EC-DLP requires $O(\sqrt{n})$ steps; if we want to have an m-bit security level then $2^m \simeq \sqrt{n}$. For example, for 128-bit security (e.g., to match AES) we need a group with a 256-bit order.

b • One can go into quite some detail about the domain parameters, but essentially we just need to agree on a curve to use: assume there is a curve $E(\mathbb{F}_q)$ of order n and some generator $G \in E(\mathbb{F}_p)$ which are known to all parties.

• First select d, the private key, as a random integer in the range $1 \leq d < n$. Then compute the public key as

$$P = [d]\,G$$

and make it known to all parties.

• To sign a message m using the private key d:

i Select $k \xleftarrow{\$} \{1, 2, \ldots, n-1\}$, and compute $Q = (Q_x, Q_y) = [k]\,G$.
ii Convert $Q_x \in \mathbb{F}_q$ into the integer $x \in \mathbb{Z}$.
iii Compute $r = x \pmod{n}$; if $r = 0$ then goto step 1.

iv Given an appropriate hash function μ, compute $m' = \mu(m) \pmod{n}$.
v Compute $s = (m' + d \cdot r)/k \pmod{n}$; if $s = 0$ then goto step 1.
vi Return $\sigma = (r, s)$.

- To verify a signature $\sigma = (r, s)$ on a message m using the public key P:

i If $r, s \notin \{1, 2, \ldots, n-1\}$ then return **false**.
ii Given an appropriate hash function μ, compute $m' = \mu(m) \pmod{n}$.
iii Compute $t = s^{-1} \pmod{n}$, $u_1 = m' \cdot t \pmod{n}$ and $u_2 = r \cdot t \pmod{n}$.
iv Compute $Q = (Q_x, Q_y) = [u_1]G + [u_2]P$.
v Convert $Q_x \in \mathbb{F}_q$ into the integer $x \in \mathbb{Z}$.
vi If $x \equiv r \pmod{n}$ then return **true**, otherwise return **false**.

c During signing we have $s = (m' + d \cdot r)/k \pmod{n}$ so that means

$$k = (m' + d \cdot r)/s \pmod{n}$$

and thus

$$Q = [k]G = \frac{m'}{s} \cdot G + \frac{d \cdot r}{s} \cdot G.$$

We know $P = [d]G$ so during verification

$$\begin{array}{ccccc} Q & = & [u_1]G & + & [u_2]P \\ & = & \left[\frac{m'}{s}\right]G & + & \left[\frac{r}{s}\right]P \\ & = & \left[\frac{m'}{s}\right]G & + & \left[\frac{d \cdot r}{s}\right]G \end{array}$$

i.e., the Q computed during verification must be the same as the Q computed during signing (if the signature is valid).

d i ECC in general has the advantage of using small operand sizes to achieve the same level of security; the modular arithmetic in comparison to factoring based schemes for example, uses a smaller modulus. This is due to the resulting EC-DLP being hard(er).

ii Unlike factoring based schemes for example, ECC allows the use of special-form moduli: this allows very efficient methods for modular reduction.

iii A point subtraction is very inexpensive; this allows for signed expansions of the scalar used in a scalar multiplication. For example, the NAF method for scalar multiplication is more efficient than a binary method due to this fact.

e It is helpful to assume a specific curve; for the sake of argument, assume we are working with a prime field $\mathbb{F}_p$ and the curve

$$E : y^2 = x^3 + a_4 x + a_6$$

where $a_4, a_6 \in \mathbb{F}_p$, and the prime order (sub-)group of order n is generated by some point $G \in E(\mathbb{F}_p)$. There are (at least) two options:

i Compute $m' = m \pmod{n}$, and then $Q = [m']G$; this produces Q from m, but is rather inefficient since it requires a scalar multiplication.

ii First use a normal hash function μ to turn the message m into an digest m' and reduce it modulo p, i.e.,

$$Q_x = m' = \mu(m) \pmod{p}.$$

Then, if there is a solution

$$Q_y = \sqrt{Q_x^3 + a_4 Q_x + a_6}$$

the point $Q = (Q_x, Q_y)$ is our result produced from m. If not, then we need to update Q_x deterministically, e.g.,

$$Q'_x = f(Q_x) = Q_x + 1,$$

and try again until there is a valid solution Q'_y to the curve equation. Presuming there are enough points that we eventually find a suitable Q_x, this is typically more efficient than the first option (provided we use an efficient algorithm for computing square roots in $\mathbb{F}_p$).

▷ **S8.** There are a wide variety of answers to each part of the question; the goal is simply to cite some sensible cases, such as the following:

a p and q must clearly be prime, and large enough that any prescribed security level is satisfied; they should ideally have balanced sizes, and certainly need to be selected at random.

The latter properties are more subtle than primality, but vitally important; for example, if there is a known non-random relationship between them (e.g., one is the next prime after the other) N will be easier to factor.

b Clearly selection of an appropriate security level will ensure the keys are secure and hence usable for the longest time; however, the question specifically asks for post-generation guidelines. Since the context relates to long-lived key material, this also means techniques such as key refresh are not appropriate.

One example is careful use of secure programming techniques, such as zeroisation, to minimise the issue of data remanence; one might aim to ensure the key material cannot be recovered by inspecting memory using a cold boot attack, for example. The other obvious example is to minimise the impact of leakage by ensuring any use of the key material (e.g., decryption using the private exponent) employs appropriate countermeasures (e.g., exponent blinding).

▷ **S9.** First, note that:

- The `Issuer` field identifies the party who issued the certificate; in essence, the issuer "creates" the certificate upon request by the subject. In this case a series of X.500 fields, such as Common Name (CN) and Organization (O), identify Thawte as the issuer.
- The `Subject` field identifies the party to whom the certificate has issued by the issuer; in essence, the subject is the "owner" of the certificate. In this case, a series of X.500 fields, such as Common Name (CN) and Organization (O), identify Google (specifically `www.google.com`) as the subject.

Upon downloading the certificate, the goal of the client is to establish whether it is valid or not; this acts as a means of establishing whether the public key presented by the server is valid or not.

- A chain exists from some root certificate C_0 to this one, say C_{n-1}, i.e., the chain is

$$\langle C_0, C_1, \ldots C_{n-1} \rangle.$$

- C_0 is implicitly trusted: it might be embedded in the web-browser for example, and probably self-signed. Given this fact, the signature on C_1 by the root CA can be verified, since the public key in C_0 is authentic. This process continues step-by-step for each C_i using the public key in C_{i-1} until C_{n-1} is validated.
- In this case, we definitely are not looking at a certificate from a root CA since, for example, the `CA` field (within the X.509 extensions section) says so. Assume the certificate chain in this case is

$$\langle C_0, C_1 \rangle$$

so the client is dealing with C_1 (of the subject): it first needs to obtain and validate C_0 (of the issuer). Assume C_1 is downloaded from the URL in the `Authority Information Access` field, and the client trusts it (maybe it is self-signed).
- C_1 includes time-stamps in the `Not Before` and `Not After` fields that specify the start and end of a validity period; the client would typically check these and reject the certificate outright if not within the specified period (i.e., irrespective of other checks).
- Otherwise, the client takes the certificate and verifies it using the public key from C_0 and the algorithm cited in the `Signature Algorithm` field.

▷ **S10.** a The obvious answer here is simply to list various symmetric and asymmetric primitives (e.g., AES, RSA, etc.) but the question asks for some reasoning as to why one might be selected over another.

There are at least two dimensions to consider:

- The symmetric operations are typically less expensive per-invocation, but there will more of them. That is, something like AES will be used to encrypt and decrypt a lot of data once the handshake is complete.
- The asymmetric operations are typically more expensive per-invocation, but there will less of them. That is, the handshake is a bottleneck: until it has been completed, the client and server cannot do anything else wrt. their session.

At face value the computational cost will dominate, but it is worth keeping in mind the (potential) need to "batch" effort: the overhead of transfer to and from the accelerator could be non-trivial, so amortising this via batched transfer could be important. An additional issue is that the accelerator should support a range of options since the cipher suite is agreed by negotiation; we do not want to reject clients who cannot support the same things as the accelerator for example.

As a result of these issues, a sane recommendation would be to invest in an accelerator (or accelerators) that can support AES, SHA-1 and modular exponentiation since given the latter deals with RSA and DH-based key exchange, these form a large set of common cipher suites; the other operations would then be supported in software.

b The server relies on the supply (or existence) of various resources. Two examples are memory (e.g., to store the state for each session) and randomness (e.g., in the initial steps of a handshake, or within a DH-based key exchange). If either of these resources is exhausted, then the server cannot operate: service might be denied to clients which try to communicate with it thereafter. For example, there is no memory for new sessions clearly no new sessions can be created; if the entropy pool is empty, the random number generator should block which means establishment of a session takes longer.

To use this fact in a DoS attack, and attacker needs to *force* the resource to be depleted and eventually exhausted. One way to do this would be to establish a huge number of "zombie" sessions that are kept open but idle; each such session consumes some memory and uses some randomness. In addition to the network and computational load, the net result could be that subsequent legitimate sessions cannot be established.

▷ **S11.** The reason for certification in the first place is to allow a client to check whether it is communicating with the server it expects, or in more detail that the public key presented by a server is for that server. A central reason for certification being required is prevention of Man-in-the-Middle (MitM) attacks where an attacker masquerades as a server.

Once in possession of the DigiNotar private key, the attackers would be could issue certificates as if they were the CA; the in fact did this, creating a fake certificate for Google. Among numerous implications, two examples are:

a Armed with such a certificate, and if they were also able to get "between" a client and the real server (in the sense that the client contacts the attacker rather than the server), the attacker can launch a man in the middle attack per the description above. Again, there is evidence to suggest this actually happened.

b As a countermeasure against this attack, various web-browsers revoked the DigiNotar certificate; subsequent validation of any certificate chain including the DigiNotar certificate would fail. As an aside, Google Chrome was able to detect the attack due to the certificate pinning feature included.

c A final implication is that trust in DigiNotar was lost to such an extent that the company was dissolved having been declared bankrupt. That is, beyond the technology-related impact above there is social impact as well; one could argue confidence in SSL impacts on confidence in e-commerce more generally for example.

▷ **S12.** a For key exchange, ephemeral Diffie-Hellman is used; note that there is a clear difference between static and ephemeral Diffie-Hellman.

Based on a public p and g (a generator of $\mathbb{Z}_p^*$) supplied in a message from the server, the idea is

i The client selects a random c_{PRI} and computes $c_{PUB} = g^{c_{PRI}}$.

ii The server selects a random s_{PRI} and computes $s_{PUB} = g^{s_{PRI}}$.

iii The client and server exchange c_{PUB} and s_{PUB}.

iv The client computes $k = s_{PUB}^{c_{PRI}} = g^{s_{PRI} \cdot c_{PRI}}$.

v The sever computes $k = c_{PUB}^{s_{PRI}} = g^{c_{PRI} \cdot s_{PRI}}$.

b To verify that the parties involved are authentic (i.e., that the client/server are who the server/client expects), RSA-based certificates are used.

The client authentication is optional, and only performed at the request of the server. So, focusing on server authentication, the idea is that the server sends a certificate to the client. An X.509 certificate, created by a trusted (and globally known) Certificate Authority (CA), for example, contains:

i some meta-data, or credentials C including

- a certificate serial number and validity period,
- the server public key,
- the server and CA domain names,

and

ii a signature σ by the CA on the meta-data C.

The client holds a list of CAs, and can therefore check that the signature σ is valid *and* that the meta-data matches the communication session it is engaged in (e.g., whether domain names match). When RSA signatures are used per above, the CA computes

$$\sigma = \mu(C)^{d} \pmod{N}$$

i.e., the hash of C encrypted with the private exponent d. Then, the client can verify this by testing whether

$$\mu(C) \overset{?}{\equiv} \sigma^{e} \pmod{N}$$

given the CA public exponent e held in the entry for that CA it holds.

c For application data encryption, AES-128 (i.e., AES with a 128-bit key) is used in CBC mode.
Once the master key has been established and then an application key k is subsequently generated, this is used to encrypt each plaintext block m_i as follows

$$\begin{array}{rcl} c_0 & = & iv \\ c_i & = & \text{AES-128.}\textsc{Enc}(k, m_i \oplus c_{i-1}) \end{array}$$

where iv is an Initialisation Vector (IV).

d To ensure that the encrypted application data is received without error (or being tampered with), a MAC based on SHA-1 (probably HMAC) is computed over the ciphertext blocks.

▷ **S13.** a The encryption (resp. decryption) of streamed video is a high-throughput application: a large amount of video data will need to encrypted by the server under the key owned by a client, and then decrypted by the client itself. This demand on performance means encryption (resp. decryption) performance is important. Therefore, a symmetric scheme (i.e., a block cipher such as DES or AES) is a better choice than an asymmetric scheme (i.e., RSA) since the latter is more computationally expensive. Of DES and AES, DES is both less efficient and has a lower security margin; therefore AES is the better choice.

b Use of "textbook" DES, AES and RSA is unattractive for similar reasons, in all cases the problem is essentially determinism: under a given key, a given plaintext is always encrypted into the same ciphertext. Usually this implies an unattractive security property, e.g., a lack of semantic security; in the context of the application, repeated frames of encrypted video can be detected. The alternative is to wrap RSA in a padding scheme such as OAEP, and DES or AES in a mode of operation such as CBC mode.

c The compression of data is typically based on non-randomness, or rather repetition or redundancy. An effective encryption scheme will produce ciphertext that has neither of these properties; attempts to compress encrypted data will therefore be less effective, i.e., there will be less saving.

d If each user has the same key, the server need do less work: it just needs to encrypt each video once. On the other hand, the users can collude and reduce the profit gathered by the company; in short, only one user need pay for a key that he can then distribute to all other users. If each user has a separate key then this is less likely: if one of the users gives his key to another user, the company can detect the same video being downloaded by two computers. On the other hand, this choice means the server must now encrypt each download for that particular user.

e In each case, the idea is to pre-compute as much as possible; this reduces the amount of computation to a minimum and maximises performance:

- For DES this can be achieved by pre-computing (and reusing) the key schedule for multiple blocks, and by expanding the S-boxes into larger look-up tables by pushing the various permutations into them.
- For AES this can be achieved by pre-computing (and reusing) the key schedule for multiple blocks, and by expanding the S-boxes into so-called T-tables that pre-compute the `SubBytes` and `MixColumns` operations.
- For RSA the pre-computation is harder since most approaches rely on pre-computation wrt. the base (i.e., the x in $x^y \pmod{N}$) that represents the message, rather than exponent that represents the key. Given the task of encryption, it could be the public exponent e is already fixed (and small) in order to yield a short, efficient addition chain; the value of e might be common to all clients, even if they have different N. Even if e is *not* of this form, it could be worth pre-computing and then later using an efficient addition chain for each client. Even if this improves marginally on generic exponentiation, the pay-off could be worthwhile provided a large number of encryptions are (eventually) performed.

▷ **S14.** A summary of pertinent points is as follows:

a The underlying concept here is that of an Instruction Set Extension (ISE), namely an (ideally minimal) extension of some (typically general-purpose) base ISA, which aims to deliver (more) special-purpose or even domain-specific features; said features are typically efficiency- and/or security-enhancing in some way, and so span both functionality- and behaviour-oriented cases. AES-NI (or "AES New Instructions") is an ISE designed and deployed by Intel in modern generations of their x86-compatible micro-processor cores.

b The statement is not true, or at least it is lacks enough detail that various counter-arguments are clear.

c The central issue here is what "best" means, because use of an ISE clearly as positive *and* negative implications. On the positive side, use of AES-NI typically implies 1) more efficient software-facing metrics, such as execution latency ($\sim$ 20 instructions per block), memory footprint ($\sim$ 0), and so on, and 2) some inherent protection against certain classes of attack (e.g., based on data-dependent execution latency). On the negative side, however, use of AES-NI is not possible on *every* platform: for this general-purpose library, one would require a non-ISE fall-back.

▷ **S15.** a The period of the LCG is at most p: it cannot generate more than p distinct outputs since it operates modulo p. Given a clock frequency of 8kHz, we have 8000 clock cycles per-second; at this rate, the LCG will repeat roughly every

$$65535/8000\text{s} \simeq 8\text{s}.$$

Clearly this is not ideal: if an attacker engages in the protocol at eight second intervals, there is a high chance the same nonce will be produced by the smart-card each time: given this violates the requirement for the nonces to be random, the protocol might therefore be attacked.

b Typically, an LCG is viewed as more suited to implementations in software since a processor will usually have an ALU that supports integer addition and multiplication. In hardware, the need for such components is often avoided by opting for a Linear Feedback Shift Register (LFSR) instead: although such schemes usually only output 1 rather than $\log_2 p$ bits per step, they are often less expensive to realise since they do not require the same ALU-like support.

c Something like the ANSI X9.17/X9.31 algorithm would be more ideal. Imagine the block cipher encryption algorithm $\textsc{Enc}$ accepts an n_k-bit key and uses n_b-bit blocks; the smart-card has a cipher key k and seed s embedded in it, both of which are kept secret within non-volatile memory.

The following algorithm generates n random numbers as a sequence $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle$:

i Generate d, a 64-bit value based on a precise representation of the current time and/or date; compute $i = \textsc{Enc}(k, d)$.

ii For $j \in \{0, 1, \ldots, n-1\}$

- first compute $x_j = \textsc{Enc}(k, i \oplus s)$,
- then update $s = \textsc{Enc}(k, i \oplus x_j)$.

The only remaining issue is that each x_i is b bits; we want 16-bit values, so for example we need to truncate them.

▷ **S16.** a Two possible reasons for selecting one other the other are:

- The second shellcode program is longer. This could be deemed a disadvantage if it is too long to fit into the target buffer.
- The first shellcode program contains zero bytes (i.e., elements equal to `0x00`). This could be deemed a disadvantage in that if the target program uses `strcpy` to copy into the target buffer, said elements will terminate the copy early (resulting in the shellcode being incorrectly injected).

b There are a wide range of possibilities for countermeasures in this context. Some examples are:

- Use of a "canary" or checksum value (cf. StackGuard) which can be used to detect overwriting of stack content (and hence abort before control-flow is redirected).
- A network-based approach whereby packets sent from $\mathcal{E}$ are scanned by $\mathcal{D}$ to ensure they do not include any shellcode-like content.
- Use of a length-aware version of `strcpy`, e.g., one that terminates before writing outside the target buffer.
- Use of bounds checking to prevent overflow: this means altering each access to the buffer so the index is first check to ensure it is within a valid range.
- Randomisation-style approaches (cf. ASLR in Windows) whereby the address of various process components (e.g., the stack) changes in each invocation of the target.

▷ **S17.** a A signature $\sigma = (r, s)$ on a message m can be verified using the following algorithm:

```
Input: A public key $(p, g, y)$, a signature $\sigma = (r, s)$
       and a message $m$
Output: true if $(r, s)$ is a valid signature on $m$,
        otherwise false
i  Check whether $0 < r < p$ and $0 < s < p - 1$; if
   either check fails, return false.
ii If $g^{\mu(m)} \equiv y^r \cdot r^s \pmod{p}$ then return true, other-
   wise return false.
```

This works because

$$\begin{array}{lll} & y^r \cdot r^s & \pmod{p} \\ = & (g^x)^r \cdot (g^k)^{(\mu(m)-x\cdot r)/k} & \pmod{p} \\ = & g^{x\cdot r} \cdot g^{\mu(m)-x\cdot r} & \pmod{p} \\ = & g^{x\cdot r+\mu(m)-x\cdot r} & \pmod{p} \\ = & g^{\mu(m)} & \pmod{p} \end{array}$$

and hence if the signature is valid, the left-hand and right-hand sides of the test are equivalent modulo p.

b A basic smart-card often requires power and clock inputs to be provided by the terminal it is plugged into; in a sense, this means the environment and hence potentially an attacker can control the power and clock inputs. Since the inputs need to be accessible in normal use, any manipulation is non-invasive. For example, the attacker could "glitch" the clock (forcing it to have an irregular period) or under-supply power (causing malfunction of components such as transistors). Invasive methods are also possible. For example, after depackaging an attacker might attempt to use a laser to influence computation or corrupt memory or register content.

c One idea would be to influence the generation of randomness, i.e., k, somehow. Various scenarios border on realistic. For example, if a Pseudo-Random Number Generator (PRNG) draws from environmental sensors (e.g., heat) they can potentially be biased by the attacker (e.g., heating or cooling the device); where a PRNGs maintains an entropy pool it might output "bad" randomness when the entropy is depleted (e.g., through repeated use) rather than blocking.

In the worst case, this might mean r is constant (either random but fixed, or chosen). Imagine the attacker can do this and gets two signatures $\sigma_0 = (r_0, s_0)$ and $\sigma_1 = (r_1, s_1)$ on messages m_0 and m_1. Clearly

$$\begin{array}{lll} s_0 & = & (\mu(m_0) - x \cdot r_0)/k_0 \pmod{p-1} \\ s_1 & = & (\mu(m_1) - x \cdot r_1)/k_1 \pmod{p-1} \end{array}$$

but since $r_0 = r_1$ as a result of using the same randomness, we have

$$\begin{array}{lll} s_0 & = & (\mu(m_0) - x \cdot r)/k \pmod{p-1} \\ s_1 & = & (\mu(m_1) - x \cdot r)/k \pmod{p-1} \end{array}$$

meaning

$$\begin{array}{llll} s_0 - s_1 & = & (\mu(m_0) - x \cdot r)/k(\mu(m_1) - x \cdot r)/k & \pmod{p-1} \\ & = & (\mu(m_0) - \mu(m_1) - x \cdot r + x \cdot r)/k & \pmod{p-1} \\ & = & (\mu(m_0) - \mu(m_1))/k & \pmod{p-1} \end{array}$$

This means the attacker can compute

$$k = (\mu(m_0) - \mu(m_1))/(s_0 - s_1)$$

and then recover x. Alternatively, one could imagine that a clock glitch or similar could "skip" the PRNG call which would imply the previous k is reused; similar reasoning to the above allows one to recover x.

▷ **S18.** a Assuming the round keys are provided in the right order, an example implementation would be

```
uint8_t decrypt( uint8_t* k, uint8_t c ) {
  uint8_t t = c;

  for( int i = ( r - 1 ); i >= 0; i-- ) {
    if( t & 0x01 ) {
      t = 0x8D ^ ( t >> 1 );
    }
    else {
      t =        ( t >> 1 );
    }

    t = sbox_decrypt[ t ] ^ k[ i ];
  }

  return t;
}
```

otherwise, the access `k[ i ]` needs to be something like `k[ r - i - 1 ]`.

b There are (at least) three sources of potential information leakage:

i The XOR of `c` and `k` will consume power that is related to the Hamming weight of the operands (keeping in mind that `k` is secret). As such, measuring the power consumption one can potentially recover information about `k` (which should not be leaked). A solution to this problem might be some form of masking scheme.

ii The table look-up into `sbox_decrypt` may, depending on the underlying platform, be satisfied through a cache memory; the fact that the behaviour of such structures is data-dependent means that said behaviour depends on, and may therefore leak information about, `k`.

There are several forms of attack model in this context, for example trace-based, time-based and access-based attacks; countermeasures depend on which we are considering. Examples include constant-time implementation techniques (e.g., bit-slicing), hardware-oriented approaches (e.g., disabling the cache) and so on.

iii The conditional statement uses `t` to decide which operation to perform. Notice that the two branches do different things, and will hence take a different amount of time to execute. So given `t` is derived from `k` (which is secret), by measuring the execution time one can recover information about `k` (which should not be leaked).

A solution to this problem would be to rewrite the conditional so that it executes in the same time no mater what the input is. A simple approach would be to use a small table, e.g.,

```
uint8_t decrypt( uint8_t* k, uint8_t c ) {
  uint8_t T[ 2 ], t = c;

  for( int i = 0; i < r; i++ ) {
    T[ 0 ] =        ( t >> 1 );
    T[ 1 ] = 0x8D ^ ( t >> 1 );

    t = sbox_decrypt[ T[ t & 0x01 ] ] ^ k[ i ];
  }

  return t;
}
```

although other formulations are possible (and potentially more efficient).

c Consider the final round of decryption: the last steps performed (before the plaintext is returned) substitute the current state with an element of the S-box (i.e., the table `sbox_decrypt`) then XOR it with the key. Based on this fact, the attack is simple. If, before the last round, we corrupt every element of `sbox_decrypt` by rewriting them with zero, the result of said steps will yield the key.

That is, the expression `sbox_decrypt[ t ] ^ k[ i ]` will instead be `0 ^ k[ i ]` which of course equals `k[ i ]`. This is returned by the function, and hence to the attacker!

▷ **S19.**

a Assuming a little-endian bit order, we have

$$p = 6_{(10)} = 110_{(2)} \mapsto \langle 0, 1, 1 \rangle$$

and hence $n = 3$. As such,

$$k_{message} = f(f(f(k_{root}, 1), 1), 0) = H_0(H_1(H_1(k_{root}))).$$

From this, it should hopefully be clear that the message keys are generated by traversing a binary tree: p is the "path" from a root (represented by k_{root}) to a leaf (representing a choice of $k_{message}$) generated by applying H_0 and H_1 accordingly. This is demonstrated by Figure 1.

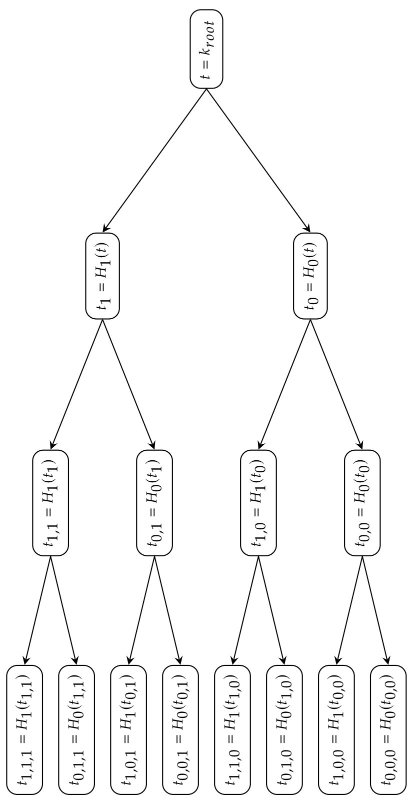
b Per above, KeyTree traverses a binary tree based on p: thus, since there are 2^n possible leaves, there are also 2^n possible message keys.

c i Simple Power Analysis (SPA) attacks, a class of side-channel attack, attempt to capitalise on the data-dependent power consumption of CMOS-based circuits. The idea is that an attacker can acquire a single trace of power consumption during execution of the target algorithm (e.g., Enc). At a given point in time, the power consumed will be influenced by both the operation being performed and the data it is being performed on. As such, the attacker analyses features in the trace and tries to infer any data which could have produced them during execution; when this data is security-critical (e.g., key material), recovering it through the side-channel clearly circumvents any normal security provided.

There are various assumptions that need to be made, but some depend on the context (e.g., the target algorithm and device). More general assumptions include the need for physical access to the device (to make acquisitions), and for the leakage to be strong enough to allow successful analysis using just one trace. Often this means the leakage must relate to gross differences in power consumption relating to data-dependent control-flow (and hence the operation performed at a given point in time).

ii The point here is that the client selects id at random, hence the server does not know it and therefore cannot compute the $k_{message}$ it needs to perform decryption. To resolve this problem, one idea would be to alter the message format so id is prepended. That is,

$$id \;\|\; \text{AES-128.Enc}(\text{KeyTree}(k_{root}, id), M)$$



$t = k_{root}$
$t_1 = H_1(t)$
$t_0 = H_0(t)$
$t_{1,1} = H_1(t_1)$
$t_{0,1} = H_0(t_1)$
$t_{1,0} = H_1(t_0)$
$t_{0,0} = H_0(t_0)$
$t_{1,1,1} = H_1(t_{1,1})$
$t_{0,1,1} = H_0(t_{1,1})$
$t_{1,0,1} = H_1(t_{0,1})$
$t_{0,0,1} = H_0(t_{0,1})$
$t_{1,1,0} = H_1(t_{1,0})$
$t_{0,1,0} = H_0(t_{1,0})$
$t_{1,0,0} = H_1(t_{0,0})$
$t_{0,0,0} = H_0(t_{0,0})$

Figure 1: *The space of keys generated by* KeyTree *for a given k_{root} and assuming $n = 3$ (i.e., a depth of 3).*

is communicated: the server can recover id and hence decrypt, but since an eavesdropper does not know k_{root} knowledge of id alone is useless.

iii The underlying idea is that the designers are trying to prevent the root key k_{root} being recovered by an SPA attack: if the attack recovers the key used by an invocation of Enc, then it recovers

$$k_{message} = \text{KeyTree}(k_{root}, id).$$

This allows decryption of the corresponding message, but *only* this message. Contrast this with the first version of the client, where the attacker recovers k_{root} and can decrypt *all* messages. As such, although using a fresh key for each message improves security in the sense one can argue there is less impact if said key is recovered, it does not protect the client in the way intended.

iv Given the SPA attack against Enc, alarm bells should sound wrt. this proposal! But this alone is not the answer: the question asks for some discussion of the impact on security, so "a bad implication" is not enough.

If we use AES-128 for H_0 and H_1 then the SPA attack implies that an attacker can recover the key used when KeyTree invokes them in the second client version. The first such invocation relates to the first iteration of the loop in KeyTree where $i = n - 1$ and we know

$$f(k_{message}, p_i) = f(k_{root}, p_i) = H_{p_i}(k_{root}, p_i) = \text{Enc}(k_{root}, p_i)$$

As such, the SPA attack can now recover k_{root} rather than simply $k_{message}$ as before. The attacker can therefore recover *all* message keys generated during the period of time k_{root} is fresh, and decrypt associated messages; before, they could only decrypt the one message relating to the $k_{message}$ recovered. Depending on the refresh period, n and so on this is a strong threat, and certainly more so than initially.

v The goal of a hiding countermeasure is to break the relationship between features in the trace acquired, and the data operated on. A simple way to do this is to use temporal randomisation: we try to randomise when an operation is performed in each execution, thereby preventing the attacker relating the trace at time t with a particular operation (and/or data).

Two options are particularly suitable for software: either

i. insert "dummy" operations (or delays) so that the real operations occur in the same order as normal but are shifted in time, or
ii. shuffle the real operations so they occur in a different order to normal (while maintaining any inter-dependencies).

d i A DPA (in contrast to SPA) requires the attacker to acquire many traces of power consumption during execution of the target algorithm; the idea is to recover security-critical data via application of statistical analysis, rather than direct inspection, of the traces. As such, and in common with most statistical approaches, the data set must be large enough to draw a meaningful conclusion: too small a data set (i.e., set of traces) means that noise will dominate any underlying features (i.e., the attack will fail).

The leakage from a target device would have to be unusually strong (i.e., the level of noise would have to be very low) for a DPA to succeed with only 10 traces, although 1000 is feasible. So if the attacker were able to collect power consumption traces while the client encrypts 1000 blocks under the same key, one might expect DPA to recover that key; with only 10 traces, this is unlikely.

ii This is clearly an open-ended question with many possible approaches; any sane idea is reasonable, as long as it is formalised and argued well.

One simple underlying idea would be to refresh the key used by AES-128, either per-block or less frequently if this is prohibitive. By doing so, the number of acquisitions an attacker can collect that relate to a given target key is limited: as above, this limits the feasibility of the attack.

Let $M[i]$ be the i-th of say m message blocks. There are various ways to achieve the refreshing; one basic idea is to now communicate

$$id[i] \parallel \text{AES-128.}\textsc{Enc}(\textsc{KeyTree}(k_{root}, id[i]), M[i])$$

using $id[i]$, the i-th of m random integers. Of course this now increases the total communicated data: using the original approach we communicated $128 \cdot m + n$ bits (i.e., the m blocks plus one m-bit id), but now this is $128 \cdot (n + m)$ bits.

Another approach would be select the parameters so the client can use a sequential "chunk" of message keys, one for each block: this means given one m-bit id, the client generates a message key

$$k_{message} = \textsc{KeyTree}(k_{root}, id \parallel i)$$

for the i-th block, and uses it to encrypt (in a sense, this is a little similar to the CTR mode of operation). Since only one id is now included in the message format, the overhead is reduced.

▷ **S20.** As a starting point, a section of principles cited by NIST IR-7977 include:

a Transparency, i.e., the idea that "*[a]ll interested and affected parties have access to essential information regarding standards and guidelines-related activities throughout the development process*". An example would be the process that produced FIPS-197 (or AES): the call for submissions issued in 1997 included a clear set of evaluation criteria, and stated guarantee that "*[a]ll of NIST's analysis results will be made publicly available*".

b Continuous improvement, i.e., the idea that NIST "*conducts research in order to stay current, to enable new cryptographic advances that may affect the suitability of standards and guidelines*". An example would be the process that produced FIPS-197 (or AES): in 2018 NIST published an report surveying the impact of AES, then in 2023 issued an update to FIPS-197.

c Innovation and Intellectual Property (IP), i.e., the idea that NIST "*has noted a strong preference among its users for solutions that are unencumbered by royalty-bearing patented technologies*". An example would be the process that produced FIPS-197 (or AES): the call for submissions issued in 1997 required completion of an intellectual property statement,