CSE 565 Solution to Homework 3

1. (135 points total) Implementations will vary depending on your choice of programming language and the cryptographic library. In general, it is expected that your implementation will properly setup a pseudorandom generator (using a strong random seed) for various uses in different algorithms (such as key generation, IV selection for encryption modes, etc.). The implementations themselves will be guided by the libraries' APIs.

   The expected performance is based on the following observations:

   (a) Creating an AES key amounts to retrieving the next portion of pseudorandom bits from the pseudorandom generator. In general, the time for generating a number of pseudorandom bytes is linear in the length of the desired number of bytes, but the time difference between obtaining 128 bits and 256 bits for an AES key may not be pronounced.

   (b) Creating RSA and DSA keys is expected to take longer with the increase in the key length. Generating RSA keys involves choosing two private large primes, and large primes become sparser with the increase in the size of the numbers. This means that in addition to having generate longer pseudorandom numbers, more trials will be needed to find primes and generate a longer key.

   DSA key setup also relies on prime numbers, but unlike RSA, the same setup can be reused across multiple keys. This means that a cryptographic library may choose a pre-determined setup for the desired key length without generating it anew, which will significantly reduce the key generation time. While the overall key generation time in that case will be significantly less than, e.g., RSA key generation time, the time is still expected to increase (proportional to the key size) with the increase in the key length.

   (c) The encryption and decryption times are expected to be linear in the size of the message/file. In the case of AES encryption modes, one extra block encryption is needed per message, which means that that cost is amortized among all blocks and technically the time per byte can be different for messages of different sizes because of this. In practice, for messages comprised of many blocks, the difference is unlikely to be noticeable.

   For RSA encryption, there is no extra block encryption, but rather each message block becomes shorter than the modulus size. For any message, the size of which is a multiple of the supported message block size, the time per byte should be the same.

   (d) AES CBC decryption is a couple of times slower than encryption because of more involved operations used in AES decryption algorithm. AES CTR mode, however, does not call AES decryption and thus the time of decryption shouldn't differ from the time of encryption because of cryptographic operations.

   RSA decryption is significantly slower than encryption because of the need to perform modulo exponentiation with a much larger exponent. The cost of modular exponentiation is linear in the size of the exponent and for that reason the expected slowdown can be approximated by dividing the modulus size by the size of the public exponent e in bits. RSA OAEP uses operations other than modulo exponentiation, but their speed (e.g., the speed of hashing a short message) is expected to be fast compared to public-key operations.

   (e) The time of AES encryption and decryption is expected to somewhat go up with the increase in the key size. For one, the algorithm needs to perform 14 rounds (for a 256-bit key) instead of 10 rounds (with a 128-bit key). There is a possibility of other sources of slowdown (e.g., during key expansion).

RSA encryption and decryption will also be slower for a larger modulus. The most straightforward implementation of modulo multiplication is quadratic in the modulus size and that of full-size modulo exponentiation (with the exponent of the same size as the modulus) is cubic in the modulus size. This means that the cost of encryption and decryption is expected to slow down by up to a couple of times (with greater slowdown for decryption), but the exact difference depends on the optimizations performed by the library.

(f) The cost of hashing is generally linear in the size of the message/file for sufficiently large sizes (a single block used internally by a hash algorithm is normally larger than that for symmetric encryption, but smaller than for public-key encryption). The costs of SHA-2 and SHA-3 with the same hash size are on the same order of magnitude, but the exact performance is somewhat different. SHA-3 is a couple of times faster than SHA-2 in hardware implementation, while it is a couple of times slower in software implementation. We typically use software implementation. You should also expect a slower performance for larger hash sizes, but not by a significant amount (the nature of the elementary operations used in hash algorithms doesn't change).

(g) Performance of DSA is expected to have somewhat different characteristics from other cryptographic tools. The message is hashed using a hash function, the cost of which is linear in the message size, while the signature itself uses a constant number of public-key operations. While the latter cost is one-time, it can dominate the overall performance. For that reason, per-byte times are expected to be different for small and large files (the per-byte cost is expected to be smaller for the larger file because the one-time cost gets amortized across a larger number of bytes).

The times of DSA signing and verification are comparable, with verification being slightly slower.

When we increase the key size, signing and verification times become slower. This is primarily due to the increase in the modulus size because the exponent size grows slower.

Unlike RSA, DSA can be implemented using Elliptic Curve cryptography, in which case it becomes significantly faster than its conventional variant (and RSA).

(h) You should expect that symmetric key operations (and other cryptographic operations such as hashing that rely on equivalent low-level operations) are much faster than public-key operations. The cost of software implementation of symmetric key encryption is comparable to that of hash functions, but the performance of the former got boosted with widely available hardware implementations.

The times you report may be additionally affected by one-time setup costs (e.g., object allocation and initialization) and noise (especially for very fast tasks).

The point distribution for this question was as follows: 10 points for each major functionality including proper setup and use of a PRG, AES key generation and encryption mode, hashing, RSA key generation and encryption/decryption, DSA key generation and signing/verifying (50 points total). Additional 2.5 points are given for each variation including: extra AES encryption mode, extra AES key size, extra hash functions (2), extra RSA key size, and extra DSA key size (plus 15 points).

1 point was given for running and measuring time of each functionality (key generation, encryption, decryption, hashing, signing, signature verification per message size). This consists of 14 points for AES, 6 points for hash functions, 10 points for RSA, and 10 points for DSA (40 points total).

There are also 30 points for justifying the performance according to the 5 different categories specified in the assignment. These are divided as 10 points for the third category and 5 points for each other category.

Points are deducted if the implementation is not reproducible (i.e., does not run on a CSE student machine or on the CSE virtual machine image).

2. (10 points each; 20 points total)

   (a) $\mathit{delete\_all\_rights}(s_1, s_2, o)$

   if ($\mathit{modify} \in A_i[s_1, o]$ and $\mathit{own} \notin A_i[s_2, o]$), then
   $A_{i+1}[s_2, o] = \emptyset$

   (b) $\mathit{copy\_all\_rights}(s_1, s_2, o)$

   $A_{i+1}[s_2, o] = A_i[s_2, o]$
   if ($\mathit{read}^* \in A_i[s_1, o]$), then $A_{i+1}[s_2, o] = A_{i+1}[s_2, o] \cup \{\mathit{read}\}$
   if ($\mathit{write}^* \in A_i[s_1, o]$), then $A_{i+1}[s_2, o] = A_{i+1}[s_2, o] \cup \{\mathit{write}\}$
   if ($\mathit{execute}^* \in A_i[s_1, o]$), then $A_{i+1}[s_2, o] = A_{i+1}[s_2, o] \cup \{\mathit{execute}\}$
   if ($\mathit{append}^* \in A_i[s_1, o]$), then $A_{i+1}[s_2, o] = A_{i+1}[s_2, o] \cup \{\mathit{append}\}$
   if ($\mathit{list}^* \in A_i[s_1, o]$), then $A_{i+1}[s_2, o] = A_{i+1}[s_2, o] \cup \{\mathit{list}\}$
   if ($\mathit{modify}^* \in A_i[s_1, o]$), then $A_{i+1}[s_2, o] = A_{i+1}[s_2, o] \cup \{\mathit{modify}\}$