Collecting randomness from the net

November 23, 2000

Abstract: In this paper the problem of producing random binary blocks for security applications is considered. Random data is collected from network time delay measurements and its quality is checked by statistical tests. A special enhancement, a system of collector-servers is proposed and its security is analysed.

Key words: generation of random values, tests of randomness, good source of random data, private and authentic communication

Introduction: the idea

Good random values are very important ingredients for algorithms used in the field of data security. In the absence of sources that use physical randomness (such as electronic noise), random numbers of guaranteed quality are not readily available. Disregarding this aspect of security may prove dangerous, as these weakly random numbers may come from a much smaller space than expected, thus simplifying an attacker's task considerably.

This paper suggests a new method for producing high-quality randomness. It is based on the butterfly effect, as applied to a large number of observed components. The Internet features a large number of network elements (routers, servers, etc.), and experiences a continuously fluctuating number of requests for different services. As a simple consequence, the time delay of an answer to any query from a client to a remote server will depend on a huge number of effects, i.e. on the instantaneous state of the net. These time-delay measurements may plausibly be expected to carry useful randomness. The first part of the paper presents some typical results obtained from such measurements.

Even if we are able to generate randomness from measurements on the net, its use in security applications is not obvious at all. The reason behind this is an attacker's (theoretical) ability to silently track our measurements, e.g. the attacker is sniffing the client's LAN. Such an attacker might in principle be able to closely estimate the extracted random values. In the second part of the paper we propose a solution, a randomness-collector-system (RCS), for this problem. The RCS is a set of servers residing on different host machines spread over the net, accepting requests from clients for collection of random binary data. The collected random data has to be safely transmitted to the client, hidden from the prying eyes of the attacker who would like to record the transmitted data. An appropriate protocol is suggested. We also investigated the problem of a nearby attacker who tries to estimate the collected data.

E-commerce security applications require random data to create good keys for cryptography, but many systems don't have a reliable random number generator. Special

computers, like set-top-boxes, embedded web servers simply don't have a good source of data to feed a random number generator. Our goal was to provide a solution to these problems.

1. The random data

Measurement set up
We have carried out extensive measurements on network time delay. Samples have been collected from two 'sources': the round trip time (RTT), measured by the standard 'ping' program and page download time from web servers. These remote points will be referred to as remote-servers (or target hosts) hereafter.

The ping "ICMP ECHO Request" resolves time in microseconds (it uses the default C "struct timeval" structure, which contains a "long" number of microseconds). Therefore typical measurements could represent the response times with two bytes.

Our other randomness-collecting program was a simple script downloading a web page each second. A web page download is a much more complicated task than a simple Echo Request. Some webpages contain dynamic data that is created from a database. We expect the webpage download times to have more randomness than a simple ping, and making it harder to anticipate its value. The web page download time was also measured in microseconds by gettimeofdate() functions.

We have made extensive measurements on the response time of different remote-servers. (ping RTT and web page download times). Our programs saved the delay times to a database, enabling many different actions on the same data to be carried out.

A simple statistical model
A simple statistical model for the measured samples is the following:

$$x(t) = s(t) + n(t) \tag{1}$$

where x(t) is the observed sample, s(t) is a slowly changing mean value and n(t) a random fluctuation, and t stands for the time parameter.

The mean value of RTT is typically around 2-10 milliseconds within a campus, and around 40-500 milliseconds for oversea connections. The RTT mean value depends on the *Internet connectivity*, on the length of the path between the two computers and on the time of day. The random component is determined by the net effect of many different factors. The net is huge and complex with continuously changing unpredictable load caused by end systems. In the end, a huge number of simultaneously changing effects determine a parameter of the net that is observed in measured samples. This can be

modeled as a result obtained from a series of random variables. The first interesting question is the 'amount of randomness' in these variables.

In Table 1. below we present a typical collection of measured values, where the type of the remote-server and elementary statistics of measured response time values (maximum, minimum, average, standard deviation) are shown.

Server No.	Type	No. of samples	Maximum	Minimum	Average	Std Dev.
1	In-campus ping	110291	13661	3167	4316	1725
2	Oversea ping	199748	321779	173796	199432	20428
3	In-country ping	429242	46696	3402	6503	5371
4	Continental ping	427626	91281	8397	16550	13218
5	Continental ping	430713	102995	64391	69600	4450
6	Local webpage	417511	59000	1583	15603	7973
7	In-country webpage	158078	435584	21132	191896	58441
8	Continental ping(sniffed)	186716	46988	3535	7125	5409
9	Continental ping (sniffed)	188671	41998	10717	14391	4060

Table 1. Elementary statistics of measured response time values (data in microseconds)

The average is significantly larger than one standard deviation in all measurements. This implies that the magnitude of the slowly changing component is - accordingly - larger than the typical range of random fluctuation. The standard deviation is well within 2^{16} , suggesting that the randomness inherent in this set of population of samples will be well below 16 bits/sample, and a conservative guess would place it at 8 bits/sample at most. In order to obtain a more accurate estimate we have processed the samples in different robust, plausible ways and checked the result by statistical tests.

Processing of raw data

Our aim was to improve the randomness properties of these measured values by applying simple transformations (some of them can be found in [1]). The following transformations were applied:

For a start, the extreme values were expurgated from the set of samples. Let the result be denoted by x'(t). We examined different plausible rules for further processing x'(t):

Rule MA:
The slowly changing component s(t) has been estimated by the moving average (MA) computed over a sliding window of 20 samples. By subtracting this MA from the samples we obtain an estimate on the random component n(t).

Rule MA+L_8:
First we subtract the moving average from the data then with the aim of improving randomness; then the low order byte (L_8) of this estimate is taken.

Rule L_8:
We simply take the low order byte of x'(t).

Rule LXORH:
We make the XOR addition of the low order (positions 0-7) and the second low order bytes (positions 8-15).

Rule L_8 +par.:
The 8 bits of the low order byte are added by XOR addition to produce 1 bit random information for every input byte.

Rule L_8_U_2:
The upper 2 bits of the low order byte are extracted. It is elaborated upon below.

Statistical tests

For assessing the quality of the obtained random values, we applied several statistical tests. The aim of these tests is to measure the deviation of the tested random data from the ideal coin tossing sequence. The following tests have been applied:

1. Maurer's test
2. Poker test
3. Monobit test
4. Runs test
5. Long runs test
6. Chi-square goodness-of-fit-test

With Maurer's test [2] we used L=8 parameter due the low amount of available data. The expected value of the test for L=8 with a perfect random number generator is fTU=7.183665. Results from a good random source must not differ greatly from the expected value.

The Poker, Monobit, Runs and Longruns tests are part of the FIPS-140-1 [3] and FIPS-140-2 standards. We used an implementation by Neal Bridges[3]. The monobit test counts the number of "1"s in a 20000-bit long bit stream. The poker test uses 20000 bits of data and tests 5000 contiguous 4-bit segments for probability. Due to the 20000-bit block length, these tests were run on more blocks of input values.

The runs test calculates the run lengths (1..5 and 6+), these "runs" have a sequence of consecutive bits of either all ones or all zeros. If the statistical result is within predefined intervals, the test passes. The long run test checks if a "run" with more than 26 bit exists in 20000 bits.

Our Chi-square goodness-of-fit-test implementation used cells defined by consecutive 8 bits (1 byte). The test doesn't succeed if the probability of the chi-square value is below about 1 percent. From 1% to 99% probability we can't say anything.

The Shannon entropy has also been calculated for 8 and 16 bits per sample. We have used some other tests too, such as [4], with similar results.

A few characteristic results are shown in Table 2.

Server No.	Type of processing	Maurer's test	Poker	Monobit	Runs	Long Runs	Chi Square (Confidence level %)	Shannon Entropy (bit size of sample)
/dev/rnd	L_8	7.184011	Pass	Pass	Pass	Pass	290.74 (6.13)	7.999448 (8)
/dev/rnd	L_16	N/A					N/A	15.727350 (16)
1	L_16	5.479157	Fail	Fail	Fail	Pass	3406391 (0)	10.9458 (16)
1	MA	5.524571	Fail	Fail	Fail	Pass	1834372.37 (0)	11.6761 (16)
1	L_8	7.168220	Fail	Pass	Pass	Pass	461.17 (0)	7.9969 (8)
1	MA+L_8	7.173395	Pass	Pass	Pass	Pass	316.51 (0.52)	7.9979 (8)
1	L_8 +par.	7.174118	Pass	Pass	Pass	Pass	244.15 (32.25)	7.9872 (8)
5	L_16	5.960379	Fail	Fail	Fail	Pass	3469926.25 (0)	12.5285 (16)
5	MA	6.016704	Fail	Fail	Fail	Pass	3401189.50 (0)	12.6337 (16)
5	L_8	7.183341	Pass	Pass	Pass	Pass	259.87 (40.37)	7.9995 (8)
5	MA+L_8	7.183805	Pass	Pass	Pass	Pass	256.37 (46.41)	7.9995 (8)
5	L_8+par.	7.183577	Pass	Pass	Pass	Pass	244.28 (32.56)	7.9967 (8)
5	L XOR H	7.181106	Pass	Pass	Pass	Pass	286.22 (8.71)	7.9995 (8)
8	L XOR H	7.186706	Pass	Pass	Pass	Pass	230.92 (14.18)	7.9991 (8)
9	L XOR H	7.186481	Pass	Pass	Pass	Pass	261.22 (38.10)	7.9990 (8)
EGD_5	L_8	7.185633	Pass	Pass	Pass	Pass	224.46 (8.37)	7.9995 (8)
8	L_8_U_2	7.180367	Pass	Pass	Pass	Pass	257.22 (44.92)	7.9960 (8)

Table 2: Results of statistical tests

The first two rows of the table (/dev/rnd) contain data corresponding to the Linux standard /dev/random random number generator. It uses interrupt timing, block devices, keyboard and mouse movement to make good random numbers. The data is subject of entropy estimates and random refining tools (hash function). These rows are used for comparison purposes. The EGD_16 row is an output made by the "entropy gathering daemon"[5] program. We have modified the EGD program to use as a source of randomness 7.5bits/byte information (EGD parameter) from one of our previous tests (test no. 16 with lower 8 bits modifier). We checked the EGD's ability to produce as good statistics with our data feed as with other random sources.

Table 2 lends itself to several interesting observations:

The most important observation is that network latency times have enough extractable randomness. The raw data of delay times should be subjected to randomizing transformations. Relatively simple transformations can substantially improve the quality of the randomness. As it has been guessed above, 16 bits of raw data doesn't have 16 bits of usable random data.

Certain data we used is not long enough for a rigorous analysis. For chi-square and other tests we need a number of cells with data. If we have 256 different cells, we need at least 2560 bytes of data to make a minimal test. If we use 65536 different cells, then many more samples are needed. Some random number generator tests (like diehard) need megabytes of random data. If we don't use infinite random numbers for statistical tests, there is a chance of a bad statistical decision. The amount of data we used in the tests is limited to the speed of the data collection. It is clear, however, that the delay times do have a certain amount of extractable random data.

2. The randomness-collector-system

Default configuration: distributed collection

The system consists of a set of collector-servers (RC), which means N servers residing on different host machines spread over the net. This RCS accepts requests from clients to collect random binary data. The "Target Host" is the target of the delay measures (RTT). These measures are made by the randomness collectors (RCs).

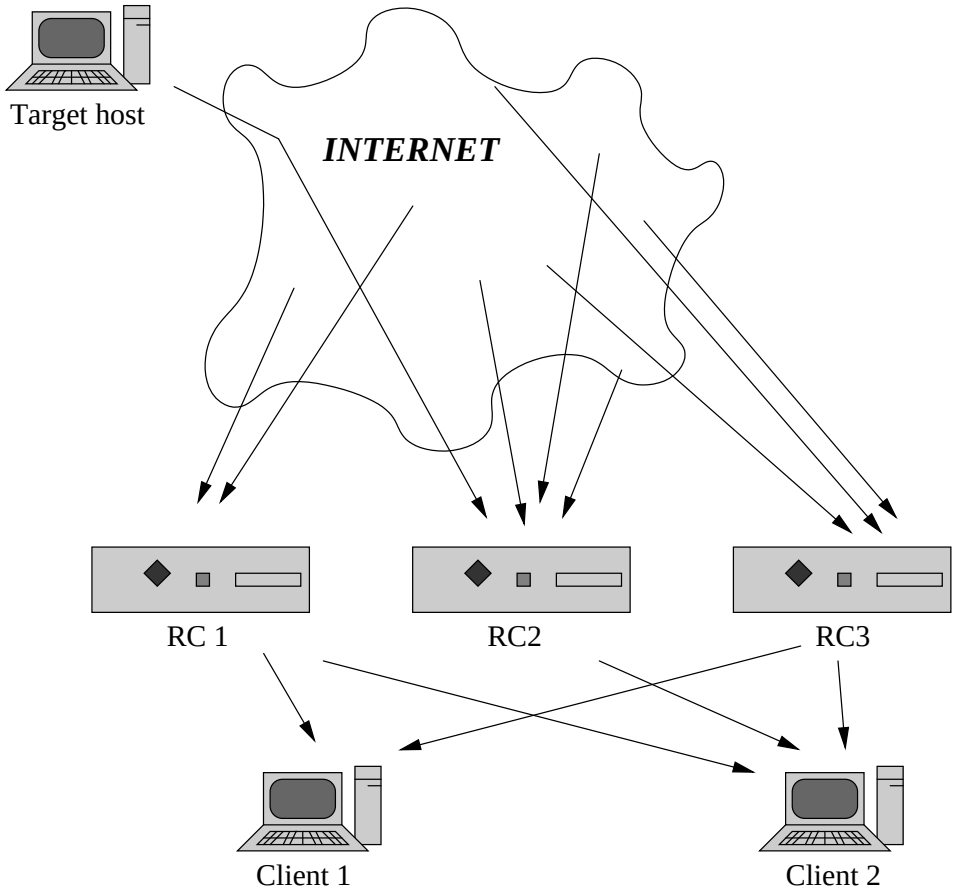

Fig. 1 The randomness-collector-system

The communication path between clients and collector-servers must provide confidentiality and authenticity. This means that data transmitted from the collector-server to the clients should only be readable by the client, and the client should also be able to check the authentic origin of the received data.

One possible implementation of these security requirements can be based on public key technology. All collector-servers and clients are assigned appropriate key pairs for performing public key encryption and decryption operations. (This implies that an initial 'randomness' exists in the system) As usual, confidentiality can be ensured by public key encryption. Authenticity can be ensured by performing the decryption operation using secret keys. Available cryptographic primitives permit different solutions to be designed [7]. For example the following two-step protocol meets the above security requirements:

$C \rightarrow S$: id, $k_{p,C}$, Cert($k_{p,C}$) (2)
$S \rightarrow C$: {M}$k_{p,C}$, {M',id}$k_{s,S}$

where
id: session identifier
$k_{p,C}$: public key of C
$k_{s,S}$: secret of S
Cert($k_{p,C}$): certificate for public key
{ }k: encryption in key k
M: data to send
M': fingerprint of data

A hybrid key solution could be based on Diffie-Hellman keys.

It is required that the hosts of collector-servers and clients are secure in the sense their HW/SW subsystem performing these security operations cannot be compromised. In any centralized security system the centers are sensitive parts.

After the client received N' random blocks, RC_i_rand, i=1,...,N', N'≤N, it combines them into one block. By default it calculates the XOR of all input blocks.

$$R = XOR_{i=1,N'}(RC_i_rand)$$

Access via proxies: centralization

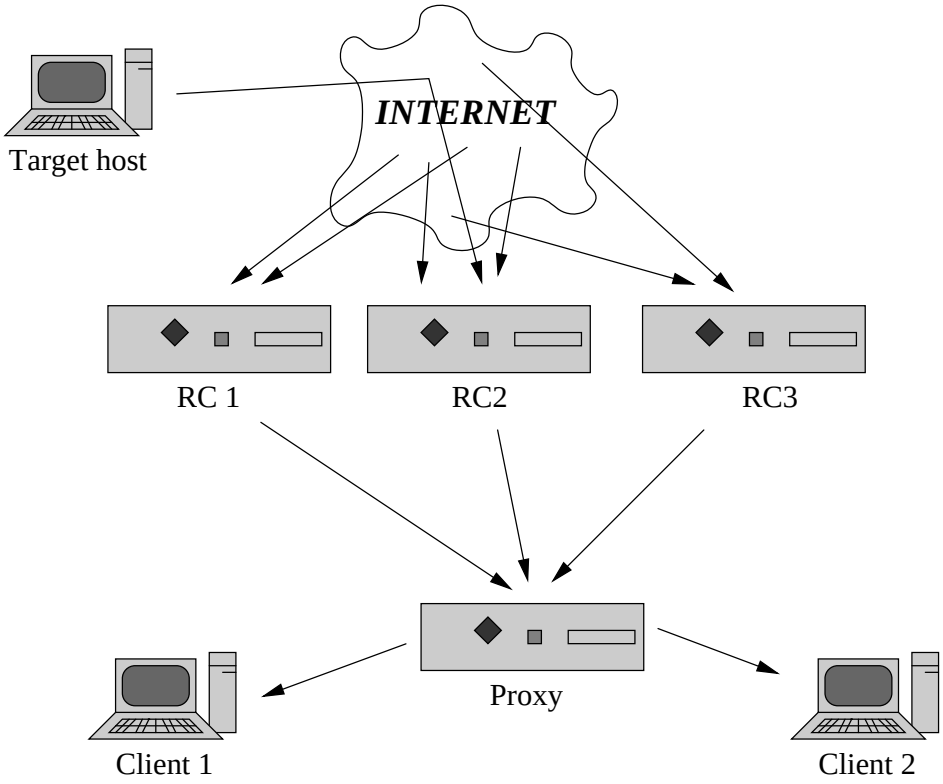

Fig. 2 Randomness collector system with central proxy

Clients send their request to a single network element, the proxy. This proxy collects and mixes the input data that has been collected by the RCs. After obtaining random data, a connection is built with clients and the data is transmitted for further use. Such a proxy can help resource-constrained clients to decrease the computational power and network bandwidth needed for secure communications.

3. Security analysis

Estimation of measured samples by an attacker

How accurately a physically nearby attacker could estimate the collected random samples is a vital question. As a worst case, assume that the attacker knows the actual remote connections (and also the time order of corresponding samples) and tries to replicate the procedure done by the client. We have made the following measurement regarding such an attacker:

We used one computer to collect data (pinging a host). Another computer on the same hub has used a packet sniffing software to detect the data sent by the original host. Let X and Y denote the associated random samples measured by the client and the

attacker respectively. We have calculated the conditional Shannon entropy H(X|Y), which measures the ambiguity about the actual value of X for the attacker, when Y is known to him. Consider the following model:

$$x(t) = h(t) + r_c(t)$$
$$a(t) = h(t) + r_a(t) \qquad (3)$$

where x(t) is observed by the client, a(t) is observed by the attacker, h(t) is the value at the hub, components $r_c(t)$ and $r_a(t)$ are produced by the HW/SW complex of the client's host and the attacker's host, respectively. From (3) it follows that

$$x(t) = a(t) + (r_c(t) - r_a(t)) \qquad (4)$$

The difference between data collected by the client and the attacker is the difference of time delays caused by the hosts. Therefore we calculated the conditional entropy

$$H(a(t) + r_c(t) - r_a(t) \mid a(t)) = H(r_c(t) - r_a(t))$$

where the equality is obtained from the independence of effects generating variable a(t) and variables $r_c(t)$ and $r_a(t)$. It is a plausible assumption that variables $r_c(t)$ and $r_a(t)$ are independent and identically distributed random variables, therefore the conditional entropy is between values $H(r_c(t))$ (= $H(r_a(t))$) and $2H(r_c(t))$. The value of conditional entropy is generated by two sources, while the value of variable x(t) is affected only by one of them.

We have measured H(X|Y)=5.3 (delay times to server no. 8). This means that a even close attacker, who is able to track the collecting activity of the client, experiences a fairly high ambiguity about the random values seen by the client.

The test with the L_8_U_2 modifier uses just the upper 2 bits of the low order byte. We expect that it doesn't contain randomness from the hardware ($r_c(t)$ and $r_a(t)$), only randomness from the network.

Protecting RCs from sniffing

It is assumed that an attacker is technically able to silently track our measurements launched from a given (known) host on the net. The RCs are collecting random data from different Internet connections. If anyone can sniff the Internet connectivity of an RC, the information in the time delay might compromised. We can conservatively assume that such kind of attack against a machine hosting a collector-server cannot be prevented.
However such an attacker must be physically (propagation time) close to the measuring machine. Therefore it is a plausible assumption that this capability of the attacker should exponentially decrease with the number of machines from which the measurement is simultaneously carried out. Consider the following simple statistical model. Assume that an attacker is able to track measurements from RCS independently and the probability that he can track a given machine is p, $0<p<1$. Consequently

P(an attacker can track j machines simultaneously)= B(p,N,j),

where j=1,2.,,,N and B is the binomial distribution with parameters p and N.. Here p is typically a low value, say p<1/30 could be considered a very conservative estimate (calculations based on [6]). The probability $P^*=p^N$ that the attacker is able to simultaneously attack all machines of RCS decreases exponentially with the value of parameter N (size of the RCS).

Hiding association between measured samples and clients
The servers in the system collect random data for lots of simultaneous requests. Because the servers own secret keys, they can distribute the random samples among requesters. If the system is centralized, then the center could perform this task of distribution.
It is easy to see that enhancing the system with such a properly working (i.e. preloaded with lots of requests) association hiding option substantially decreases the risk of attackers sniffing around the inputs of collectors.

Attacking communication path between clients and servers
When an attacker is interested in a specific client, it is plausible that instead of attacking lots of collectors spread over the net, he would try to get close to the client with the aim of observation and/or manipulation of random data arriving from the collector system. The data should be protected against the following attacks: eavesdropping, fabrication of bogus message and replay of old compromised messages. The needed cryptographic technology is available and well elaborated [7]. We can protect the data against eavesdropping by applying encryption. (Note that the used encryption technique can be unique because after decrypting the message we don't have to get the same random data as the original.) Using secret keys can prevent fabrication of a message. Insertion of nonce values (e.g. session identifier) into the messages can prevent the third type of attack.

Compromising the hosts of servers and clients
It is assumed that the servers and clients can protect their secret keys from an attacker. The whole RCS needs a proper initialization. If a part of the subnet is compromised, then the compromised keys should be changed and the system should be verified against backdoors as in the case of other Internet servers.

DoS attack against the servers
It is well known that the Internet is unfortunately highly vulnerable to DoS attacks. Our distributed collector system shows some structurally inherent robustness against DoS attacks. A centralized configuration is much more vulnerable.

The amount of randomness

Network delay is going to be smaller and smaller. If we have very fast Internet connections, there is less randomness in network time delays. In fact, network delays

won't go under a minimum, which is the speed of light. An Internet connection around the planet can be 20.000 km, which needs about 66 milliseconds. On the other hand, as the network links get faster, lines carry ever-increasing numbers of connections that add more randomness in network time delay variation. QoS may be a concern, as it enforces rules for network time delay variation, but not with infinite precision (and QoS is often only available on small subnetworks, not in general use).

Different platforms use different hardware and different hardware clocks, which have an effect on the collected data. On many systems, the clock does not resolve time to microseconds, etc. We are not saying that every operating system and every computer is fit to be an RC. We need RCs for clients that are unable to make good random numbers for themselves, so we can prescribe even microsecond capability on the RC hardware. To eliminate problems in the lower bits of the time delay, we use the bit mixing functions as described.

Limited usage of randomness

The random data is protected by some kind of cipher algorithm from the randomness collector to the client host. If the client gets N random bits, but the channel is protected with a cipher with a key size of M, then an exhausting attack for an application that uses the N random bits should run with a maximum of $2^{\min(N,M)}$ trials, assuming the presence of an eavesdropper on the network.
For this reason, the random data generated by the random network should not be used to make larger keys than the strength of the cipher that protects the submitted random data. Using mixing functions with initialization keys or combining the system with various other random source and other solutions could help to lower the chance of the random data getting compromised. With some modifications on the model (special use of random data source, selecting-mixing, etc.) we can enhance the effect of the encryption. With such modifications an attacker would have to try much more possibilities than $2^{\min(N,M)}$.

References

[1] Basic Statistics for Business and Economics, Hoel,P.G. – Jessen, R.J.,John Wiley & Sons Inc.,1982
[2] A universal statistical test for random number generators, Maurer, Ueli M., Journal of Cryptology,vol.5, no. 2., 1992, pp. 89-105.
[3] Security Requirements for Cryptographic Modules, Federal Information Processing Standards Publication 140-1, 1994, http://csrc.ncsl.nist.gov/fips
Implementation: Neal Bridges,2000.
[4] Ent, A Pseudorandom Number Sequence Test Program, John Walker, 1998, http://www.fourmilab.ch/
[5] EGD: an Entropy Gathering Daemon v0.8, Brian Warner
[6] An analysis of Security Incidents on the Internet 1989-1995, Dr. John D. Howard, 1997 http://www.cert.org/research/JHThesis/Chapter9.html
[7] Applied Cryptography, Part I, Cryptographic Protocols, Schneier,B., John Wiley & Sons Inc.,1996.