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

Handout 3
March 9, 2021
Due: March 24, 2021

Problem Set 2

This problem set is due on *Wednesday, March 24, 2021* 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. *Your solutions to all problems should be written up in a single pdf.* Have **one and only one group member** submit the finished pdf containing the problem writeups. Please title the PDF with the Kerberos of your group members as well as the problem set number. (i.e. *kerberos1_kerberos2_kerberos3_pset2.pdf*).

You are to work on this problem set in groups. For problem sets 1, 2, and 3, we will randomly assign the groups for the problem set. After problem set 3, you are to work on the following problem sets 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-staff@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! 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.

Problem 2-1. Authentication

MACs are a powerful technique for achieving authentication, and are used to construct CCA-secure encryption schemes. In this problem, we shall focus on MAC schemes and their security. We will walk you through the basic idea behind the Galois Counter Mode (GCM). We won't cover GCM in lecture or recitation, but it is one of the industry standards of authenticated AES. It is quite popular, as it does a better job of leveraging parallel processing. If you would like to learn more about GCM, we encourage you to check out the following video: `https://www.youtube.com/watch?v=g_eY7JXOc8U`

(a) Consider the following MAC scheme: First, we associate with each message M the homogeneous polynomial $f_M(x) = M[1]x + M[2]x^2 + ... + M[L]x^L$ defined over the finite field $GF(2^{128})$, where $M[1], M[2], \ldots, M[L] \in \{0,1\}^{128}$ form a partition of the message M into chunks, each of length 128 bits (we assume that the length of M is a multiple of 128; as we saw in class, by padding M appropriately, this assumption is without loss of generality). Consider the MAC scheme defined by $MAC(a, b, M) = f_M(a) + b$, where (a, b) is the MAC key, where a and b are chosen independently at random from $GF(2^{128})$.

A MAC scheme is said to be one-time secure if a PPT adversary cannot generate an accepting MAC for any message M, after seeing a MAC for a *single* message $M' \neq M$ of its choice (except with negligible probability). Namely, it is one-time secure if it is secure according to the definition presented in class (secure against adaptive chosen message attacks) but where the adversary is restricted to query its oracle only once.

1. Argue that the MAC scheme defined above is one-time secure.
2. Suppose we fix b to be 0. Is the scheme now one-time secure? If so, explain why. If not, provide a counterexample.
3. Is the scheme many-time secure (namely, secure against adaptive chosen message attacks, as defined in class)? Explain why or why not.

(b) Let's look at a general method for converting a one-time secure MAC to a many-time secure MAC. Fix any one-time secure MAC scheme MAC and a pseudorandom function (PRF) F. Consider the *randomized* (many-time) MAC scheme MAC' defined by:

$$MAC'(k_1, k_2, M; r) = (r, F(k_1, r) \oplus MAC(k_2, M)),$$

where k_1 and k_2 are randomly and independently chosen keys, M is the message being MACed, and r is the randomness used to generate the MAC.

1. Show that MAC' is (many-time) secure.
2. Would MAC' remain (many-time) secure if we used keys k_1 and k_2 where $k_1 = k_2$? If yes, explain why. If not, provide a counterexample by constructing a one-time secure MAC and a PRF F, such that the corresponding MAC' is insecure.

Problem 2-2. Collisions!

In this problem, we will explore finding and analyzing collisions in greater detail. It is often desired that hash functions be collision-resistant. One hash function (designed by Prof. Rivest) that has been utterly broken in terms of its collision-resistance is the MD5 hash function.[1]

(a) Using the `fastcoll` software,[2] find a MD5 collision for two messages with an identical prefix. That is, select a prefix string p of your choice (such as $p =$ "crypto is fun!") and run `fastcoll` on your prefix to output two messages m_1 and m_2. m_1 and m_2 will be of the form $m_b = p||x_b$ where x is a binary blob generated by `fastcoll`. Importantly, $x_1 \neq x_2$ despite the fact that $MD5(m_1) = MD5(m_2)$.

Once you have m_1 and m_2 from the output of `fastcoll`, append a *suffix* to both files. As with the prefix, the suffix s can be a string of your choice, such as $s =$ "collision resistance is hard :(". Now you have messages $m_1 = p||x_1||s$ and $m_2 = p||x_2||s$. Does $MD5(m_1) = MD5(m_2)$ still?

For this part please submit your two files that were output by `fastcoll` and provide a short explanation of why appending a suffix to these files didn't change the fact that they share the same MD5 hash value. The filenames for your collisions should have the same format as your pdf: kerberos1_kerberos2_kerberos3_m1 and kerberos1_kerberos2_kerberos3_m2

Help with `fastcoll`: for all platforms, usage should be `fastcoll prefix_file` where "prefix_file" is a file containing your prefix text string. It will output two files, "md5_data1" and "md5_data2", that have the same MD5 hash and begin with the string from "prefix_file".

To install on Linux/Mac you will need: `git, cmake, gcc, g++`
First clone the repository using git: `git clone https://github.com/upbit/clone-fastcoll.git`
Next, while in the `clone-fastcoll` directory, build the `fastcoll` executable by running `make`.
If you have problems building or running `fastcoll` please ask on Piazza!

To concatenate two files you can use `cat`. Running `cat prefix suffix > out` will produce a file named "out" such that $out = prefix||suffix$.

To check the MD5 hash of two files you can use `openssl` as: `openssl dgst -md5 file1 file2`

(b) Your attack in the previous part generated a collision with two messages that shared the same prefix. However, it is also possible to generate collisions that do **not** share a common prefix. A *chosen prefix attack* allows the generation of collisions with different prefixes. In this case, an adversary could generate MD5 collisions such that $MD5(p_1||m_1) = MD5(p_2||m_2)$ where $p_1 \neq p_2$. Give an example of how this might be more advantageous to an adversary than an identical prefix attack and explain your reasoning.

[1] The MD in MD5 stands for "message digest," which is the older terminology used for secure hash functions — hence MD5 here, versus "secure hash" functions like SHA256.

[2] Linux/Mac: `https://github.com/upbit/clone-fastcoll`
Windows: `http://www.win.tue.nl/hashclash/fastcoll_v1.0.0.5.exe.zip`

To help you with this, please refer to the 2012 paper by Stevens *et al.*[3]

For the remainder of this problem, we turn to an interesting application: detecting a certain kind of malware. Cryptographic software is seldom coded new; it is typically obtained from libraries. It can be hard to tell if cryptographic software has been hacked or modified by an adversary who has access to it in the software supply chain, since it typically produces "random-looking bits."

How can you tell that it has been hacked? One way might be to notice that the software does not interoperate with other software libraries. That does not work, however, if the software operates on its own (e.g. think of a file-encryption application that executes without calls to software beyond itself).

Here we will theoretically analyze another way for detecting malicious modification with a specific compromise. Suppose we are using a library to implement $\texttt{Enc}(k, m)$, an encryption of message m with key k. Suppose for concreteness that m and k each have 128 bits. In a *key-shrinking attack* the provided (hacked) functionality actually implements $\texttt{Enc}(f(k), m)$, where $f(k)$ is some sort of compression or shrinking function, such as $f(k) = \texttt{Enc}(0, k) \bmod 2^t$, the rightmost t bits of the encryption of the "message" (the old key k) with "key" 0 (all zeros). Thus, $f(k)$ "looks random" but has only t bits of entropy. This is a nasty attack since encryption seems to work fine at a first glance, but an adversary only has to check 2^t keys to break the encryption.

(c) Assume messages and keys are n bits long, but the compromise shrinks the effective key space to size 2^t for some $t \leq n$. Suppose you wish to test some potentially-compromised encryption software for the presence of such key-shrinking malware. How much time and space is required to do so with a reasonable level of confidence? You may assume calls to the encryption software are a constant-time subroutine (the software should run in polynomial time, so your result here will be correct up to a polynomial factor).

(Hint: model keys as being drawn uniformly at random from all t-bit strings and apply the birthday paradox.)

Problem 2-3. AES and Finite Fields In AES, we think of the bytes as elements in the finite field $GF(2^8)$. In this problem, we will consider AES over a different finite field, and try to figure out what modifications (if any) are needed.

(a) As a warm up, we will first get some practice with finite field operations. For the following parts, the field in question is $GF(2^8)$ defined by the polynomial $m(x) = x^8 + x^4 + x^3 + x^1 + 1$. Note that this is the same field used in AES (albeit, as is true for all finite fields, there is a single finite field (up to isomorphism) with 256 elements). I.e., all operations take place in $GF(2^8)$, which are different than (normal) byte operations.

1. Compute 65 + D1 and 65 · D1 in $GF(2^8)$.
2. Calculate the (multiplicative) inverse of 4B in $GF(2^8)$.

For the last part of this problem, we will redesign AES to use 16-bit values instead of 8-bit values. Recall that AES operates on a 4×4 array of bytes (elements of $GF(2^8)$) called the State. It gets initialized with the 128 bits of the input message, and its final state corresponds to the 128 output bits (the ciphertext). In this problem, we will consider a variant of AES where the state is a 4×4 array of elements of $GF(2^{16})$ instead of $GF(2^8)$ (so, the block size is now $4 \times 4 \times 16 = 256$ bits).

The AES S-box works by first computing the inverse of a non-zero byte x (if the byte is zero, use x directly instead): $S_0(x) \coloneqq 0$ if $x = 0$, else x^{-1}.

Then, an affine transformation T is applied to $S_0(x)$. So, the mapping defined by the S-box is $S(x) \coloneqq T(S_0(x))$. Two important properties satisfied by S are that $S(x) \neq x$ (no fixed points) and $S(x) \oplus x \neq \text{FF}$ (no "opposite-fixed" points).

[3]Stevens, Marc, Arjen K. Lenstra, and Benne de Weger, "Chosen-prefix collisions for MD5 and applications," Int. J. Applied Cryptography, Vol. 2, No. 4, 2012. `https://documents.epfl.ch/users/l/le/lenstra/public/papers/lat.pdf`

(b) We consider a transformation S defined a bit differently on $GF(2^{16})$ as follows, using S_0 defined as above (but for elements x in $GF(2^{16})$):

$$S(x) = f^{-1}(f(x) \oplus c)$$

where c is a fixed non-zero constant in $GF(2^{16})$, and where $f = S_0$ is a one-to-one function defined on $GF(2^{16})$.

Argue that S has no fixed-points for any one-to-one function f.

Argue that c can be chosen so that S has no opposite fixed-points, either. (For the definition of opposite fixed points, we use $FFFF$ rather than FF, since we are working over $GF(2^{16})$ now; no opposite-fixed points now means $S(x) \oplus x \neq$ FFFF).