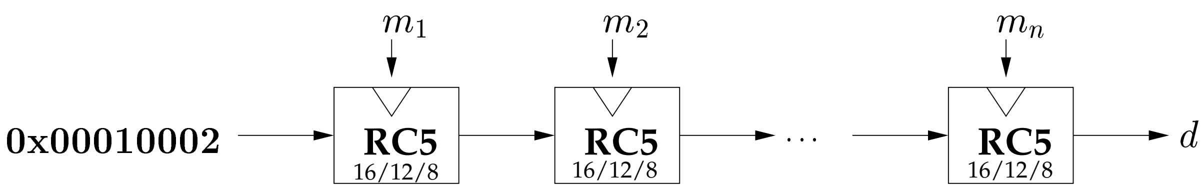
Massachusetts Institute of Technology — Handout 8
6.857: Network and Computer Security — September 25, 2001
Professor Ronald L. Rivest

Problem Set 1 Solutions

Problem 1-1. Hash functions

The hash function is described by the picture below. The input message m is broken into 8-byte blocks $m = m_1, m_2, ..., m_n$. The last block is padded with zeros on the right as needed. The first block, m_1, is used as the key to encrypt the initial vector `0x00010002` using the RC5-16/12/8 cipher. Each subsequent message block is used as the key to encrypt the output of the previous encryption operation. The message digest, $h(m) = d$, is the result of the final encryption operation. RC5-16/12/8 is a cipher that operates on two 16-bit words at a time, has 12 rounds, and requires an 8-byte key.



(a) Describe an attack that inverts this hash function.

Cristian Cadar, Cãtãlin Frâncu, and Ovidiu Gheorghioiu gave us this solution:

Because this hash function involves breaking the message into blocks, and, moreover, the RC5 cipher allows decryption, we can mount the following attack:

Let D_{start} be the start digest (0x00010002 in this case) and D_{stop} be the desired digest. Consider a set A of pairs (D, M), such that D_{start} encrypts to D using the corresponding, 8-byte message M as a key. Similarly, consider a set B of pairs (D, M) such that D encrypts to D_{stop}, using the corresponding, 8-byte message M as a key. If a digest D appears in both A and B (that is, there is a pair (D, M_1) in A and a pair (D, M_2) in B), then we have inverted the hash function for the case of D_{stop}: the 16-byte concatenation of M_1 and M_2 will hash D_{start} to D_{stop}. If the sets A and B are large enough, there is a high probability that such a D exists.

In the proposed attack, we start with two empty sets A and B. At each step, we generate a random, 8-byte message M, we encrypt D_{start} with key M into D_1, and we decrypt D_{stop}, again with key M, into D_2. It follows that we can add the pairs (D_1, M) and (D_2, M) to A and B, respectively. We then check whether D_1 is present in a pair of B, or, alternatively, whether D_2 is present in a pair of A. If either of these is true, then A and B contain a common digest and, as explained above, we have inverted the function. The figure below gives an example.

In this figure, if D_{start} encrypts to D_1 using the key M_1, and if $D_2' = D_1$ encrypts to D_{stop} using the key M_2, then the 16-byte message $m = (M_1, M_2)$ breaks the code.

If we use a constant-time set membership operation (e.g., a hash table), then the running time of this algorithm is simply $O(N)$, where N is the number of operations used. In practice, the program always found a solution within 1 second. How come? A probabilistic analysis follows.

We want to assess $P(N)$, the probability that we find an inversion in N attempts or less. A bit of dynamic programming helps. Obviously, $P(0) = 0$ – we cannot expect to win without running any trials! Now three things may happen at the $N + 1^{st}$ trial:

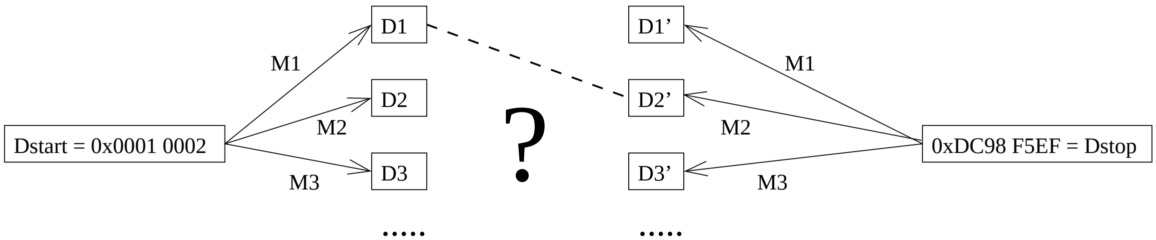


Figure 1: Our evil plot

- We may have already found a hit in the first N experiments;
- D_{N+1} is equal to one of $D'_1, D'_2, \ldots, D'_N$. Because D_i are pseudo-random 32-bit numbers, each equality happens with probability 2^{-32}.
- D'_{N+1} is equal to one of $D_1, D_2, \ldots, D_{N+1}$.

Consequently, $P(N+1) = P(N) + [1 - P(N)](Np + (1 - Np)(N+1)p)$, where $p = \frac{1}{2^{32}}$. The consequences are most remarkable. The table below shows that most of the times we can crack the code with less than 100,000 experiments – a trifle for today's computational power.

A slightly more intuitive way to model the probability is that $P(N) = 1 - (1 - \frac{N}{q})^N$ where $q = 2^{32}$. Consider the probability of failure of the first member of set B matching anything in set A. This is $\frac{q-N}{q} = 1 - \frac{N}{q}$ since the member from set B can be one of q values and N of these values will result in success. Now consider the probability of failure that none of the members of B match with any member of A. Because each comparison is independent and the size of both sets is N, this probability is $(1 - \frac{N}{q})^N$. Hence the probability of success is 1 minus that value $= P(N) = 1 - (1 - \frac{N}{q})^N$.

Table 1: Probability versus the number of trials

Number of trials [N]	Probability of success [$P(N)$]
0	0.0000
20000	0.0889
40000	0.3110
60000	0.5675
80000	0.7746
100000	0.9025
120000	0.9650
140000	0.9895
160000	0.9974
180000	0.9994
200000	0.9999

(b) Implement your attack and find a message that hashes to `0xDC98 F5EF`.

Jessica Huang, Kai Huang, Edward Huang, and Melissa Shi gave us this entertaining set of hash inversions:

We implemented our attack in C. The commented code and sample output are available on the 6.857 Web page.

(Extra!) We wanted to find meaningful input messages that hashed to the target message vector. We refined our program to select keys from a list of common English words, all less than eight letters long. We also changed the message length to 32 characters. This new code is also attached for reference, but it is not commented. The following are some interesting four-word phrases that hash to the target.

```
"Shelat  with    rivest  nasty   "    "peace   treaty     too  swiss   "
"Kevin   drummed   kevin pointed "    "states     will  debate vitally "
"  boy   kevin       fu   large  "    "protect    who     what   from  "
"  this  lousy   earthly  client "    "cakes       for      a      fox "
"gallium revise  kitty's anatomy "    "martian clatter   beeps  widely "
"washing  all      with  might   "    " not       used  passive    yet  "
"praise   tries     all  women   "    "   her  things  spun    through "
"now      test     on     adults "
```

Problem 1-2. Le BigMAC

$$
\begin{aligned}
\mathrm{BigMAC}_k(u,e) &= (u \oplus \mathrm{AES}_k(e), (\mathrm{AES}_k(u \oplus \mathrm{AES}_k(e))) \oplus e) \\
&= (u \oplus f, g \oplus e) \\
&= (l, h)
\end{aligned}
$$

The Web site will allow you to create new accounts and set passwords for any username not already created. The Web site allows both printable and non-printable characters in usernames. The account login process can be modeled as a function α which takes as input a username and password to produce a cookie-based authenticator:

$$
\alpha(u, password) \rightarrow \begin{cases} (u, e, (l,h)), & \text{if the password is correct for the username } u \\ & \text{where } e \text{ is a timestamp[1]} \\ & \text{and } (l,h) = \mathrm{BigMAC}_k(u,e) \\ \text{INVALID}, & \text{otherwise} \end{cases}
$$

After logging in and receiving a cookie-based authenticator, one can model the verification routine on the BSJ Web server as:

$$
\omega(\text{cookie}) \rightarrow \begin{cases} \text{true}, & \text{if cookie can be unmarshalled to } (u, e, (l,h)) \\ & \text{and if } e > \text{the current time} \\ & \text{and } \mathrm{BigMAC}_k(u,e) = (l,h) \\ \text{false}, & \text{otherwise} \end{cases}
$$

[1] You can assume the timestamp is in seconds since UNIX epoch time approximately 2 hours from the time of login. This detail is not important for the break.

There were many solutions to this problem, but all involved exploitation of the commutative property of XOR in an adaptive chosen message attack. Timo Burkard, John Gu, and Kenneth Yu provided this solution:

(a) Explain using the notation above how an interrogative adversary can make an existential forgery for this user authentication scheme.

Assumptions:

- We can create a new username.
- For our existential forgery, the username that we forge will be such that it consists of 128 bits that are practically garbage.

Description of the attack:

1. Create an Account for some user that doesn't exist yet.
2. Obtain a cookie for that user.

That cookie contains our username u and an expiration time e that has been issued by the server. Furthermore, we are given $\text{BigMAC}_k(u, e) = (l, h) = (u \oplus \text{AES}_k(e), (\text{AES}_k(u \oplus \text{AES}_k(e))) \oplus e)$

Next, we create a second account and obtain a cookie for that account, consisting of u', e', and its $\text{BigMAC}_k(l', h')$.

Next, we will construct a forged cookie that will grant access to some other user account that we haven't created (and that in fact will turn out to be have a rather awkward username, see assumption above).

For the forged cookie, our expiration time e_f we will use the expiration time of the second cookie that we obtained, e'. Since we know u', $\text{AES}_k(e')$ can easily be computed using the formula $\text{AES}_k(e') = l' \oplus u'$. Hence, since $e_f = e'$, $\text{AES}_k(e_f) = l' \oplus u'$.

Using the same identity, we can easily figure out $\text{AES}_k(e)$, it is simply $l \oplus u$.

In order to generate the second part of the BigMAC for our forged cookie, h_f, we need to have $\text{AES}_k(u_f \oplus \text{AES}_k(e_f))$. However, since we don't know k, we cannot calculate this function ourselves - we use some value of that function provided by the server. So we simply chose u_f such that $u_f \oplus \text{AES}_k(e_f) = e$ (so that the AES that we have to compute is simply $\text{AES}_k(e)$), i.e. $u_f = e \oplus \text{AES}_k(e_f)$. ($\text{AES}_k(e_f)$ has already been calculated, see above).

Now we can very easily calculate the BigMAC of the forged cookie, (l_f, h_f):

$l_f = u_f \oplus \text{AES}_k(e_f) = u_f \oplus (l' \oplus u')$ (see definition of e_f above).

$h_f = (\text{AES}_k(u_f \oplus \text{AES}_k(e_f)) \oplus e_f = \text{AES}_k(e) \oplus e_f$.

Notice that with very high probability, u_f will be different from u and u'. Furthermore, the cookie that we forged is valid. Therefore, we succeeded in making an existential forgery.

(b) Explain how an interrogative adversary can make a selective forgery for this user authentication scheme if the account login process is instead:

$$\alpha(u, password, e) \rightarrow \begin{cases} (u, e, (l, h)), & \text{if the password is correct for the username } u \\ & \text{where } (l, h) = \text{BigMAC}_k(u, e) \\ \text{INVALID}, & \text{otherwise} \end{cases}$$

Assumptions:

- When creating cookies, we may transmit any 128-bit sequence as the desired expiration time e to the server.

Attack to create a cookie for username u_f with expiration time e_f without knowing the password of that account:

Create a new account for user u. Then, have the server create a cookie for that user with expiration time e_f. Suppose that cookie has the $\text{BigMAC}_k(u, e_f) = (l, h)$. Now, by looking at the equations in the problem statement, one can easily see that $\text{AES}_k(e_f) = l \oplus u$.

Next, we compute $u_f \oplus \text{AES}_k(e_f)$. This is trivial since both u_f and $\text{AES}_k(e_f)$ are known to us. Let's call the resulting value x. However, in order to compute the correct MAC for the forged cookie (specifically, the second component of the MAC), we need to know $\text{AES}_k(x)$.

Since we don't know k, it is impossible for us to directly compute the hash function. However, we can use the following trick and get the server to compute it for us:

For our own user account u, we simply request the server to create a cookie with expiration time x. (which is possible, by our assumptions stated above.)

Suppose the cookie returned for that user and expiration time has the $\text{BigMAC}_k(l', h')$. By the same argument that I used above, $\text{AES}_k(x) = u \oplus l'$. Therefore, we now know $\text{AES}_k(x)$.

Computing the $\text{BigMAC}_k(l_f, h_f)$ for our forged cookie with username u_f and expiration time e_f is now pretty straightforward:

$l_f = u_f \oplus \text{AES}_k(e_f) = u_f \oplus (l \oplus u)$

$h_f = (\text{AES}_k(u_f \oplus \text{AES}_k(e_f))) \oplus e_f = \text{AES}_k(x) \oplus e_f = (u \oplus l') \oplus e_f$.

By construction, this forged cookie is valid. Hence we managed to perform a selective forgery by creating a valid cookie for a user u_f and an expiration time e_f, without knowing the user's password.