Lecture Notes, Week 9

1 Common Hash Functions

Many cryptographic hash functions are currently in use. For example, the openssl library includes implementations of MD2, MD4, MD5, MDC2, RIPEMD, SHA, SHA–1, SHA–256, SHA–384, and SHA–512. The SHA–xxx methods are recommended for new applications, but these other functions are also in widespread use.

1.1 SHA–1

The revised Secure Hash Algorithm (SHA–1) is one of four algorithms described in U. S. Federal Information Processing Standard FIPS PUB 180–2 (Secure Hash Standard). (See http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.) It states,

> "Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits)."

SHA–1 produces a 160-bit message digest. The other algorithms in the SHA–xxx family produce longer message digests.

1.2 MD5

MD5 is an older algorithm (1992) devised by Rivest. We present an overview of it here. It generates a 128-bit message digest from an input message of any length. It is built from a basic block function g : 128-bit $\times$ 512-bit $\rightarrow$ 128-bit.

The MD5 hash function h is obtained as follows: First the original message is padded to length a multiple of 512. The result m is split into a sequence of 512-bit blocks $m_1, m_2, \ldots, m_k$. Finally, h is computed by chaining g on the first argument.

We look at these steps in greater detail. As with block encryption, it is important that the padding function be one-to-one, but for a different reason. For encryption, the one-to-one property is what allows unique decryption. For a hash function, it prevents there from being trivial colliding pairs. For example, if the last partial block is simply padded with 0's, then all prefixes of the last message block will become the same after padding and will therefore collide with each other.

The function h can be regarded as a state machine, where the states are 128-bit strings and the inputs to the machine are 512-bit blocks. The machine starts in state s_0, specified by an initialization vector IV. Each input block m_i takes the machine from state s_{i-1} to new state $s_i = g(s_{i-1}, m_i)$. The last state s_k is the output of h, that is,

$$h(m_1 m_2 \ldots m_k) = g(g(\ldots g(g(IV, m_1), m_2) \ldots, m_{k-1}), m_k).$$

The basic block function $g(s, b)$ consists of 4 stages, each consisting of 16 substages. Recall that b is 512-bits long, so we may divide b into 32-bit words $b_1 b_2 \ldots b_{16}$. At stage i, substage j, a permutation π_i of $\{1, \ldots, 16\}$ is used to select word b_ℓ, where $\ell = \pi_i(j)$. A new state is generated by computing $f_{i,j}(s, b_\ell)$, where s is the old state and $f_{i,j}$ is a bit-scrambling function that depends on i and j. Since a state can be represented by four 32-bit words, the arguments to $f_{i,j}$ occupy only 5 machine words, which easily fit into the high-speed registers of modern processors.

2 Extending Fixed-Length Hash Functions

We now consider general hash functions and some techniques for manipulating them.

2.1 Doubling the Reduction Amount

Suppose we are given a particular fixed-length hash function h : 256-bits $\rightarrow$ 128-bits. How can we use h to compute a 128-bit strong collision-free hash of a 512-bit input block? We consider several possible ways to extend h to a hash function H : 512-bits $\rightarrow$ 128-bits. In the following, we suppose that m is 512-bits long, and we write $m = m_1 m_2$, where m_1 and m_2 are 256 bits each.

Method 1 Define $H(m) = H(m_1 m_2) = h(m_1) \oplus h(m_2)$. Unfortunately, this fails to be either strong or weak collision-free since $m' = m_2 m_1$ always collides with m under H (except in the special case that $m_1 = m_2$.

Method 2 Define $H(m) = H(m_1 m_2) = h(h(m_1)h(m_2))$.

Theorem 1 *The H of method 2 is strong collision-free assuming that the h from which it is derived is strong collision-free.*

Proof: Assume the contrary, that one can find a colliding pair (m, m') for H. We show that one can then find a colliding pair for h, contradicting the assumption that h is strong collision-free.

Write $m = m_1 m_2$ and $m' = m'_1 m'_2$ for 256-bit blocks m_1, m_2, m'_1, m'_2. Since m collides with m', we have that $m \neq m'$ but $H(m) = H(m')$. We consider two cases.

Case 1: $h(m_1) \neq h(m'_1)$ or $h(m_2) \neq h(m'_2)$. Let $u = h(m_1)h(m_2)$ and $u' = h(m'_1)h(m'_2)$. Then $u \neq u'$, but $h(u) = H(m) = H(m') = h(u')$, so (u, u') is a colliding pair for h.

Case 2: $h(m_1) = h(m'_1)$ and $h(m_2) = h(m'_2)$. Since $m \neq m'$, then either $m_1 \neq m'_1$ or $m_2 \neq m'_2$ (or both). But then whichever pair is unequal is a colliding pair for h.

In either case, we have found a colliding pair for h, contradicting the assumption that h was strong collision-free. ■

2.2 A General Chaining Method

Assume now that we have a hash function h : m-bits $\rightarrow$ t-bits, where $m \geq t + 2$. In the above example, $m = 256$ and $t = 128$. Divide the message m after appropriate padding into blocks $m_1 m_2 \ldots m_k$, each of length $m - t - 1$. Compute a sequence of t-bit states as follows:

$$\begin{aligned} s_1 &= h(0^t 0 m_1) \\ s_2 &= h(s_1 1 m_2) \\ &\vdots \\ s_k &= h(s_{k-1} 1 m_k). \end{aligned}$$

Then $H(m) = s_k$.

Theorem 2 *Let H and h be the functions of section 2.2. Then H is strong collision-free assuming that h is.*

Proof: Assume to the contrary that H is not strong collision-free, so we are able to find a colliding pair (m, m') for H. We show how to find a colliding pair for h, contradicting the assumed collision-freedom of h.

Let $m = m_1 m_2 \ldots m_k$, let $m' = m'_1 m'_2 \ldots m'_{k'}$, and let $s_1, \ldots, s_k$ and $s'_1, \ldots, s'_{k'}$ be the corresponding state sequences. We may assume without loss of generality that $k \leq k'$. Because m and m' collide under H, we have $s_k = s'_{k'}$. Let r be the least integer in $\{1, \ldots, k\}$ such that, for all $j \in \{r, \ldots, k\}$, we have $s_j = s'_{k'-k+j}$. Such an r exists since $r = k$ is one value that works. We proceed by cases:

Case 1: $r = 1$ and $k = k'$. Then $s_j = s'_j$ for all $j = 1, \ldots, k$. Because $m \neq m'$, there must be some ℓ such that $m_\ell \neq m'_\ell$. If $\ell = 1$, then $(0^t 0 m_1, 0^t 0 m'_1)$ is a colliding pair for h. If $\ell > 1$, then $(s_{\ell-1} 1 m_\ell,\ s'_{\ell-1} 1 m'_\ell)$ is a colliding pair for h.

Case 2: $r = 1$ and $k < k'$. Let $u = k' - k + r$. Then $s_1 = s'_u$. Since $u > 1$ we have that

$$h(0^t 0 m_1) = s_1 = s'_u = h(s'_{u-1} 1 m'_u),$$

so $(0^t 0 m_1,\ s'_{u-1} 1 m'_u)$ is a colliding pair for h. Note that this is true even if $0^t = s'_{u-1}$ and $m_1 = m'_u$, a possibility that we have not ruled out.

Case 3: $r > 1$. Then $u = k' - k + r > 1$. By the definition of r, we have $s_r = s'_u$, but $s_{r-1} \neq s'_{u-1}$ since r was chosen to be as small as possible. Hence,

$$h(s_{r-1} 1 m_r) = s_r = s'_u = h(s'_{u-1} 1 m'_u),$$

so $(s_{r-1} 1 m_r,\ s'_{u-1} 1 m'_u)$ is a colliding pair for h.

In each case, we have found a colliding pair for h. The contradicts the assumption that h is strong collision-free. Hence, H is also strong collision-free. ■

2.3 Hash Functions Do Not Always Look Random

Intuitively, we like to think of $h(y)$ as being "random-looking", with no obvious pattern. Indeed, it would seem that obvious patterns and structure in h would provide a means of finding collisions, violating the property of being strong-collision free. But this intuition is faulty, as I now show.

Suppose h is a strong collision-free hash function. Define $H(x) = 0 \cdot h(x)$. Clearly, H also enjoys these same properties. If (x_1, x_2) is a colliding pair for H, then it is also a colliding pair for h. Thus, H is strong collision-free, despite the fact that the string $H(x)$ always begins with 0. Later on, we will talk about how to make functions that truly do appear to be random (even though they are not).

3 Birthday Attack

Recall that the MD5 hash function produces 128-bit values, whereas SDA–1 produces 160-bit values. How many bits do we need for security? Both 2^{128} and 2^{160} are more than large enough to thwart a brute force attack that simply searches randomly for colliding pairs (m, m'). However, the so-called *Birthday Attack* reduces the size of the search space to roughly the square root of the original size. Thus, MD5 has roughly the same resistance to the birthday attack as a cryptosystem

with 64-bit keys would have to a brute force attack. Similarly, SHA–1's effective size in terms of birthday attack resistance is only 80-bits.[1]

The birthday attack is named for the *birthday paradox*. This is the fact that there is approximately a 50–50 chance that two people in a room of 23 strangers have the same birthday. There is a nice description of this on the web at http://en.wikipedia.org/wiki/Birthday_paradox. The probability of *not* having two people with the same birthday is is

$$q = \frac{365}{365} \cdot \frac{364}{365} \cdots \frac{343}{365} = 0.492703$$

Hence, the probability that (at least) two people have the same birthday is $1 - q = 0.507297$. This probability grows quite rapidly with the number of people in the room. For example, with 46 people, the probability that two share a birthday is 0.948253.

The birthday paradox can be applied to hash functions to yield a much faster way to find colliding pairs than choosing pairs at random. The idea is to choose a random set of k messages and then see if any two messages in the set collide. There are $\binom{k}{2} = k(k-1)/2$ different pairs of messages in a set of size k, so on can test this many pairs at a cost of only k evaluations of the hash function. Of course, these $\binom{k}{2}$ pairs are not uniformly distributed, so one needs a birthday-paradox style analysis of the probability that a colliding pair will be found. The general result is that the probability of success is at least one half for k roughly the size of $\sqrt{n}$, where n is the size of the message space.

Two problems remain that make this attack difficult to use in practice. First, there is the problem of finding duplicates in the list of hash values. That can be done in time $O(k \log k)$ by sorting the list and then looking for adjacent equal elements. The more serious problem with this approach, and with the birthday attack in general, is the amount of storage required. While carrying out 2^{64} computational steps is almost on the verge of feasibility, finding that much storage is still way out of the question, so MD5 and other 128-bit hash functions are still safe from this attack. Nevertheless, the birthday attack is one of the more subtle ways that cryptographic primitives can be compromised.

4 Hash from Cryptosystem

We've already seen several cryptographic hash functions as well as methods for making new hash functions from old. Here's a way to make a hash function from a symmetric cryptosystem with encryption function $E_k(b)$. Assume that the key length and block length are the same. Let m be an arbitrary length message. Pad it appropriately and divide it into block lengths appropriate for the cryptosystem. Compute the following state sequence:

$$\begin{aligned} s_0 &= \mathrm{IV} \\ s_1 &= f(s_0, m_1) \\ &\vdots \\ s_k &= f(s_{k-1}, m_k). \end{aligned}$$

The output $H(m)$ of the new hash function is s_k. IV is an initial vector and f is a function built from E. Some possibilities for f are

$$\begin{aligned} f_1(s, b) &= E_s(b) \oplus b \\ f_2(s, b) &= E_s(b) \oplus b \oplus s \\ f_3(s, b) &= E_s(b \oplus s) \oplus b \\ f_4(s, b) &= E_s(b \oplus s) \oplus b \oplus s \end{aligned}$$

[1] A recent attack reported by Chinese researchers Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu (mostly from Shandong University) have reduced this number to only 69-bits.

You should think about why these particular functions do or do not lead to a strong collision-free hash function. For example, if $k = 1$ and $f = f_1$, then $H_1(b) = E_{IV}(b) \oplus b$. E_{IV} itself is one-to-one (since it's an encryption function), but what can we say about $H_1(b)$? Indeed, if bad luck would have it that E_{IV} is the identity function, then $H_1(b) = 0$ for all b, and all pairs of message blocks collide!

5 Authentication Problem

The *authentication problem* is to identify who one is communicating with. For example, if Alice and Bob are communicating over a network, then Bob would like to know that he is talking to Alice and not to someone else on the network. Knowing the IP address or URL is not adequate since Mallory might be in control of intermediate routers and name servers.

As with signature schemes, we need some way to differentiate the real Alice from other users of the network. We generally do this by assuming that Alice possess some secret or password that is not known to anyone else. Then Alice authenticates herself by proving that she knows the secret password.

5.1 Passwords

Password mechanisms are widely used for authentication. In the usual form, Alice authenticates herself by sending her password to Bob. Bob checks that it matches Alice's password and grants access. This is the scheme that is used for local logins to a computer and is also used for remote authenticated telnet, ftp, rsh, and rlogin sessions. Such schemes have two major security weaknesses:

1. Except for local logins, the password is sent over the network in the clear. This exposes it to various kinds of eavesdropping, ranging from ethernet packet sniffers on the LAN to corrupt ISP's and routers along the way. The real threat of password capture in this way is so great that it is highly recommended that one *never* send a password over the internet in the clear. Users of the old insecure Unix tools should switch to secure replacements such as ssh, slogin, and scp, or kerberized versions of telnet and ftp.

   Logins into web sites often use the SSL (Secure Socket Layer) protocol to encrypt the connection, making it safe to transmit passwords to the site, but some do not. Depending on how you have it configured, your browser will warn you whenever you attempt to send unencrypted data back to the server.

2. Even if the password reaches the server safely, it is no longer the case that Alice is the only one who knows her password. Now the server also knows. This is no problem if the only use of the password is to authenticate Alice to *that particular* server, but what it means is that from then on, the server can impersonate Alice to any other service that uses the same password.

   Users these days may have accounts with dozens of different web sites. In order make the task of remembering the user names and passwords on all those sites, one is tempted to use the same user name-password pairs on all of them. But that means that anyone with access to the password database on one site could log into Alice's account on any of the other sites. Typically the different sites have very differing sensitivity of the data they protect. An online shopping site may only be protecting a customer's shopping cart, whereas a banking site allows access to a customer's bank account.

> My advice is to use a different password for each account. Of course, nobody can keep dozens of different passwords straight, so the downside of my suggestion is that the passwords must be written down and kept safe. If the primary paper on which they are written gets lost, then one should have a backup copy so that one can go to all of the sites ASAP and change the passwords (and learn if the site has been compromised).

The real problem with simple password schemes is that Alice is required to send her secrets to other parties in order to use them. We will see in the next lecture authentication schemes that do not require this.

5.2 Secure password storage

Another issue with traditional password authentication schemes is the necessity of storing the passwords on the server for later verification. The file in which the passwords are stored is obviously highly sensitive. While operating system protections can (and should) be used to protect it, they are not really sufficient. For one thing, legitimate sysadmins can access it and might conceivably use the passwords found there to log into users' accounts at other sites. Hackers who manage to break into the computer and obtain root privileges could also do the same thing. Finally, files get copied onto backup tapes that are not subject to the same system protections, so someone with access to a backup tape could read everybody's password from it.

Rather than store passwords in the clear, it is usual to store "encrypted" passwords, which really means the hash value of the password under some cryptographic hash function. The authentication function takes the cleartext password from the user, computes its hash value, and sees if that matches the hashed value in the password file. Since the password does not contain the actual password, and it is computationally difficult to invert a cryptographic hash function, knowledge of the hash value does not allow an attacker to easily find the password.

5.3 Dictionary attacks

Nevertheless, access to the password file, even if only hashed passwords are stored, opens up the possibility of a *dictionary attack*. The idea here is that many users choose weak passwords—words that appear in an English dictionary or in other available sources of text. If one has access to the password hashes of legitimate users on the computer (such as is contained in `/etc/passwd` on Unix), an attacker can hash every word in the dictionary and then look for matches with the password file entries. This attack is quite likely to succeed in compromising at least a few accounts on a typical system. Even one compromised account is enough to allow the hacker to log into the system as a legitimate user, from which other kinds of attacks are possible that cannot be carried out from the outside.

A way to make dictionary attacks more expensive is to add *salt* to each password. Salt is a random number that is attached to a user's account and stored along with the user name and hashed password in the password file. The hash function takes two arguments, the password and salt, and produces a hash value. Because the salt is stored (in the clear) in the password file, the user's password can be easily verified. However, a particular password hashes in different ways depending on the salt value. This means that a successful dictionary attack would have to encrypt the entire dictionary with every possible salt value (or at least with every salt value that appeared in the password file being attacked). This increases the cost of the attack by orders of magnitude.

6 Authentication While Preventing Impersonation

A fundamental problem with all of the password authentication schemes discussed so far is that Alice reveals her secret to Bob every time she authenticates herself. This is fine in an environment where she trusts Bob but not otherwise, for after authenticating herself once to Bob, then Bob can in turn masquerade as Alice to others.

When neither Alice nor Bob trust each other, there are two requirements that must be met:

1. Bob wants to make sure that an impostor cannot successfully masquerade as Alice.
2. Alice wants to make sure that her secret remains secure.

At first sight these seem contradictory, but there actually are ways for Alice to prove her identity to Bob without compromising her secret.

6.1 Challenge-response authentication protocols

In a challenge-response protocol, Bob presents Alice with a challenge that only the true Alice (someone knowing Alice's secret) can answer. Alice answers the challenge and sends her answer to Bob, who verifies that it is correct.

A challenge-response protocol can be built from a digital signature scheme (S_A, V_A) as shown in Figure 1. (The same protocol can also be implemented using a symmetric cryptosystem with shared key k.)

	Alice		Bob
1.		$\xleftarrow{r}$	Choose random string r.
2.	Compute $s = S_A(r)$	$\xrightarrow{s}$	Check $V_A(r, s)$.

Figure 1: Simple challenge-response protocol.

The problem with this protocol is that it exposes Alice's signature system to a chosen plaintext attack. With this protocol, a malicious Bob can get Alice to sign any message of his choosing. Among other things, this means that Alice had better have a different signing key for use with this protocol than she uses to sign contracts.

While we hope our cryptosystems are resistant to chosen plaintext attacks, such attacks are very powerful and are not easy to defend against. Anything we can do to limit exposure to such attacks can only improve the security of the system.

We now look at some ways that Alice might limit Bob's ability to carry out a chosen plaintext attack. In the protocol of Figure 2, instead of signing a string r of Bob's choice, Alice signs a string r that is constructed from a part r_1 chosen by Alice and a part r_2 chosen by Bob. The idea is that

	Alice		Bob
1.	Choose random string r_1	$\xrightarrow{r_1}$	
2.		$\xleftarrow{r_2}$	Choose random string r_2.
3.	Compute $r = r_1 \oplus r_2$		Compute $r = r_1 \oplus r_2$
4,	Compute $s = S_A(r)$	$\xrightarrow{s}$	Check $V_A(r, s)$.

Figure 2: Attempt to resist chosen plaintext attack: Alice goes first.

neither party be able to control r. Unfortunately, that idea does not work here because Bob gets r_1 before choosing r_2. Instead of choosing r_2 randomly, a cheating Bob can choose $r_2 = r \oplus r_1$, where r is the string that he wants Alice to sign as part of his chosen plaintext attack on her cryptosystem. Thus, the protocol of Figure 2 is no more secure against chosen plaintext attack than the simpler protocol of Figure 1.

Another possibility is to choose the random strings in the other order—Bob chooses first and then Alice—giving the protocol of Figure 3. Now Alice is the one who has complete control over r.

	Alice		Bob
1.		$\xleftarrow{r_2}$	Choose random string r_2.
2.	Choose random string r_1	$\xrightarrow{r_1}$	
3.	Compute $r = r_1 \oplus r_2$		Compute $r = r_1 \oplus r_2$
4,	Compute $s = S_A(r)$	$\xrightarrow{s}$	Check $V_A(r, s)$.

Figure 3: Attempt to resist chosen plaintext attack: Bob goes first.

This indeed thwarts Bob's chosen plaintext attack since r is completely random (i.e., all strings r are equally likely). No matter how Bob chooses r_2, Alice choice of a random string r_1 ensures that r is also random. Thus, Alice only signs random messages.

Unfortunately, the protocol of Figure 3 is totally insecure against active eavesdroppers. Suppose Mallory listens to a legitimate execution of the protocol between Alice and Bob. From this, he easily acquires a valid signed message (r_0, s_0). Now Mallory can impersonate Alice by choosing $r_1 = r_0 \oplus r_2$ in step 2 and $s = s_0$ in step 4. Bob computes $r = r_1 \oplus r_2 = r_0$ in step 3, so his verification in step 4 succeeds.

Both of these protocols can be improved by letting r be $r_1 \cdot r_2$ (concatenation) instead of $r_1 \oplus r_2$. That way, neither party has full control over r. This weakens Bob's ability to launch a chosen plaintext attack in the protocol of Figure 2, and it weakens Mallory's ability to impersonate Alice in the protocol of Figure 3. A still better idea might be to let $r = h(r_1 \cdot r_2)$, where h is a cryptographic hash function, since this further weakens the control that either party has on the choice of r.

7 Feige-Fiat-Shamir Authentication Protocol

In all of the challenge-response protocols above, Alice releases some partial information about her secret by producing signatures that Bob could not compute by himself. As we will see, the Feige-Fiat-Shamir protocol allows Alice to prove knowledge of her secret without revealing any information about the secret itself. Such protocols are called *zero knowledge*, which we will discuss in subsequent lectures.

The Feige-Fiat-Shamir protocol is based on the difficulty of computing square roots modulo composite numbers. Alice chooses $n = pq$, where p and q are distinct large primes. Next she picks a quadratic residue $v \in \mathrm{QR}_n$ (which she can easily do by choosing a random element u and letting $v = u^2 \bmod n$). Finally, she chooses s to be the smallest square root of $v^{-1} \pmod n$.[2] She can do this since she knows the factorization of n. She makes n and v public and keeps s private.

Alice authenticates herself by successfully completing a protocol that requires knowledge of s. We present a simplified version of the protocol in Figure 4. In a single round of the protocol, Bob has at least a 50% chance of catching an impostor Mallory. By repeating the protocol t times, the

[2]Note that if v is a quadratic residue, then so is $v^{-1} \pmod n$.

error probability (that is, the probability that Bob fails to catch Mallory) drops to $1/2^t$. This can be made acceptably low by choosing t to be large enough. For example, if $t = 20$, then Mallory has only one chance in a million of successfully impersonating Alice.

	Alice		Bob
1.	Choose random $r \in \mathbf{Z}_n$.		
	Compute $x = r^2 \bmod n$.	$\xrightarrow{x}$	
2.		$\xleftarrow{b}$	Choose random $b \in \{0, 1\}$.
3.	Compute $y = rs^b \bmod n$.	$\xrightarrow{y}$	Check $x = y^2 v^b \bmod n$.

Figure 4: One round of the simplified Feige-Fiat-Shamir authentication protocol.

To see that this works when both parties are honest, we just have to verify that

$$x = y^2 v^b \bmod n. \tag{1}$$

But this follows since

$$y^2 v^b \equiv (rs^b)^2 v^b \equiv r^2 (s^2 v)^b \equiv x(v^{-1} v)^b \equiv x \pmod{n}.$$

7.1 Cheating Alice

We now turn to the security properties of the protocol when Alice is dishonest, that is, when a party Mallory is attempting to impersonate Alice.

Theorem 3 *Consider one round of the Feige-Fiat-Shamir protocol of Figure 4. Suppose Mallory, who is attempting to impersonate Alice, doesn't know a square root of v^{-1}. Then Bob's verification will fail with probability at least 1/2.*

Proof: In order for Mallory to successfully fool Bob, he must come up with x in step 1 and y in step 3 satisfying (1). He does not know which value b Bob will choose when he is sends x in step 1. Let y_b be the string that Mallory sends to Bob in response to query b. We consider two cases.

Case 1: There is at least one $b \in \{0, 1\}$ for which y_b fails to satisfy (1). Since $b = 0$ and $b = 1$ each occur with probability 1/2, this means that Bob's verification will fail with probability at least 1/2, as desired.

Case 2: y_0 and y_1 both satisfy (1) (for their respective values of b), so we have

$$x = y_0^2 \bmod n$$

and

$$x = y_1^2 v \bmod n.$$

We can solve these equations for v^{-1} to get

$$v^{-1} \equiv y_1^2 y_0^{-2} \pmod{n}$$

But then $y_1 y_0^{-1} \bmod n$ is a square root of v^{-1}. Since Mallory was able to compute both y_1 and y_0, then he was also able to compute a square root of v^{-1}, contradicting the assumption that he doesn't "know" a square root of v^{-1}. ■

We remark that it *is* possible for Mallory to cheat with success probability 1/2. Here's what he does. He guesses the bit b that Bob will send him in step 2. He then generates a pair (x, y). If he guesses $b = 0$, then he chooses $x = r^2 \bmod n$ and $y = r \bmod n$, just as Alice would have. If he guesses $b = 1$, then he chooses y arbitrarily and $x = y^2 v \bmod n$. He then sends x in step 1 and y in step 3. The pair (x, y) passes Bob's check if Mallory's guess of b turns out to be correct, which will happen probability 1/2.

7.2 Cheating Bob

We now consider the case of a dishonest Mallory impersonating Bob. Alice would like assurance that if she follows the protocol, her secret is protected, regardless of what Bob does.

Consider what Mallory knows at the end of the protocol. If he sent $b = 0$ in step 2, then he ends up with a pair (x, y), where x is a random number and y is its square modulo n. Neither of these numbers depend in any way on Alice secret s, so it's intuitively obvious that this gives Mallory no direct information about s. It's also of no conceivable use to Mallory in trying to find s by other means, for he can compute such pairs by himself without involving Alice. If having such pairs allows him find a square root of v^{-1}, then he was already able to compute square roots, contrary to the assumption that finding square roots modulo n is difficult.

Instead, suppose Mallory sent $b = 1$ in step 2. Now he ends up with the pair (x, y), where $x = r^2 \bmod n$ and $y = rs \bmod n$. While y might seem to give information about s, observe that y itself is just a random element of $\mathbf{Z}_n$. This is because r is random, and the mapping $r \to rs \bmod n$ is one-to-one for all $s \in \mathbf{Z}_n^*$. Hence, as r ranges through all possible values, so does $rs \bmod n$. What does Mallory learn from x? Nothing that he could not have computed himself knowing y, for $x = y^2 v \bmod n$. So again, all he ends up with is a random number (y in this case) and a quadratic residue that he can compute knowing y.

In both cases, Mallory ends up with information that he could have computed without interacting with Alice. Hence, if he could have discovered Alice's secret by talking to Alice, then he could have also done so on his own, contradicting the hardness assumption for computing square roots. This is the sense in which Alice's protocol releases zero knowledge about her secret.