Massachusetts Institute of Technology
6.857: Network and Computer Security
Professors Ronald L. Rivest and Yael Tauman Kalai

Handout 4
March 7, 2016
Due: March 21, 2016

Problem Set 3

This problem set is due on *Monday, March 21, 2016* at **11:59 PM**. Please note our late submission penalty policy in the course information handout. Please submit your problem set, in PDF format, on Gradescope. *Each problem should be in a separate PDF.* Have **one and only one group member** submit the finished problem writeups. Please title each PDF with the Kerberos of your group members as well as the problem set number and problem number (i.e. *kerberos1_kerberos2_kerberos3_pset3_problem1.pdf*).

You are to work on this problem set with groups of your choosing of size three or four. If you need help finding a group, try posting on Piazza or email `6.857-tas@mit.edu`. You don't have to tell us your group members, just make sure you indicate them on Gradescope. Be sure that all group members can explain the solutions. See Handout 1 (*Course Information*) for our policy on collaboration.

Homework must be submitted electronically! Each problem answer must be provided as a separate pdf. Mark the top of each page with your group member names, the course number (6.857), the problem set number and question, and the date. We have provided templates for LaTeX and Microsoft Word on the course website (see the *Resources* page).

Grading: All problems are worth 10 points.

With the authors' permission, we may distribute our favorite solution to each problem as the "official" solution—this is your chance to become famous! If you do not wish for your homework to be used as an official solution, or if you wish that it only be used anonymously, please note this in your profile on your homework submission.

Our department is collecting statistics on how much time students are spending on psets, etc. For each problem, please give your estimate of the number of person-hours your team spent on that problem.

Problem 3-1. Security of "two-pass CBC"

In class we saw a definition of security for an encryption mode of operation. In particular, we would like a block cipher mode that provides indistinguishability under a chosen ciphertext attack (IND-CCA, defined in lecture 9 as a game with an adversary).

Consider the following two-pass CBC mode:

Given a key k and a message $M_1, M_2, \ldots, M_n$ (assume for convenience that the message is a multiple of the block size so that we don't have to worry about padding or ciphertext stealing), encryption works as follows:

Initialization
$A_0 =$ randomly chosen block
$A_i = M_i$ for $i = 1, 2, \ldots, n$

Pass 1
$B_0 = E_k(A_0)$
$B_i = E_k(A_i \oplus B_{i-1})$ for $i = 1, 2, \ldots, n$

Pass 2
$C_0 = E_k(B_0 \oplus B_n)$
$C_i = E_k(B_i \oplus C_{i-1})$ for $i = 1, 2, \ldots, n$

The output of the two-pass CBC mode encryption is $C_0, C_1, \ldots, C_n$.

(a) Explain how to decrypt $C_0, C_1, \ldots, C_n$.

(b) Does this encryption mode meet the security definition mentioned above (IND-CCA)? Explain why or why not.

Next, consider an alternative two-pass CBC mode in which we reverse the intermediate blocks before running the second pass. Concretely, the initialization and first pass are the same as defined above. The second pass is changed to the following:

Pass 2
$C_0 = E_k(B_0 \oplus B_n)$
$C_i = E_k(B_{n+1-i} \oplus C_{i-1})$ for $i = 1, 2, \ldots, n$

Again the outputs of this modified two-pass CBC mode encryption is $C_0, C_1, \ldots, C_n$.

(c) Explain how to decrypt $C_0, C_1, \ldots, C_n$ for this modified two-pass CBC mode.

(d) Does this encryption mode meet the security definition mentioned above (IND-CCA)? Explain why or why not. (*Hint: consider what happens when you modify one ciphertext block during decryption.*)

Problem 3-2. The Legend of SATelda

Ben Bitdiddle is designing a new game, which he calls the "The Legend of SATelda".

Ben has become enthusiastic about SAT-solver (`https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`) which have become amazingly efficient in recent years, able to solve SAT instances with large numbers of variables and clauses.

In Ben's new (single-player) game, there is a large secret K that the player is trying to find. Here K is a randomly chosen 256-bit secret key for that game. (Each game session has a fresh key.) When the player determines K, he can decrypt the Golden Orb and become Master of the Universe.

Every time the player defeats a demon in combat, he receives a random 3-variable clause consistent with the secret key K. More precisely: let the bits of K be denoted

$$K = k_1, k_2, ..., k_{256}$$

so each k_i is a bit.

A clause is a disjunction of three literals, where each literal is either a variable or its negation. The variables are the bits k_i.

For example, suppose the secret is

$$K = 0110100010....$$

Then when a player defeats a demon he may receive the clause $\{1, -3, -4\}$ (meaning (k_1 or (not k_3) or (not k_4))) which is true for the given K, since k_4 is false. Another consistent clause is $\{2, 5, -7\}$, which has two true literals.

A clause is generated randomly for K by picking three distinct positions out of $\{1, 2, ..., 256\}$, then choosing signs for those literals randomly (except for the one signage that isn't true for K), so each of the 7 valid signages for that clause is equally likely.

When the player collects enough clauses, he can apply his favorite SAT solver to solve for K. (He might need hundreds of clauses, but that's OK; the game is long...) Ben can "try to solve the Golden Orb" whenever he likes with a trial key K; there is no penalty for doing so. He can try as many times as he likes until he solves it.

(a) How is Ben's game like "secret sharing"? How is not like "secret sharing"?

(b) Suggest why the NP-completeness of 3-SAT is perhaps not a problem here. Or is it?

(c) MiniSAT (`https://www.msoos.org/2013/09/minisat-in-your-browser/`) is a SAT solver which, given a list of short clauses, can generate a valid SAT expression that satisfies all the clauses, if one exists.

How many demons might a player have to defeat before he can win the game? Specifically, suppose that the player wins if he determines $K = 111..111$. Run three trials of generating random 3-literal clauses, and determine how many are needed for the MiniSAT to give the correct value for K, i.e. the MiniSAT should output 0 conflict literals, and "SATISFIABLE v 1 2 3 .... 256 0".

Submit any code used to determine this in your pdf write-up, and give the number of demons the player had to have defeated in each of the three trials.

It may be useful to use binary search to determine how many of the clauses are needed. Assume that the min number of clauses you need is at least 86 (i.e., 256/3), and the max number of clauses you would need is 20000, to facilitate the binary search. Note that if x clauses was enough to win the game ever, you can put that as one of your trials. You may also round the number of clauses you need to the nearest 10. Don't feel restricted to using the browser MiniSAT function. You may also use a alternative versions online, e.g. a python SAT solver (`https://github.com/netom/satispy`).

Problem 3-3. Speck or Random? Distinguishing Reduced Round Speck

A common way to encrypt messages of variable length is to use a stream cipher. These stream ciphers are commonly built upon trusted block ciphers. One such block cipher is the Speck block cipher (`https://en.wikipedia.org/wiki/Speck_(cipher)`). It is remarkably simple, consisting of only 64-bit xor, addition, and cyclic shifts, giving it great performance on software platforms. Much cryptanalysis has gone into the Speck 128/128 cipher (which has 128-bit key and 128-bit block). No one has had any (public) success in recovering a secret key when more than 17 out of 32 rounds are run (`https://eprint.iacr.org/2014/320.pdf`). Let's see how we can do!

Instead of conducting key-recovery attacks, we will be looking at distinguishing Speck output from random.

One method for distinguishing a cipher is to use the Chi-Squared test `https://en.wikipedia.org/wiki/Pearson's_chi-squared_test#Discrete_uniform_distribution` . The Chi-Squared test helps determine how similar two distributions are. The Chi-Squared statistic is defined as

$$\chi^2 = \sum_{i=0}^{N-1} \frac{(E_i - O_i)^2}{E_i}$$

Where N is the number of possible events of a random variable, E_i is the expected value of event i, and O_i is the observed value of event i.

We will generate B blocks of Speck output by setting a random key and incrementing the input plaintext nonce for each block (CTR mode). We can leverage this Chi-Squared distribution by considering the random variable R_i = the number of times a specific byte in the 128-bit output has value i throughout all of the B total output blocks. Now we can use the Chi-Squared distribution between R_i and uniformly random bytes. In this case, $N = 256$, and $E_i = B/16$. We then compute O_i by storing counts of each value of our byte of interest in the Speck output. Finally, we compute χ^2 and decide the output is not random if χ^2 deviates by more than three standard deviations from the mean.

In our case, the Chi-Squared statistic has expected value and standard deviation

$$E(\chi^2) = N - 1$$

$$\sigma(\chi^2) = \sqrt{2(N-1)}$$

Use the Chi-Squared test to distinguish r-round Speck 128/128 for as high of an r as you can. You may generate as many ciphertext blocks as you wish (arbitrarily high B). Please submit your code, an explanation of your distinguishing techniques, and your results to Gradescope.

We have provided a python Speck implementation in the file speck.py, a test file challenge.py. And you'll implement distinguish.py. (Feel free to use implementations in another language if you wish, for more efficiency. But it is not required that you do so.)