CSCE 465 : Hash Function and Public Key Cryptography

Due on 18 November, 2019

Gu Fall 2019

M. Hunter Martin
825002697

Written Problems

5.2 (8 pts)

> Message digests are reasonably fast, but here's a much faster function to compute. Take your message, divide it into 128-bit chunks, and *xor* all the chunks together to get a 128-bit result. Do the standard message digest on the result. Is this a good message digest function?

No. It would be relatively easy to find collisions and that would ruin the security of the function.

5.14 (12 pts)

> For purposes of this exercise, we will define random as having all elements equally likely to be chosen. So a function that selects a 100-bit number will be random if every 100-bit number is equally likely to be chosen. Using this definition, if we look at the function "+" and we have two inputs, x and y, then the output will be random if at least one of x and y are random. For the following functions, find sufficient conditions for x, y and z under which the output will be random:

1) $\sim x$

If x is random, $\sim x$ will also be random. y and z are independent.

2) $x \oplus y$

If either x or y are the random, and they are not equal, the output will be random. z is independent.

3) $x \vee y$

If either x or y are the random, and they are not equal, the output will be random. z is independent.

4) $x \wedge y$

If either x or y are the random, and they are not equal, the output will be random. z is independent.

5) $(x \wedge y) \vee (\sim x \wedge z)$ **[the selection function]**

If $(x \wedge y)$ is non-zero, or $(\sim x \wedge z)$ is non-zero, the output should be random assuming at least one of the inputs is also random.

6) $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ **[the majority function]**

As long as either x, y, or z are different, and at least one is random, then the output should also be random.

7) $x \oplus y \oplus z$

As long as at least 2 of the numbers are different (i.e. x and y, or x and z, etc.) and at least one of the unique numbers is random, the output will be random.

5.14 (12 pts) continued on next page...

8) $y \oplus (x \vee \sim z)$

If $(x \vee \sim z) \neq y$, the output will be random if at least one of the numbers is random.

6.2 (8 pts)

> In the KPS textbook, it states that encrypting the Diffie-Hellman value with the other side's public key prevents the man-in-the-middle attack. Why is this the case, given that an attacker can encrypt whatever it wants with the other side's public key?

The main point of Public-Key Encryption is for private key distribution. The Private key is needed the decrypt the Diffie-Hellman value, so the Man-in-the-Middle won't be able to decrypt the key.

6.8 (8 pts)

> Suppose Fred sees your RSA signature on m_1 and on m_2 (i.e., he sees $m_1^d \bmod n$ and $m_2^d \bmod n$). How does he compute the signature on each of $m_1^j \bmod n$ (for positive integer j), $m_1^{-1} \bmod n$, $m_1 m_2$ nod n, and in general $m_1^j m_2^k \bmod n$ (for arbitrary integers j and k)?

m_1^j mod n (for positive integer j)?

Signature $= (m_1^d)^j \bmod n = (m_1^j)^d \bmod n$

m_1^{-1} mod n?

Signature $= (m_1^d)^{-1} \bmod n = (m_1^{-1})^d \bmod n$

$m_1 m_2$ mod n?

Signature $= (m_1 m_2)^d \bmod n = (m_1^d \bmod n)(m_2^d \bmod n) \bmod n.$

$m_1^j m_2^k$ mod n (for arbitrary integers j and k)?

Signature $= (m_1^j m_2^k)^d \bmod n = (m_1^d \bmod n)^j (m_2^d \bmod n)^k \bmod n.$

Task 1: Generating Message Digest and MAC [7 pts]

The following is my input.txt.

```
The FitnessGram Pacer Test is a multistage aerobic capacity test
that progressively gets more difficult as it continues. The 20
meter pacer test will begin in 30 seconds. Line up at the start.
The running speed starts slowly, but gets faster each minute after
you hear this signal. [beep] A single lap should be completed each
time you hear this sound. [ding] Remember to run in a straight
line, and run as long as possible. The second time you fail to
complete a lap before the sound, your test is over. The test will
begin on the word start. On your mark, get ready, start.
```

MD5 Message Digest

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -md5 input.txt
MD5(input.txt)= 0e8385f49f93d6214d19a0e8f023f979
```

The hash function generated a hash from my input. The hash was somewhat short.

SHA1 Message Digest

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha1 input.txt
SHA1(input.txt)= 9b5f924bbe67ee9e0bcf9115c13fe401dbb00cd7
```

The hash function generated a hash from my input. The hash was somewhat short, but longer than the MD5 hash. This is likely part of what makes this hash function more secure than the MD5 hash.

SHA256 Message Digest

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 input.txt
SHA256(input.txt)= 8b0bc4ec9ac98fcfbd607e90f3defde015e024f8656a0feae80f70c2a838ab9b
```

The hash function generated a hash from my input. The hash was much longer than the MD5, and somewhat longer than the SHA1, which probably explains why it's considered so much more secure.

Task 2: Keyed Hash and HMAC [9 pts]

The following is my input.txt.

```
The FitnessGram Pacer Test is a multistage aerobic capacity test
that progressively gets more difficult as it continues. The 20
meter pacer test will begin in 30 seconds. Line up at the start.
The running speed starts slowly, but gets faster each minute after
you hear this signal. [beep] A single lap should be completed each
time you hear this sound. [ding] Remember to run in a straight
line, and run as long as possible. The second time you fail to
complete a lap before the sound, your test is over. The test will
begin on the word start. On your mark, get ready, start.
```

MD5 Hash

MD5 Hash with the standard key,

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -md5 -hmac "abcdefg" input.txt
HMAC-MD5(input.txt)= e3f324753bf7944670464bde7e61ccfa
```

and again with an extended key.

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -md5 -hmac "abcdefghijklmnop" input.txt
HMAC-MD5(input.txt)= 3ab06d179cb271ae0b0cc463d7d7f08c
```

SHA1 Hash

SHA1 Hash with the standard key,

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha1 -hmac "abcdefg" input.txt
HMAC-SHA1(input.txt)= 0f7ba25ae538f3b10067d8ae25b89a4ce537e4f6
```

and again with an extended key.

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha1 -hmac "abcdefghijklmnop" input.txt
HMAC-SHA1(input.txt)= 156af9b931a81594209afae861bd6c639e6997a3
```

SHA256 Hash

SHA256 Hash with the standard key,

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 -hmac "abcdefg" input.txt
HMAC-SHA256(input.txt)= 00c58e5ea38f0f4da78a32b64ac23669d400b1818877da3b3ed3425b619792db
```

and again with an extended key.

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 -hmac "abcdefghijklmnop" input.txt
HMAC-SHA256(input.txt)= cc000cbaacfb6915c8e66ebfc7979a06be0834ee553e2849d5501810bbbd67a6
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 -hmac "abcdefghijklmnop" input.txt
HMAC-SHA256(input.txt)= cc000cbaacfb6915c8e66ebfc7979a06be0834ee553e2849d5501810bbbd67a6
```

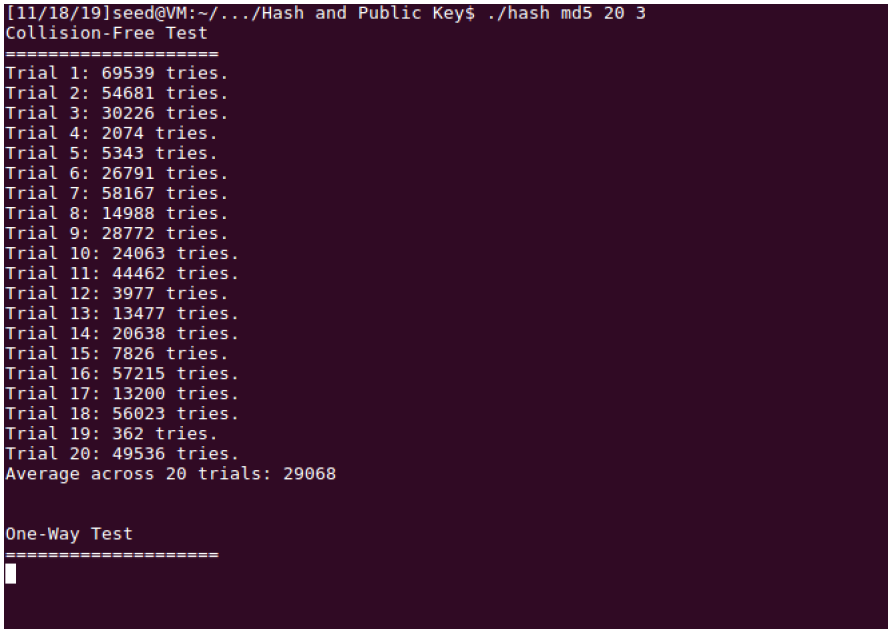
The keys didn't seem to affect the length of the final output, and the output was consistent between runs. This leads me to believe that the Key Size did not affect the hash, and a fixed key is not required.

Task 3: The Randomness of One-way Hash [9 pts]

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -md5 -hmac "abcdefg" input.txt
HMAC-MD5(input.txt)= e3f324753bf7944670464bde7e61ccfa
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -md5 -hmac "abcdefg" input_modified.txt
HMAC-MD5(input_modified.txt)= 83076d2e2962fc4b0b4288f9daf026eb
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 -hmac "abcdefg" input.txt
HMAC-SHA256(input.txt)= 00c58e5ea38f0f4da78a32b64ac23669d400b1818877da3b3ed3425b619792db
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 -hmac "abcdefg" input_modified.txt
HMAC-SHA256(input_modified.txt)= 3b2ec4167f111f92aaa698ca7de21faac6b7f8af2e6052ce7a9dde00d67edc87
```

The difference between the two with respect to each hash type is obvious. When we flipped even just one bit in the input, the entire output is totally different. This demonstrates out even a small change in the input generates a totally different output with these secure hashes.

Task 4: Hash Collision-Free Property [20+10 bonus pts]



The average in the first test took about 29068 tries on average, with a maximum near 70000, and a minimum of 362. The average of the one-way test was much higher, well into the hundreds of millions of tries. My implementation does work, but takes a very long time to find a collision. This is a testament to just how difficult it is to find a specific collision. Finding any collision is easy, but a specific one is very difficult.

Proof of the Difference (+10 bonus points)

The proof of this is similar to what we call *the Birthday Paradox.* In the Birthday Paradox, we can prove than in a classroom of 23 students, there is a 50-50 chance that *a student* will have the same birthday as *some other student.* With 75 students, there will be a 99.97% chance of this collision occurring. This is true because we consider the chance that every single student has a birthday in common with every other student. The odds in each are low, but when you have to multiply the inverse of each of these odds together, we eventually get a relatively good chance, and with only a few more students, we can all but guarantee a collision will occur.

This same paradox holds to hash function collisions. In the test for finding just any collision, we found them relatively quickly, with only 30000 tries on average, but in the specific hash (one-way) collision, it's much harder. We are only going to have an astronomically low chance each time we try. The average was well into the hundreds of millions. In the below test, I simulated the easier collision problem by giving up and retrying after 100,000 tries.

```
[11/18/19]seed@VM:~/.../Hash and Public Key$ ./hash md5 1 3000
Collision-Free Test
====================
Trial 1: 3752 tries.
Average across 1 trials: 3752


One-Way Test
====================
Trial 1: 0 tries.
Trial 2: 0 tries.
Trial 3: 0 tries.
Trial 4: 0 tries.
Trial 5: 0 tries.
Trial 6: 0 tries.
Trial 7: 0 tries.
Trial 8: 25982 tries.
Trial 9: 0 tries.
Trial 10: 0 tries.
```

We again see the lower level. The birthday paradox is great for finding just a collision, but luckily the security is achieved with a *specific* hash, so our hashes remain safe with their super-low chance of being collided with.

Task 5: Performance Comparison: RSA versus AES [8 pts]

RSA

Encrypting...

```
huntermartin@Hunters-Macbook-Pro Desktop % time openssl rsautl -encrypt -inkey public_key.pem -pubin -in message.txt -out message_encrypted.txt
openssl rsautl -encrypt -inkey public_key.pem -pubin -in message.txt -out   0.00s user 0.00s system 77% cpu 0.004 total
```

Decrypting...

```
huntermartin@Hunters-Macbook-Pro Desktop % time openssl rsautl -decrypt -inkey private_key.pem -in message_encrypted.txt -out message_decrypted.txt
Enter pass phrase for private_key.pem:
openssl rsautl -decrypt -inkey private_key.pem -in message_encrypted.txt -out  0.00s user 0.00s system 1% cpu 0.449 total
```

Bench-marking...

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl speed rsa
Doing 512 bit private rsa's for 10s: 210206 512 bit private RSA's in 9.98s
Doing 512 bit public rsa's for 10s: 1585692 512 bit public RSA's in 9.98s
Doing 1024 bit private rsa's for 10s: 56044 1024 bit private RSA's in 9.99s
Doing 1024 bit public rsa's for 10s: 611147 1024 bit public RSA's in 9.98s
Doing 2048 bit private rsa's for 10s: 9298 2048 bit private RSA's in 9.92s
Doing 2048 bit public rsa's for 10s: 179666 2048 bit public RSA's in 9.92s
Doing 4096 bit private rsa's for 10s: 1268 4096 bit private RSA's in 9.97s
Doing 4096 bit public rsa's for 10s: 48719 4096 bit public RSA's in 9.95s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000047s 0.000006s  21061.8 158946.7
rsa 1024 bits 0.000178s 0.000016s   5608.6  61244.8
rsa 2048 bits 0.001067s 0.000055s    937.4  18117.5
rsa 4096 bits 0.007863s 0.000204s    127.2   4898.7
huntermartin@Hunters-Macbook-Pro Desktop %
```

During the individual tests, the encryption and decryption occured very quickly, but decryption did seem to take a bit longer.

During bench-marking, we can see that the average speed for an RSA operation is dependent on both the size of the key (shorter keys are much faster), and public is always much faster than private.

AES

Encrypting...

```
huntermartin@Hunters-Macbook-Pro Desktop % time openssl aes-128-cbc -in message.txt -out aes_message_encrypted.txt
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
openssl aes-128-cbc -in message.txt -out aes_message_encrypted.txt  0.00s user 0.00s system 1% cpu 0.240 total
```

Bench-marking...

```
huntermartin@Hunters-Macbook-Pro Desktop % openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 28568846 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 64 size blocks: 8051455 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 256 size blocks: 2061775 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 1024 size blocks: 519071 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 8192 size blocks: 63181 aes-128 cbc's in 2.99s
Doing aes-192 cbc for 3s on 16 size blocks: 23501056 aes-192 cbc's in 2.98s
Doing aes-192 cbc for 3s on 64 size blocks: 6546412 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 256 size blocks: 1649312 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 1024 size blocks: 418264 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 8192 size blocks: 51598 aes-192 cbc's in 2.99s
Doing aes-256 cbc for 3s on 16 size blocks: 20191640 aes-256 cbc's in 2.98s
Doing aes-256 cbc for 3s on 64 size blocks: 5517031 aes-256 cbc's in 2.99s
Doing aes-256 cbc for 3s on 256 size blocks: 1417857 aes-256 cbc's in 2.99s
Doing aes-256 cbc for 3s on 1024 size blocks: 351566 aes-256 cbc's in 2.98s
Doing aes-256 cbc for 3s on 8192 size blocks: 44296 aes-256 cbc's in 2.98s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
aes-128 cbc     152434.40k   171616.44k   175754.76k   176983.62k   173072.76k
aes-192 cbc     126241.42k   139941.94k   141135.53k   143133.25k   141401.36k
aes-256 cbc     108562.20k   118181.52k   121425.82k   120945.78k   121857.00k
huntermartin@Hunters-Macbook-Pro Desktop %
```

As we noted with the RSA encryption, the speed was really too fast to measure for encryption, likely only a few hundredths of a second.

During bench-marking, we noticed encryption does seem to occur faster than with RSA, but is still highly dependent on the key size and on the block-size. The larger the blocks and the larger the key, the fewer encryptions we can do in our 3 seconds.

Task 6: Create Digital Signature [7 pts]

```
huntermartin@Hunters-Macbook-Pro Desktop % cat message.txt
ac14bd8101e2226f
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 -sign private_key.pem -out sign.sha256 message.txt
Enter pass phrase for private_key.pem:
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 -verify public_key.pem -signature sign.sha256 message.txt
Verified OK
huntermartin@Hunters-Macbook-Pro Desktop % vim message.txt
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 -verify public_key.pem -signature sign.sha256 message.txt
Verified OK
huntermartin@Hunters-Macbook-Pro Desktop % vim message.txt
huntermartin@Hunters-Macbook-Pro Desktop % openssl dgst -sha256 -verify public_key.pem -signature sign.sha256 message.txt
Verification Failure
huntermartin@Hunters-Macbook-Pro Desktop %
```

Digital signatures can ensure that the message you're receiving hasn't been tampered with on the way to its destination. Even a slight change will cause the verification to fail. In the above screenshot, you can see that I create a signature from my own private key, then my friend can verify my message with my public key, effectively blocking the Man-in-the-middle attack. While it wouldn't be a solution to stopping the man-in-the-middle from reading my message, I could stop him from changing my message.

Attachments

[1] `task4.c` - C code for hash collisions. Usage will be printed when ran without arguments.

All pictures included in this report are also included in the Report/Source folder in the submission.

References

[1] Gu. Homework 5: Hash Function and Public Key Cryptography.

[2] Openssl. https://www.openssl.org/docs/manmaster/man3/EVP_DigestInit.html