CSci 5271: Introduction to Computer Security

Exercise Set 3 **due: November 25th, 2024**

Ground Rules. You may complete these exercises in a group of up to three students. Each group should turn in **one** set of answers, designating all group members using the group submission feature of Gradescope. You may use any written source you can find to help with this assignment but you **must** explicitly reference any source you use besides the lecture notes,assigned readings, or textbook. Unlike exercise set 1, AI tools like ChatGPT may **not** be used on this assignment. Submit an electronic copy of your solution online by 11:59pm on Monday, November 25th.

For this assignment, we will use Gradescope's template-based submission style. We will supply a template document with spaces to insert your answers in, in your choice of LaTeX or Google Docs formats. Prepare your answers by filling in this document and converting it to a PDF in a way that preserves the template formatting, and submit this to Gradescope. Note that we still ask that you type your written answers, rather than hand-writing and scanning them, since it is best for both you and us if your answers are easy to read.

1. Caesar's block cipher. (20 pts) The Caesar cipher is a historical encryption method based on advancing letters circularly through the alphabet. To discuss it in a modern context on ASCII, we can consider it to be a block cipher with an 8-bit block and a 5-bit key k. The encryption function E_k is defined as:

$$E_k(b) = \begin{cases} \texttt{0x41} + ((b - \texttt{0x41} + k) \bmod 26) & \text{if } \texttt{0x41} \leq b \leq \texttt{0x5A} \\ \texttt{0x61} + ((b - \texttt{0x61} + k) \bmod 26) & \text{if } \texttt{0x61} \leq b \leq \texttt{0x7A} \\ b, & \text{otherwise} \end{cases}$$

Recall that `0x41` through `0x5A` are the ASCII codes for `A` through `Z`, and similarly `0x61` through `0x7A` are `a` through `z`. The inverse operation is just shifting by the same amount in the opposite direction, so $D_k = E_{26-k}$ (we use the convention that the result of mod is always positive when the modulus is). ROT-13 corresponds to the special case $E_{13} = D_{13}$.

(a) CaesarCrypt S.p.A. is an Italian computer security company which got started building on their national heritage to market modern block ciphers that also have an 8-bit block size, but they have taken the lesson that the original Caesar cipher had too small a key size. Their first flagship product CCEA1 was a 8-bit block cipher with a 2048-bit key size. Their new successor cryptosystem, named CCEA2, increases CCEA1's key size to 4096 bits. CaesarCrypt's marketing materials suggest that this yields an astronomical increase in security by a factor of 2^{2048}. What do you think of this security claim: can CCEA2 really be more secure than CCEA1?

(b) In fact there are some general problems that affect any block cipher with a small block size. Describe a chosen-plaintext attack that would easily break any block cipher with an 8-bit block size.

2. Poor block ciphers. (30 pts)

Creating good block ciphers is difficult. In this question, your job is to point out why each of these proposals is not a good block cipher. Each cipher operates on 64-bit blocks and uses a 64-bit key. The cipher is shown in the form of C code for encryption and decryption functions operating on values of type `uint64_t`, which is equivalent to `unsigned long` on x86-64.

These flaws/possible attacks should be of one of four categories: a block cipher is not usable if it is *not invertible*, or it is insecure if it is vulnerable to a *key recovery*, *plaintext recovery*, or *distinguisher* attack under the chosen-plaintext assumption. For each part, state which kind of flaw you have identified (more than one may be possible), and then give the specifics of the flaw. Often the flaw/attack can be expressed in a short mathematical formula or example. You can also use words (and a brief explanation might help qualify for partial credit), but you don't need to write code. Variables P, C, and k (and with subscripts) always stand for plaintext blocks, ciphertext blocks, and keys respectively.

A cipher is *not invertible* if there is any key and plaintext message for which the decryption of the ciphertext is not the same as the original plaintext, i.e. $\text{Dec}(\text{Enc}(P, k), k) \neq P$. The setup for a chosen plaintext attack is that there is a key k_C and a plaintext P_C which are chosen randomly and initially unknown to the attacker (subscript C for "challenge"); let $C_C = \text{Enc}(P_C, K_C)$. The attacker receives C_C, and then chooses one or more plaintexts $P_1, P_2, \ldots, P_n$, and gets their encryptions under the challenge key, $C_i = \text{Enc}(P_i, k_C)$. Given this setup, the attacker is successful in *key recovery* if they can efficiently (much faster than brute force) compute k_C. Similarly, the attacker succeeds in *plaintext recovery* if they can efficiently compute P_C. The attacker succeeds as a *distinguisher* if they can compute a condition whose probability of being true with the randomly keyed block cipher is significantly different from the probability if the block cipher were replaced with a random permutation (one selected uniformly at random among all permutations on the block space). Some attacks may work for any values of the plaintext(s) without the attacker having to choose them (a *known plaintext* attack), in which case you don't need to describe choosing them.

(a) The identity function.

```
uint64_t enc(uint64_t p, uint64_t k) { return p; }
uint64_t dec(uint64_t c, uint64_t k) { return c; }
```

(b) Multiplication.

```
uint64_t enc(uint64_t p, uint64_t k) {
    if (k == 0) return p;
    return p * k; }
uint64_t dec(uint64_t c, uint64_t k) {
    if (k == 0) return c;
    return c / k; }
```

(c) XOR.

```
uint64_t enc(uint64_t p, uint64_t k) { return p ^ k; }
uint64_t dec(uint64_t c, uint64_t k) { return c ^ k; }
```

(d) A more complex use of XOR.

```
uint64_t enc(uint64_t p, uint64_t k) {
    return (p ^ (k >> 8)) ^ k; }
uint64_t dec(uint64_t c, uint64_t k) {
    return (c ^ k) ^ (k >> 8); }
```

(e) A Feistel cipher.

```
uint64_t enc(uint64_t p, uint64_t k) {
  /* Split into two 32-bit halves */
  uint32_t left = p >> 32;
  uint32_t right = p & 0xffffffff;
  for (int i = 0; i <= 55; i += 5) {
    uint32_t new_left = right;
    right = left ^
      (right << ((k >> i) & 31));
    left = new_left;
  }
  /* Reassemble the halves */
  return ((uint64_t)left << 32) | right;
}
```

```
uint64_t dec(uint64_t c, uint64_t k) {
  /* Split into two 32-bit halves */
  uint32_t left = c >> 32;
  uint32_t right = c & 0xffffffff;
  for (int i = 55; i >= 0; i -= 5) {
    uint32_t new_right = left;
    left = right ^
      (left << ((k >> i) & 31));
    right = new_right;
  }
  /* Reassemble the halves */
  return ((uint64_t)left << 32) | right;
}
```

3. (Mis-)using message authentication codes. (20 pts) Armed with a copy of Schneier's *Applied Cryptography* from a used bookstore, Sly can't wait to design his own encrypted thingamadoodad protocol. He starts off with a super-secure key exchange protocol that ends with Alice and Bob sharing secret keys for encryption (K_e) and authentication (K_a). Now he wants to design a secure symmetric channel using these keys.

(a) Sly decides at first that he wants to use a CBC-MAC based on AES with 128 bit blocks for integrity. He looks carefully at his key exchange protocol and realizes that an adversary can interfere to make Alice and Bob end up deciding on different keys. So the first message sent over by Alice will be $\tau_0 = \mathsf{cbcMAC}_{K_a}(0^{128}) = \mathsf{aesEncrypt}_{K_a}(0^{128})$. (The notation 0^n means n zero bits.) If Bob's local value doesn't check out, he aborts, otherwise the channel is usable. Afterwards, whenever Alice wants to send the message M over the secure channel, she'll compute $\tau_M \leftarrow \mathsf{cbcMAC}_{K_a}(M)$ and send the pair (M, τ_M) over the channel; Bob will check whether $\tau_M = \mathsf{cbcMAC}_{K_a}(M)$ and if so will conclude that Alice said M.

This is a pretty bad idea. Show how to use the values τ_0, M and τ_M to compose a message to Bob that will convince him Alice meant to say the two-block message (M, τ_M) instead of just M. Explain why your message will convince Bob that Alice meant to say (M, τ_M) rather than just M. Hint: try writing a recursive definition of CBC-MAC, and use the facts that for any string A, $A \oplus A = 0^{|A|}$ and $A \oplus 0^{|A|} = A$.

Since τ_M is just 128 random-looking bits, why is this a big deal?

(b) Sly's friend Sally notices the same attack on his scheme. She proposes a different method of authenticating (and encrypting) messages: ignore the key K_a. Instead, to authenticate and encrypt the message M, first compute $H(M)$ using SHA-256; then encrypt $(M, H(M))$ together, using AES-CTR encryption. So the message sent on the insecure channel would be $\mathsf{CTR\text{-}Encrypt}_{K_e}(M, H(M))$; Bob would decrypt the message using K_e, check that the last 256 bits of the plaintext are the hash of all of the previous bits, and accept the message if they are.

Show that this is also a bad idea: if Alice ever sends a ciphertext corresponding to the message M, where Eve knows M, Eve can generate a ciphertext corresponding to any message M', (of the same length as M) that Bob will accept. (For example, if Alice sends the message "ATTACK AT TEN AM" Eve can drop it and make Bob accept the message "GO BACK HOME BOB" instead.)

4. Protocol (an)droids. (16 pts) Two robots Artoo and C3-2-0 often fly on different starships and need to alert each other to their presence when their ships come in contact—otherwise they might accidentally blow each other up! They agree on a shared key K and a MAC algorithm that outputs 256-bit tags to use in the following protocol.

1. $A \longrightarrow C$: a random 256-bit string N_A and $\mathsf{MAC}_K(N_A)$.
2. C: on message n, t check that $\mathsf{MAC}_K(n) = t$, and if so, accept A, otherwise blow up the other party.
3. $C \longrightarrow A$: $\mathsf{MAC}_K(t)$.
4. A: on message t' check that $t' = \mathsf{MAC}_K(\mathsf{MAC}_K(N_A))$. If so, accept C, otherwise blow C up.

The idea here is that A proves it is A by correctly MACing N_A (which, if the key is secret, only A or C could do) and C proves it is C by MACing the MAC. But...

(a) A and C use this protocol for a while and then discover, to their dismay, that sometimes the evil galactic robo-emperor, E, has been successfully fooling C into believing it is A. Even supposing that robot-in-the-middle attacks are prevented by speed-of-light limitations or some other plot contrivance, what is a simple way for E to do this?

(b) A and C decide that one way to prevent the attack is for C to remember every value of N_A used in a previous challenge and reject if one is ever reused. Suppose E sees one authentication between A and C. How can it fool C into believing it is A as many times afterwards as it wants?

5. Hashing and Signing. (14 pts) Nearly every digital signature scheme works by first hashing a message to be signed (with a cryptographic hash function) and then performing some operation on the hash—so in essence, we are "signing the hash" and not the message. In particular, if Eve sees Alice's signature on the message M and can find a message $M' \neq M$ so that $H(M) = H(M')$, she can convince people that Alice signed M'. This is OK, since a good cryptographic hash function H will resist finding targeted collisions (second pre-images) like this.

Suppose our signature scheme uses a hash function H with an output length ℓ that is sufficient to resist second pre-images but NOT resistant to free collisions (e.g. the hash length is around 100-120 bits). Then it is possible that if Eve can get Alice to sign one of a pair of colliding messages, she can later claim that Alice signed the other.

The classic birthday attack works by hashing random messages until two have the same hash. This could already be a problem in some applications, but you might object that Alice is unlikely to agree to sign a random message. So let's think about how to create a collision with more specific messages.

Suppose that a message is "favorable" if it is something that Alice would sign, for example "I will pay \$10 to McDonald's for my lunch." Suppose that a message is "undesirable" if it is something that Alice would not sign, like "I will pay \$10,000,000 to Eve for her lunch." Notice that we can generate 256 different "favorable" messages from the example above, for instance by varying the number of space characters between words between 1 and 2. Extend this idea to show how to generate a pair of messages, one favorable and one undesirable, with the same hash. Your attack should compute about as many hashes as the birthday attack.

Then, describe how Eve completes the attack using the pair she generates to her advantage.