Problem Set 4 Solutions

Problem 4-1. Paillier Encryption

(a) Groups wrote implementations of Paillier in C, C++, Java, and Python. The shortest implementation was in Python; the second shortest was in C++ using the NTL large number library.

(b) Groups were awarded 5 points for a correct decryption and 5 points for a correct encryption.

Many students asked how the number to be decrypted was chosen. The number was simply a randomly-generated 10 digit number. We graded the problem by doing quick scan through the log file to verify that the number you reported was given to someone in your group.

The PINs were to prevent one group of students from sabotaging another student's results. Luckily that didn't happen, so we didn't need to use them.

(c) There are two security flaws with Ben's idea. The first is that a person could vote more than once. However, on the night before the problem set was due, we sent out email saying that this flaw would be handled by standard voter registration techniques.

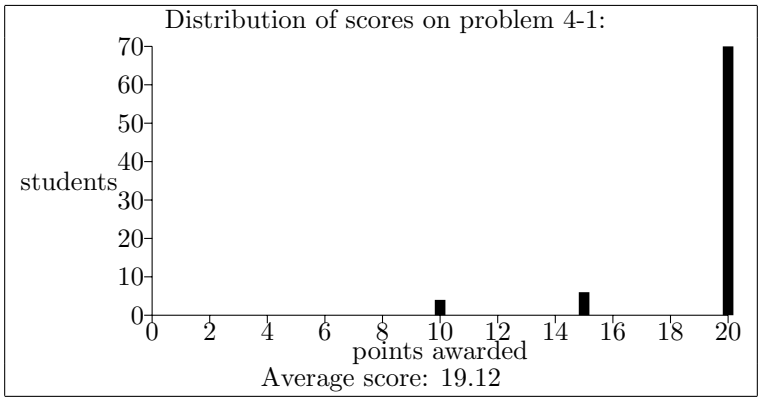
The second flaw — the one we were looking for — is that a person can stuff the ballot box with a single vote by submitting a ciphertext for an $m > 1$. In fact, a person could even take away votes by voting with a negative m, which would be an m that is somewhat less than n; that is, if Ben wanted to remove 10 votes, he could vote with $m = n - 10$.

A third flaw is that the scheme does not protect the votes of voters, since the agency is able to decrypt any individual voter's vote at any time. You need to trust the agency.

One group of students suggested an active attack: if you are in favor of the resolution, multiply each ciphertext by g, and if you are opposed multiply each ciphertext by g^{-1}. That's a lot of work; you could just multiply a single ciphertext by g^{500} to add 500 votes to the resolution.

It was incorrect to state that there were only two valid ciphertexts, allowing an attacker to create a dictionary of possible ciphertexts. That's the whole point of a randomized cryptosystem — involving the random value r in the calculation of each ciphertext prevents this kind of attack.

Grading policy: 5 points for working code, 5 points for a valid encryption, 5 points for a valid decryption, and 5 points for identifying a valid flaw.



Problem 4-2. MegaSoft Encryption (Courtesy of Subhasish Bhattacharya, Javed Samuel, Chi-Heng Wang, and David Wilson.)

There were a few common errors made on this problem:

- Arguing that $a = g^{2r} \pmod p$ has order q, because $a^q = 1 \pmod p$. This is true, but you also need to show that $a^x \neq 1 \pmod p$ for $1 \leq x < q$, otherwise a might have order smaller than q.

- Arguing that raising the ciphertext to the q power, and comparing the result to 1, was correct decryption *because the encryption is unambiguous.* This is a logical error: one could just as easily claim that raising the ciphertext to the $p-1$ power (which will *always* produce 1), and comparing the result to 1, is correct decryption! In order to make a correct argument, you need to prove that an encryption of a zero bit will produce 1, while an encryption of a one bit will not.
- Supplying a or b as the base to the discrete log-finding algorithm. The problem statement said that the base must be a generator mod p; a and b are not generators because their order is too small.
- Assuming that the discrete log of a (or b), base g, is always $2r$ (or $2q$, respectively). It's fine if your decryption algorithm assumes this (because you're in charge of choosing a when you generate a public key), but as an adversary you can only assume that a has order q, and your break must work for any such a.

The following solution was submitted by Subhasish Bhattacharya, Javed Samuel, Chi-Heng Wang, and David Wilson:

(a) One would find such a prime p by first finding large primes q and r and then testing if $p = 2qr + 1$ is prime. It is expected that there are sufficiently many primes of this form to make this algorithm reasonably efficient.

(b) Since for any $m \in \mathbb{Z}_p^*$, $m^{p-1} \equiv 1 \pmod{p}$, we know that if m has order t then t divides $p-1$. Specifically, in this case t divides $2qr$. Thus, the possible orders are the factors of $2qr$, namely 1, 2, q, r, $2q$, $2r$, qr, and $2qr$. Since there are $\phi(t)$ elements of order t, there are the following number of elements of the following orders:

Order	# elements
1	1
2	1
q	$q-1$
r	$r-1$
$2q$	$q-1$
$2r$	$r-1$
qr	$(q-1)(r-1)$
$2qr$	$(q-1)(r-1)$

To find a generator, one would randomly guess a candidate g and test that $g^x \not\equiv 1 \pmod{p}$ for all possible orders x (other than $2qr$). In fact, it is sufficient to test for $x = 2q, 2r, qr$, because all the other orders divide one of those three.

(c) Since g has order $p-1 = 2qr$, note that g^{2r} has order q (since $(g^{2r})^q \equiv 1 \pmod{p}$, but if $(g^{2r})^x \equiv 1 \pmod{p}$ and $x < q$, then g would have order less than $2qr$). Similarly, g^{2q} has order r. Now, since a and b are ideally random, let k_a and k_b each be random elements of $\mathbb{Z}_{p-1}^*$, and let $a = g^{2rk_a} \pmod{p}$ and $b = g^{2qk_b} \pmod{p}$. Since k_a and k_b each share no common factors with $2qr$, they do not affect the order of a and b in this case. Thus, a is a random element of order q and b is a random element of order r.

(d) If a particular ciphertext can arise in both ways, that means that there exist some $k_1 \in \mathbb{Z}_q^*$ and $k_2 \in \mathbb{Z}_r^*$ such that $a^{k_1} \equiv b^{k_2} \pmod{p}$. This would mean that

$$g^{2rk_ak_1} \equiv g^{2qk_bk_2} \pmod{p}$$

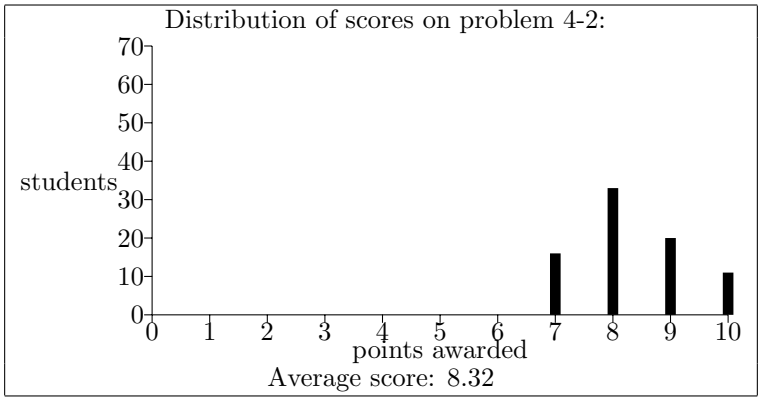
or, since g is a generator,

$$2rk_ak_1 \equiv 2qk_bk_2 \pmod{2qr}.$$

The left-hand side of the above equation is a multiple of r. However, notice that the right-hand side of this equation is not: since r is prime, it shares no factors with 2 or q, k_b was explicitly selected from $\mathbb{Z}_{2qr}^*$, and k_2 was explicitly selected from $\mathbb{Z}_r^*$. These two values cannot therefore differ by a multiple of $2qr$, so we have a contradiction. Thus, encryption is unambiguous.

(e) Since from (d) any encryption of a 0 is of the form g^{2rx} where $x \in \mathbb{Z}_q^*$, and any encryption of a 1 is of the form g^{2qy} where $y \in \mathbb{Z}_r^*$. Thus, the recipient has only to raise the ciphertext to the q power. If the plaintext is a 0, this becomes $g^{2qrx} \equiv 1 \pmod{p}$. If the plaintext is a 1, this becomes $g^{2q^2y} \pmod{p}$; since $y \in \mathbb{Z}_r^*$, this is not congruent to 1 $\pmod{p}$. Thus, the recipient can decrypt each bit with a single modular exponentiation.

(f)
1. Calculate $v = \mathcal{A}(p, g, b) = 2qk_b \pmod{2qr}$.
2. Use the Euclidean algorithm to find $\gcd(v,\ p-1) = \gcd(2qk_b + n * 2qr,\ 2qr) = 2q$ (since, once again, k_b shares no common factors with $2qr$, in particular r). Divide by two to obtain q.
3. Now that the adversary knows the value q, he can decrypt as described in (e), just like the intended recipient.



Problem 4-3. Sebek

(a) Courtesy of Alexandros Kyriakides, Saad Shakhshir, and Ioannis Tsoukalidis.

Everybody seemed to enjoy writing their answers to the Sebek2 problem and, for the most part, the answers were all very good. However, it takes time to write a good short abstract, and many groups obviously didn't take that time. Most of the essays were longer than we requested. Many omitted what we thought were key technical details from the paper.

For grading, all groups started with 10 points. Points were removed for any of the following:

- Significantly inaccurate technical assertions (up to -3)
- Failure to mention that Sebek crafts its own UDP packets (or simply that it sends packets directly to the ethernet interface) (-1)
- Failure to mention that Sebek is a client/server system (-1)
- Failure to mention the cleaner (or how Sebek hides) (-1)
- Failure to mention that Sebek2 works by intercepting `read()` system calls. (-1)
- Failure to mention how Sebek2 hides packets from other Sebek2-equipped systems on the LAN. (-1)
- Failure to discuss both pros and cons of publishing the Sebek2 paper. (-1)
- Failure to note that Sebek2 can be used by bad guys (or by corporations) for monitoring legitimate users (up to -2).
- Failure to note that publication of the Sebek2 paper can give bad guys information on how to avoid being caught by it. (-1)
- Poor writing (up to -3)

Common errors that were seen on the Sebek write-up:

- Many groups asserted that Sebek2 replaced the `read()` system call with its own version of this call. That's partially correct. Sebek2 actually intercepts the `read()` call with its own version, then calls the standard `read()`, then logs what `read()` returned, then returns that same data to the caller. (Notice that the logging happens *after* the original `read()` runs, not before. Many groups got the order reversed.)
- Many groups asserted that the reason that Sebek2 has access to unencrypted data is because it runs inside the kernel. This is not true. The network device driver also resides in the kernel and it does not have access to the unencrypted plaintext of data sent over an encrypted SSH connection.
  The reason that Sebek2 has access to unencrypted keystroke information and other data has to do with the decision on the part of the `ssh` designers to have encryption in one process and interpretation of commands in a second process, and to have those processes communicate across a Unix pipe. When you open up a connection to a remote computer, your `ssh` client communicates with an `sshd` server. The server then creates a pty (a kind of bidirectional pipe that supports terminal control), attaches to one side of the pty, and creates a shell that it attaches to the other side of the pty. That shell uses the unix `read()` system call to read commands typed by the user. So the reason that Sebek2 has access to unencrypted data is because the shell expects to read unencrypted data from `stdin`, so the `sshd` daemon needs to decrypt the data before sending it over the pipe. Likewise, data that the `sshd` daemon reads from the shell is decrypted; the `sshd` daemon then encrypts the data before sending it over the link. That's the other place that unencrypted data is captured and logged.
  If the `sshd` daemon was modified to execute user commands directly, rather than requiring the use of a sub-shell, then Sebek2 would not have access to unencrypted data, even though it would still reside in the kernel.
- Many groups did not properly explain that the magic number in the Sebek2 packet header is used to tell other Sebek2-compromised machines on the same LAN to discard Sebek2 monitor packets.

The following solution to part (a) was submitted by Alexandros Kyriakides, Saad Shakhshir, and Ioannis Tsoukalidis:

> The paper "Know your Enemy: Sebek2" describes a software system designed for the monitoring of honeypots. A honeypot is a host on the Internet, put online specifically for the purpose of being attacked. This is a useful security tool. The administrator of the honeypot can observe the attacker's actions, therefore gaining insight on what attacks are possible, and how attacks are performed. This helps to uncover security holes in systems.
>
> **Capturing Data:** One way of monitoring a honeypot is by passively listening to the honeypot's network traffic. A problem arises however, when the attacker uses encryption for his network connections. The designers of Sebek2 decided to circumvent this problem by capturing data after it is decrypted. Even if the attacker's traffic is encrypted in transit, at some stage it will be decrypted so that it can be used by an application. Once installed, Sebek2 changes the pointers in the kernel's System Call Table, so that system calls (that Sebek2 wishes to monitor) point to Sebek2 code and not the standard kernel code. The `syscall_read()` system call for example, is one that Sebek2 "intercepts". When a `syscall_read()` call is made, control is passed to Sebek2 code. Therefore, Sebek2 has access to all unencrypted data passed as arguments to the `syscall_read()`.
>
> **Saving Data:** Sebek2 uses honeypot clients and a Sebek server on the same local network. The honeypots send data to the server using UDP packets. The server reads these packets and stores them in a database for later use. Using this database, the administrator of Sebek2 can recreate all the attacker's actions on the honeypot, as well as observe the activity of malware programs that the attacker might have installed on the honeypot.
>
> **Hiding from the attacker:** It is important that the attacker does not realize that he is being monitored when he is attacking the honeypot. Sebek2 uses three methods for hiding

from the attacker.

1. The Sebek2 client is installed on the honeypot as a kernel module. Sebek2 also installs a *"cleaner"* program whose function is to hide the Sebek2 kernel module from the attacker.
2. Sebek2 installs its own packet generator. This allows Sebek to by-pass the TCP/IP stack so that the attacker cannot locally sniff Sebek packets (or even block them).
3. Sebek2 installs its own Raw Socket interface. Sebek2 packets contain a special signature so that they can be identified by other Sebek2 clients. When a Sebek2 client sees Sebek2 packets, it drops them. In this way, an attacker on Honeypot X cannot sniff Sebek2 traffic originating from honeypot Y.

(b) Courtesy of Dian Chen, Siddique Khan and Andrew Tsai.

Every group agreed that posting papers such as Sebek2 should be posted.
We saw some common errors in the essays:

- Many groups erroneously asserted that Sebek2 is a piece of protective security software, and posting the paper and Sebek2's source code allowed security professionals to analyze it for potential errors that an attacker could exploit. These groups then proceeded to write eloquent essays attacking the notion of "security through obscurity" and extolling on the virtues of Open Source Software.
  Why do you accept such arguments on their face without proof? Many Open Source programs have had considerable security bugs that have been undetected for *years*. Likewise, there are many closed-source programs (some of them classified) that are reported to provide security that is equal-to or greater than what is available in the Open Source community.
  You should know that the jury is out on whether Open Source or Closed Source is in general a better strategy for producing security software. What does seem to be the case is that other issues dominate the Open Source/Closed Source decision — issues like design methodologies, choice of programming language, development tools, and programmer experience. Simply waving the magic wand of Open Source and trying to excise the boogie-man of "security through obscurity" sounds great, but in practice things are not always so easy.
  Alexandros Kyriakides, Saad Shakhshir and Ioannis Tsoukalidis found a really interesting reference arguing that Security Through Obscurity isn't as bad as you think it is — it may even be useful. You will find it at `http://www.bastille-linux.org/jay/obscurity-revisited.html`.
- Many groups forgot that Sebek2 is designed for use with honeypots and that, in general, there are very few individuals and organizations actually running honeypots. Hackers don't know when they have broken into a honeypot system, and most computers are the Internet *are not honeypots.* For these reasons, the Honeynet Project could probably have decided to keep this technology secret, using it internally and not sharing it with others. It is not at all clear that the standard critique of "security through obscurity" applies to this case.
  The real issue here, then, is this: is the Honeynet Project better off describing its very best technology, with the hopes that this will cause more people to set up honeynets of their own, or are they better off holding their best technology in reserve, delaying the day that the bad guys find out about it?
- Some groups spent so much space arguing the Open Source/Closed Source issue that they forgot one very real danger of publishing this paper: that it will tell the bad guys that Sebek2 exists, and give them information they need to disable it. This is somewhat different from, say, Microsoft distributing the Windows Activation system in Windows XP and then hoping that people don't figure out how it works. With Sebek2, most attackers wouldn't have a computer on which they knew it was running, so it is reasonable to guess that it would have been somewhat more secure from reverse-engineering attacks.
- Many groups spent so much space arguing the whole "security-through-obscurity" issue that they forgot that the Sebek2 technology is inherently malicious technology that, when used properly, allows a system administer (or worse, an attacker who has compromised a machine) to spy on other users without being noticed. This technology has a significant potential for misuse!

None of the groups discussed the relative merits of publishing the Sebek2 paper online rather than trying to present it in a refereed forum such as a journal or conference.

Of all the essays written for part (b), your TAs found this one from Dian Chen, Siddique Khan and Andrew Tsai to be the most interesting...

The Honeynet Project's objectives are to raise awareness of computer vulnerabilities and to educate the computer security community on how to secure a system against hackers. By making the papers like Know Your Enemy: Sebek2 public, the Honeynet Project allows other security professionals to see the technology behind how they capture data on an attacker's methods, which does not interfere in any way with the aims of the project. Revealing how the Project obtains its information from the real-world (methods used by hackers) to educate the public is not important to its main goal of raising awareness and educating professionals.

The blackhat community makes many hacker tools and methods available to the public to educate and allow others in the community to examine and possibly extend existing methods. Because the hacker community is always advancing its techniques, computer security professionals must keep up and improve at the same pace. Because the Honeynet Project is a non-profit organization made up of volunteers, it does not have the resources to thoroughly research and test its systems (the Sebek2 paper itself is not very well-written with spelling and grammar mistakes). Publications are a good channel for the Project to test its developments out on the community and procure feedback. For example in the Sebek paper, the authors even asked for bug reports and suggestions for improving the system.

Like cryptography, which should never rely on security by obscurity, computer and network security should also make its methods public. Publishing research and new techniques allows for peer review so that systems can be better protected.

It is important to note that while we believe the Sebek2 paper is appropriate for the Honeynet Project to publish, it may not be a good idea for other organizations, companies, and government agencies to publicize their data capturing tools. For example, it would be completely unacceptable for an organization to advertise how they captured terrorists' hacker activities on a computer system, a move that could cut channels of intelligence and jeopardize national security. Therefore, the appropriateness of publications like the Sebek2 paper depends on the context and the purpose it serves.