Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ronald L. Rivest

Handout 23
October 22, 2002

Problem Set 4 Solutions

Problem 4-1. Uniformly selected primes and factorization of $p-1$

(a) Implement the random number generator

See `http://web.mit.edu/6.857/www/handouts/H22/` for the source code from the TAs.

The most common errors are the following:

- Forgetting to output r *only* with probability r/n.
- Counting only the numbers of calls to the primality testing function for the candidate prime p, forgetting to count the calls within Kalai's algorithm (the majority of the calls).
- Not having the proper bounds (by choosing n [the input to Kalai's algorithm] to be a random b-bit number instead of 2^b-2 for instance).
- Not having the proper output distribution by "optimizing" Kalai's algorithm: by discarding the case where $r=1$ (which does not matter much) or where $l=1$, by adding $s_{l+1}=2$, etc. Those optimizations do not speed up the algorithm much, but make it false.
- Using an improper threshold for the probabilistic primality test. For Miller-Rabin test, 10 iterations gives an error probability of $2^{-10} \approx 0.1\%$, which is satisfactory.

Also please follow the directions, and submit concise code!!! The longer your code is, the less likely the TAs will try to find your mistakes and give you partial credit if your code does not work. The actual Java code (counting only the algorithmic part) is no more than 50 lines of code or so.

And please reuse the same notations as in the algorithm: it makes it easier for you and the TAs to follow/debug the code.

(b) Compare to your co-Sophie-Germain prime generator

1) Kalai generator

According to Kalai's paper[1], the expected number of primality tests for Kalai's algorithm is $\Theta(\ln^2 N)$.

We are to set N to be 2^b-2, so that $\lg N \approx b$ and $\ln N = \ln 2 \cdot \lg N \approx \ln 2 \cdot b$. To get $p-1$ between $N/2 = 2^{b-1}$ and $N = 2^b-2$, we reject the output of Kalai's algorithm approximately half of the time.

The expected number of primality tests to get a b-bit *number* p with known factorization is thus $\Theta(\ln^2 N \cdot 2) = \Theta(2\ln^2 2 \cdot b^2) = \Theta(b^2)$.

Now the probability of p being prime is $\approx \frac{1}{\ln 2^b} = \frac{1}{\ln 2 \cdot \lg 2^b} = \frac{1}{\ln 2 \cdot b}$, so that the expected number of restarts is $\ln 2 \cdot b$, and the expected total number of primality tests to get a b-bit *prime* p with known factorization is $\Theta(b^2 \cdot \ln 2 \cdot b) = \boxed{\Theta(b^3)}$.

An actual asymptotic equivalent is $\boxed{2 \cdot (\ln^3 2) \cdot e^\gamma \cdot b^3 \approx 1.2 \cdot b^3}$, where $\gamma \approx 0.561$ is Euler's constant.

For $b = 512$, we get $\boxed{\approx 1.5 \cdot 10^7}$ (around 150 millions) primality tests.

[1] `http://web.mit.edu/akalai/factor/factor.html` or `http://web.mit.edu/akalai/www/PAPERS/factor.ps`

2) Sophie Germain generator

If we generate only odd numbers q, its probability of being prime is $\approx 2\frac{1}{\ln 2^{b-1}} = \frac{2}{\ln 2\cdot(b-1)} \approx \frac{2}{\ln 2\cdot b}$. As $p = 2q + 1$, its probability of being prime is $\approx 2\frac{1}{\ln 2^b} = \frac{2}{\ln 2\cdot b}$.

The probability of generating a valid (p, q) pair is thus $\approx (\frac{2}{\ln 2\cdot b})^2$, and the expected number of *trials* is $\approx (\frac{\ln 2}{2}b)^2$.

As we only test if p is prime when q is, the average number of primality tests is $1 + \frac{2}{\ln 2\cdot b} \approx 1$.

The expected total number of primality tests to get a b-bit Sophie Germain prime p is thus $\boxed{\approx (\frac{\ln 2}{2}b)^2}$. In Θ notation, this is $\boxed{\Theta(b^2)}$.

For $b = 512$, we get $\boxed{\approx 3.1 \cdot 10^4}$ (around 31 thousands) primality tests.

Note: For $b = 80$, the expected number of primality tests is $\approx 6.1 \cdot 10^5$ for the Kalai generator and $7.7 \cdot 10^2$ for the Sophie Germain generator.

(c) Generate 'em

The results submitted have been tested using the code which source is in `http://web.mit.edu/6.857/www/handouts/H22/`.

Using the TAs' code on a P4-1.8 GHz computer (the performance of the computer only affects the total running time), we get:

1) Kalai generator

```
Generating a 512-bit prime ...
Prime p: 12712465721168423836486185642789464410811025127024434288521101953391345\
         49219669643711401094721093310125244881617906182702266943651696202427685\
         841532481000981
Factors of p-1: 27842438909041253374851663628511157517402071215947601124552298 1\
                 61218633466572581514417996887266478851582463119037765384082801
                 1270573514578967 65173 10687 8599 5 3 2 2
Total number of primality tests: 139561906
Total time: 503492.507 seconds
```

The code took around 140 hours to complete (or 5.8 days).

The total number of primality tests (around 140 millions) is rather close to our estimate of 150 millions.

2) Sophie Germain generator

```
Generating a 512-bit prime ...
Prime p: 13333348233785115781455418662257197051824529646256620900693216176284 98\
         78814647681850264887659372824860532436488324070925963594151191775574 58\
         373289237541207
Factors of p-1: 66666741168925578907277093311285985259122648231283104503466080 8\
                 81424939407323840925132443829686412430266218244162035462981797 0\
                 75595887787291866446187706 03 2
Total number of primality tests: 25480
Total time: 247.556 seconds
```

The code took around 27 minutes to complete.

The total number of primality tests (around 25 thousands) is rather close to our estimate of 31 thousands.

Note: For a 80-bit size, the code took in average on an Athena station (on a sample of 100 trials):

1) Kalai generator

```
Average number of primality tests per run: 618202.7
Average time: 76.4395 seconds
```

2) Sophie Germain generator

```
Average number of primality tests per run: 3969.22
Average time: 0.55386 seconds
```

(d) The probability of generating a weak key

Transforming the condition ($\mathcal{C}$) on the q_i's in Dixon's theorem, we get:
$\frac{\lg n}{\lg q_i} \geq u \Leftrightarrow \frac{\lg n}{u} \geq \lg q_i \Leftrightarrow 2^{\lg n/u} \geq q_i$.

As $\lg n \approx |n| = 1024$, the condition becomes $q_i \leq 2^{|n|/u} = 2^{1024/u}$.

To find the probability that all the prime factors q_i of $p-1$ are $\leq 2^{160}$, we let $\boxed{u = 1024/160 = 6.4}$, and apply Dixon's theorem to get the probability $\boxed{u^{-u} = 6.4^{-6.4} \approx 6.9\ 10^{-6}}$.

Problem 4-2. How secure is your operating system?

Many students were surprised about what was actually running on their machines after reading `http://www.sans.org/top20/`. A few students maintain that they have iron-clad security. Most of the operating systems were Windows-based, but there were also sprinklings of MacOS X, *BSDs, and various Linux distributions. Some students were concerned about updating the Windows registry, fearing that it may do more harm than good — perhaps making the machine inoperable.

Several students explained that they never had time to update their computer or fix security holes. This was justified by students saying that their computers had nothing interesting to steal. On the contrary, most attackers don't care about your personal data, but want to use your resources to attack others or store pirated files. Lawmakers are debating whether those who leave open insecure computer resources should also be held legally liable. All things considered, most attackers break into the least secure machines. For better or worse, keeping your system more secure than another person's machine seems effective.

Here is a write up from Michael Kahan.

W1. IIS

First of all, I knew IIS on Microsoft was relatively insecure before reading this article, but even still, the number and severity of known insecurities was striking. After previously running IIS for a month or two, I disabled it this August because of the security risk it posed. Fortunately, my machine was not as vulnerable as it could have been. Previously, I had thought I was current with

the suggested patches (thanks to the Windows Update feature), but I was proven wrong as a result of doing this assignment (see W4). I could not find any sample applications the website suggested to look for (ism.dll, newdsn.exe,viewcode.asp, winmsdp.exe, etc.). Overall, my computer was fairly secure to these types of vulnerabilities. After reading this article, I will more seriously consider the security risks before restarting IIS in the future.

W2. (MDAC) – Remote Data Services

Running Windows XP and not having Visual Studio installed helps protect my machine from these types of vulnerabilities. My machine was not vulnerable from running IIS a few months back because I was using the MS Jet 4.0 engine and MDAC 2.1. Apparently, in one of the Windows Updates I had performed (I found the .reg file suggested by the fix), the update changed the following key to value 1, as recommended by the website (the update occurred before I started running IIS):

```
HKEY_LOCAL_MACHINE\Software\Microsoft\DataFactory\HandlerInfo\
Keyname: HandlerRequired
Value:   DWORD:1 (safe) or 0 (unsafe)
```

W3. Microsoft SQL Server

My machine was not vulnerable to these attacks because I do not currently have any version of MSSQL server installed.

W4. NETBIOS – Unprotected Windows Networking Shares

After a little analysis with the suggested tools, I found I was sharing some folders that allowed everyone full access to the folders' contents (I promptly stop sharing those folders). More important were my next findings, produced by the Microsoft Baseline Security Analyzer: contrary to what I had thought before, I was not up-to-date on security patches (I downloaded the two I was missing and installed them - they were for software I did not currently have installed, but had (not fully) removed from my computer some time ago). In addition, several IIS sample applications did exist on my computer (I tracked them down and deleted them). For the website I was running (and had since shut down), parent paths were enabled (I disabled them). As an added precaution, I installed and ran IIS lockdown, which removed some script mappings and installed a URL scanner, among other things.

W5 Anonymous Logon – Null Sessions

My machine was vulnerable to anonymous logons, but because I am running it in a domain environment and domain controllers need the null sessions to gain required information, all I could do was "limit the leakage" of information, as the website said. To do so, I edited the following registry key and set it to 1:

```
HKLM/System/CurrentControlSet/Control/LSA/RestrictAnonymous
```

W6. LAN Manager Authentication – Weak LM Hashing

As the paper says, all Windows machines are vulnerable to this defect. Because some legacy windows machines exist on my network, I could not change the domain controller registry key value to be too strict. Instead, I changed my computers'

```
Key: System\CurrentControlSet\Control\LSA
Value: LMCompatibilityLevel
```

to

```
(3=Send NTLMv2 authentication only).
```

In addition, I added the following new value to prevent the LM Hash from being stored locally on my machine:

```
Hive: HKEY_LOCAL_MACHINE
Key: System\CurrentControlSet\Control\Lsa
Value: NoLMHash
Type: REG_DWORD - Number
Data: 1
```

In the time I had to check the security of my system (3 hrs), that is all I could accomplish. This exercise opened my eyes to several of the major security holes that exist (some of which my computer was vulnerable to). Also, I learned that the Windows Update feature cannot be trusted to find all the necessary updates for a computer (as demonstrated by running the Baseline Security Analyzer). In fact, the Windows Update feature probably gives people a false sense of security. In addition, I confirmed my thoughts that IIS is too insecure to be run on a personal computer (or a computer with critical data), and I was surprised by the number/variety/severity of its security holes.