Alain H. SCHUMACHER

White Paper

AHS - Random Number Generation

&

RPP – OTP

“ Randomly Permuted Positions - One-Time Pad ”

Version 1.0
April 2006

AHS-RANDOM & RPP-OTP

Opening new doors in Cryptography

Table of contents

Alain H. Schumacher
6, rue de la Forêt Verte
L-7340 Heisdorf
Luxembourg

Tel : +352 44 27 42 (office)
Fax : +352 45 48 04

sicapas@pt.lu

The information in this document reflects in a fair way the actual state of our research.

Introduction

This white paper is addressed to the open-minded. It introduces the invention of a new method in the field of random number generation and its application in high-secure cryptography. As a private non-academic researcher I have chosen the platform of the International Exhibition of Inventions 2006 in Geneva to present my "baby" to the public. As a member of the AAAS for more than 10 years, I adhere to the principles of strict scientific correctness, so you do not have to fear any marketing arguments. In return I expect the reader to refrain from any polemics based on prejudices, and to be willing to enter the fascinating world of digital randomness.

In 1951 John von Neumann wrote his well-known statement: "Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."
Many people seem to confuse arithmetical methods and computer-arithmetic. Does anyone know an arithmetical method to win a chess competition? For sure, the answer will be no! Nevertheless, even the World-Champion already lost some matches against chess-playing computers, which were running without any doubt based on computer-arithmetics!

The goal of the development was to get a method which mimics the very basic principle of coin flipping. In our case that means to leave the arithmetical methods and to find a way to determine one or zero with a 50/50 chance for every bit. So the leitmotiv was: "Give chance a chance", in German: "Gib dem Zufall eine Chance".

For cryptography the result opens the door to different new solutions, and the quality of the randomness seems to be in the green area, if we evaluate the test-results from the generally admitted test-suites like the NIST. In the second field of potential use, the scientific simulations, only time will tell about the usefulness of the method, as the scientific community first has to come to an objective opinion. In the one simulation test we ran, a simulation of 10 series of 3.6 billion "birthday-paradox" with 23 persons, the AHS-Random generator performed well, compared to the MT19937.

About Randomness

If you are a collector and enjoy having something that nobody else on earth will possess, you may do the following :
Take a coin, flip it 256 times and note every outcome with “H“ for head and “T” for tail. In the end you will get a string similar to this one :

```
THTHTTTHTTHTTHHTHTTHHTHTHTTHHTHHHTHTTHHHTTTTHHHHHTHHTTHTTTHHHTTT
HTHTTTHHHHTTTTHTHHHTTTHTTTHTHHHTHHTHTHTHHTTTTTTHHHHHTHHTTHHTHTHT
THTTHHTTHHHHTTHHTHTHTHTTTHHHTHTTHHHTTTTTTHHHHHTHHTTHHTTTHTTHTTTH
HTHTHTTHHTTHTHTHHTTHHTTTTTTHHHHHTTTTHHTHHHTTTTHTTTHTTTTTTHTTHHTT
```

By replacing every H with a zero, and every T with a one, the string looks like this:

```
1010111011011001011001010110010001011000111100000100110111000111
0101110000111101000111011101000100101010011111100000100110010101
1011001100001100101010111000101100011111100000100110011101101110
0101011001101010011001111110000011110010001111011101111110110011
```

If you are familiar with computer-arithmetic you recognize that you have produced a bit-string of 256 bit length, and that we may convert this 256-bit number to a decimal number. In our example this will give the number

`79086541248287290464391135115232873378785628640428536280431363714368868695987`

This number is one out of all the possible numbers between 0 and 1.15792…*10$_{exp}$77. Considering that the total of atoms of the known universe, with an estimated 50 billion galaxies containing each around 100 billion stars, is in the region of 10$_{exp}$79, you will admit that the number you just produced by assembling 256 random bits is most likely a unique number that never showed up elsewhere in the world.
In cryptography the uniqueness of keys is an important requirement for unbreakable privacy.

The relatively easy way to get a unique 256-bit random number is due to the fact that all the sub elements, i.e. the bits, are independent from each other, as the coin has no memory to remember the outcome of the previous results, but any new flipping gives you a new 50/50 chance for head or tail (if you use an unbiased, so-called “fair” coin).
Unfortunately this is not true if you use your computer and the workhorse for pseudo-random number generation, the Linear Congruential Generator (LCG). This Pseudo-Random Number Generator (PRNG) uses a simple arithmetical function to calculate numbers that give an appearance of randomness, but are missing fundamental properties of true random numbers.

To illustrate the problem we may present a reduced model, i.e. a LCG running in an abnormal limited number space from 0 to 511. This will easily demonstrate the problem on a microscopic level in the same way as it exists on a normal level. The maximum of 511 plus one gives us a modulus (m) of 512. Let’s choose the value 97 as multiplier (a) and the value 13 as increment (c). With these parameters we will get the maximum of possible numbers (the period) of 512. The calculation to be done is the following:

Xnew = (Xold * a + c) modulus m

Modulus m means that we divide the result by 512 and take the remaining as random number. To start the LCG, we have to select a (random) value, named seed. Let’s choose 312 as seed, and store this value to the register named X. By producing the next 520 random numbers in the described way we get the following sequence (from left to right) :

69	50	255	172	313	166	243	32	45	282	231	404	289	398	219
264	21	2	207	124	265	118	195	496	509	234	183	356	241	350
171	216	485	466	159	76	217	70	147	448	461	186	135	308	193
302	123	168	437	418	111	28	169	22	99	400	413	138	87	260
145	254	75	120	389	370	63	492	121	486	51	352	365	90	39
212	97	206	27	72	341	322	15	444	73	438	3	304	317	42
503	164	49	158	491	24	293	274	479	396	25	390	467	256	269
506	455	116	1	110	443	488	245	226	431	348	489	342	419	208
221	458	407	68	465	62	395	440	197	178	383	300	441	294	371
160	173	410	359	20	417	14	347	392	149	130	335	252	393	246
323	112	125	362	311	484	369	478	299	344	101	82	287	204	345
198	275	64	77	314	263	436	321	430	251	296	53	34	239	156
297	150	227	16	29	266	215	388	273	382	203	248	5	498	191
108	249	102	179	480	493	218	167	340	225	334	155	200	469	450
143	60	201	54	131	432	445	170	119	292	177	286	107	152	421

402	95	12	153	6	83	384	397	122	71	244	129	238	59	104
373	354	47	476	105	470	35	336	349	74	23	196	81	190	11
56	325	306	511	428	57	422	499	288	301	26	487	148	33	142
475	8	277	258	463	380	9	374	451	240	253	490	439	100	497
94	427	472	229	210	415	332	473	326	403	192	205	442	391	52
449	46	379	424	181	162	367	284	425	278	355	144	157	394	343
4	401	510	331	376	133	114	319	236	377	230	307	96	109	346
295	468	353	462	283	328	85	66	271	188	329	182	259	48	61
298	247	420	305	414	235	280	37	18	223	140	281	134	211	0
13	250	199	372	257	366	187	232	501	482	175	92	233	86	163
464	477	202	151	324	209	318	139	184	453	434	127	44	185	38
115	416	429	154	103	276	161	270	91	136	405	386	79	508	137
502	67	368	381	106	55	228	113	222	43	88	357	338	31	460
89	454	19	320	333	58	7	180	65	174	507	40	309	290	495
412	41	406	483	272	285	10	471	132	17	126	459	504	261	242
447	364	505	358	435	224	237	474	423	84	481	78	411	456	213
194	399	316	457	310	387	176	189	426	375	36	433	30	363	408
165	146	351	268	409	262	339	128	141	378	327	500	385	494	315
360	117	98	303	220	361	214	291	80	93	330	279	452	337	446
267	**312**	69	50	255	172	313	166	243	32					

The 512^{th} value is our original seed, and from the next position on we replicate the first values produced. This is inevitable as the same calculation has to produce the same result on a computer. The biggest anomaly is the fact that every possible number between 0 and 511 appears one time. This is contrary to the characteristic of true random numbers.
If we do the same test with real random numbers, then approximately 36.75 % of the possible numbers will not appear, 36.82 % will appear once, 18.41 % will appear twice, 6.12 % will appear three times, 1.52 % four times, 0.30 % five times, and so on.

Let's do a small simulation: first we discard all the numbers from 365 to 511. If we now consider 0 to represent January 1^{st}, 1 to represent January 2^{nd}, and so on up to 364 representing December 31^{st}, we obtain a series of birthdays (to simplify we leave out February 29^{th}). Now we are able to simulate the question of the so called "birthdays-paradox": How many people have to be in a room for the probability to exceed 50% that at least two persons share the same birthday, assuming that the birthdays are equally distributed over the 365 days of the year?

The erroneous expression "paradox" is generally used because most people hardly believe that from 23 persons upwards it is more probable to have at least 2 persons with the same birthday than not to, even though it is only basic probability theory.
By simulating this problem with the produced pseudo-random numbers, a real paradox will appear. With all the possible seeds we shall use, we always find out that we need 366 persons in a room before we will have two sharing the same birthday!!

This test illustrates on a microscopic scale the problem that we encounter in bigger simulations with simple pseudo-random number generators like the well-known LCG. That is the reason why today more sophisticated PRNGs like the MT19937 are used for serious simulations. In the test-series you will find one test based on the simulation of this problem with different PRNGs and the AHS-RNG.

Infinite distribution

An infinite random sequence (by the definition of randomness with an infinite distribution) has, in a strict mathematical sense, to include all possible finite sequences. So we have to conclude that an infinite random sequence has to include the "Faust I" from Goethe. In our opinion this strict mathematical approach of infinity is very disturbing when dealing with real world random numbers. To explain this opinion we may calculate the following example:

Let's assume, just theoretically, that we are able to fill the whole known universe (let's take 46 billion light-years diameter for granted) completely with well-known quantum random generators, generating each 4 Megabit/sec of true random bits.
As these small units measure only 51 x 44 x 13 mm, we can store 1.4783 * 10exp93 units in the whole universe, if we don't take care of power generation and signal-lines. If we operate these quantum random generators for 1000 years, they will have produced together approximately 1.8674 * 10exp110 random bits. Coming back to "Faust I", we would find out that, with a lot of luck, all these bit-generators randomly found only the first 50 characters of Goethe' play.

For this reason we propose, when dealing with real world random number generation, to pragmatically define infinite randomness as the possibility to get, by producing the next 256 bits, any possible combination with equal probability, including the previous one produced. By logical extension we thereby also cover the theoretical infinite sequence. This definition will allow us to do empirical tests on reasonable subsets in the range of 10 to 100 Terabits. The confidence in a given random-number generator will rise if we share (or centralize) the results from identically defined tests. The probability theory gives us only an answer in form of the probability for different possible outcomes, and thus it is scientifically incorrect to judge a single result, even when the obtained result falls in the one to a million region of the probability. Only the collection of a multitude of results will give us a high confidence.

As we will see in the section on test-results, after running different tests on more than 350 Terabits, we have not yet found any indication against our presumption that AHS-RNG has an infinite random distribution. As explained, the sole fact that we have not yet found complete chapters of "Faust I" is not a proof against this presumption.

Infinite distribution and crypto applications

In cryptography you may want to have random numbers with specific properties. For example you may want 256-bit keys having between 110 and 144 "1"s and having at least 110 changes 0/1 and 1/0. Every good number generator will occasionally produce a substring of 30 or more zeros or ones. That's part of its job, and it cannot be blamed for it. It is the user's responsibility to check produced random numbers for these properties, and to simply discard the ones not fulfilling the requirements. But be aware that, by doing so, you reduce the number of possible combinations, and take care that the remaining variants satisfy your cryptographic needs.

Description of the AHS-Random Number Generator

The AHS-RNG is based on the principle that flipping a “fair” coin is a Bernoulli trial with a probability of exactly 50% to get a 0-bit or a 1-bit (corresponding to head or tail).
Let’s forget a few papers published in the past, putting in doubt the presumption that by physically flipping a coin the outcome will be exactly fifty/fifty. The decisive fact in coin flipping is the independence of the next outcome from the previous results.
By combining different sources of randomness the AHS-RNG mimics by software the principle of coin flipping. So it is correct to call it a “fair coin simulator”.

In order to get any of the theoretically possible combinations for a bit-string of a given length (including the last one produced) with the same probability, we have to abandon the arithmetical approach of the existing PRNGs. The basic “endless” possibilities in the AHS-RNG come mainly, but not exclusively, from a random table named Bit-Fishing-Table (BFT). The size of this table, normally a number of bits of an exponent of 2, may vary from as small as 8 Kbyte to a technical limit (for 32-bit processors) of 512 Megabyte. The table has to contain an equal number of one- and zero-bits to guarantee the same probability for the production of ones and zeros. The table has to be random for unpredictability. (See the term BFT in the glossary for more details)

To generate the random numbers, the AHS-generator processes bit-position by bit-position, with a 50 % probability to get a one or a zero. The first step per bit is to produce a random address (by combining different sources of randomness) in the range of the size of the BFT, and the second step is to take this bit from the random table and to add it to the random number under construction.

In the first instance, to run an engine, we need fuel. For this purpose the AHS-RNG uses the classical pseudo random number generator LCG, in a 64-bit version. Let’s clarify immediately that the output of the AHS-RNG is in no way correlated with the random numbers produced by the LCG. The main characteristic of this LCG is the fact that 2exp64 different and unique values of 64 bit show up in a random, but predictable order. These facts, against the principles of true randomness, are used advantageously in the AHS-RNG to guarantee the uniqueness of strings with a minimum length of 2exp64 bits per seed. Thereby we will be sure to get a production of at least, without any other possible interventions, 2exp128 bits per individual BFT (as we have 2exp64 possible seeds, and every seed will produce different random-strings of at least 2exp64 bits).

We strongly recommend to use the following trick in the seeding procedure: After the transmission of the seed-value to the seeding function, we add by program 64 bits from arbitrarily chosen positions of the BFT. Thus a possible attacker will not have the openly transmitted seed-value at his disposal.
The seeding procedure of the AHS-RNG is an important part of the AHS-RNG and needs a few hundred of the first pseudo-random values produced by the LCG. In the seeding procedure we have to distinguish between two different goals. The first goal is to calculate the values for the 16 basic modifiers (BM). These basic modifiers are calculated by combining in 8 registers some information from the LCG and the BFT, and in the other 8 registers the unmodified values from the LCG. By combining LCG and BFT we exclude the possibility to guess the values from these registers by knowing the seed, and by not modifying the other half we guarantee the uniqueness per seed. These values will stay unmodified until the next

re-seeding.
The second goal is to “fill the pipe”. This means that we calculate randomly, by extracting bits from the BFT with the help of the LCG pseudo random numbers, the starting values for different registers needed in the normal production cycle. This concerns the four 32-bit feed-back-modifiers (FBM), the four 32-bit basic-randomness-values (BRV) and the register with the last 32 bits produced.
As optional speed-optimization strategy in our test-implementation we filled an additional 32-bit register with a random value to be considered as chosen part of the BFT in the first cycle. In the first production cycle we use this one and we start the request for the next one to be used in the second cycle, always one cycle in advance. By doing so we partly avoid nasty delays in accessing the BFT in the memory, as the processor can do some work during the waiting period.

Once this seeding procedure completed, we can start “flipping the coin”. The basic cycle of the AHS-generator is the production of 4 bits. This is due to the method used for the calculation of the final address of the bit to be selected from the BFT. For speed optimizing purposes, we chained two basic cycles in our test-implementation to generate 8 bit numbers in one round.

The production of one bit goes as follows:

- we calculate the next LCG number
- we recalculate one BRV (cyclically one of the four) by XOR-ing the upper 32 bits from the LCG with one FBM (cyclically one out of the four)
- we transfer, by an “AND” instruction, selected bits, defined in a specific Final-Address-Assembling-Parameter (FAAP), from the BRV1 to the Final-Address (FA) register
- we add, based on the next FAAP, some bits from the BRV2
- we add, based on the next FAAP, some bits from the BRV3
- we add, based on the next FAAP, some bits from the BRV4
- after these operations we have the address of the bit to extract from the BFT, and we will add this bit to the random number under construction

After executing one, two, three or four times the production of 8 bits (for an 8 bit, 16 bit, 24 bit or 32 bit unsigned integer), we have to leave this main-cycle to update the table of the FBMs. We transfer FBM3 to FBM4, FBM2 to FBM3 and FBM1 to FBM2. The FBM1 is recalculated by XOR-ing the last 32 bits produced with a Basic Modifier (BM) determined in a cyclic way.

The next production cycle for 1, 2, 3 or 4 bytes may now start again. If high-speed production of larger quantities of random numbers is needed, we recommend of course to produce 32 bits at a time. On the other hand the reader will easily understand from these explications that changing from one request of 32 bits to two requests of 16 bits in a random way (based maybe on the clock-ticks or the elapsed time) allows us to produce non-reproducible random numbers, as we advance differently in the Basic-Modifier cycle, and as the last 32-bit register will not be the same. For this possibility we use the term run-time randomness.
As explained we will get 2exp64 different bit-strings (one per different seed of the LCG) of 2exp64 bits each. For different BFTs we will get of course completely different strings. One might now argue that this means to have only 2exp58 unique 64-bit integers per seeding for a

given table before we run out of the period of the LCG.
Effectively, after the first cycle of the LCG, there exists a small possibility that we may enter by chance in the same state of the FBMs. But we don't have to worry. First, it is easy to calculate that even with a production of 1 billion 64-bit integers per second we have to wait more than 9 years before this will happen. Secondly, in case we may encounter one day this problem, we can force an automatic new seeding after the production of 2exp64 bits. If we want to produce large amounts of random numbers by parallel-processing on a multiprocessor-system, we can chose to take a different BFT per processor, or to take one BFT and use a different seed per processor.

Other possibilities exist too, like the run-time randomness or the automatic changing of the FAAPs etc, so we think that it is not appropriate to try to calculate a periodicity as we would need to do if we worked with PRGNs. The AHS-RNG will offer for every speed the possibility to produce never seen random numbers (if we consider 256 bit-length) and never repeating, limited only by the basic laws of probability. We may get them as reproducible or as non-reproducible as we want them to be.

For truly non-reproducible random numbers for crypto applications we need a specially designed microcomputer which automatically increases the seed at every power-up, and has a secret BFT stored on the same chip in a secured memory, unreadable from outside. This represents the famous black box producing unpredictable random numbers which cannot be reproduced. If we store a random value as first seed together with the BFT, and the operating system sends the time at every boot to randomly increase the seed, even an attacker who managed to get hands on the BFT would not be able to find out the seeds used in the past.

Concerning the achievable speed, we have measured up to 124 Megabit per second (that is 15,5 Megabyte per second) with an 8 KB BFT and 115 Megabit per second with a 64 KB BFT, on an Intel Pentium 4 with EM64T at 3 GHz. On a small ARM 9 running at 180 MHz the rate obtained was 1,3 Megabit per second. The programming language is C-99, without hand-coded assembler optimization.

Due to the differences in the access speed between the caches and the main memory, the speed decreases rapidly if we use very large BFTs.

<u>The creation of the BFT</u>

To create the BFT we may use any satisfactory method. After generating the whole length of the table, we have to trim the table, in order to reach the same number of ones and zeros. First we count the number of ones, and calculate how many ones are missing, or if we have a surplus. We then choose, with the help of random numbers, random bit-positions. If the bit on this position is of the type with a surplus, we change the bit, otherwise we don't. We repeat this procedure until the number of ones and zeros is equal.
In our test-implementation we use a method to create the BFT without the help of an other RNG. The first generation is created with the LCG, and then we increase the generation several times, up to 100 or 200 generations, using the AHS-RNG. If we have a running AHS-RNG at our disposal we may as well use its output. For special cases a random generator based on physical quantum processes can be used.

The Pseudo-Random Number Generators (PRNG) versus AHS-RNG

In order to support our thesis that the AHS-RNG must not be considered as a pseudo-random number generator, we list the characteristics of PRNGs and compare them against the AHS-RNG. It is true that not every PRNG may have all these characteristics, but normally at least a few.

Infinite distribution

We may define, in a pragmatical approach, the infinite distribution as the possibility that the next 256 bit-string to be produced has the same probability to be any out of all the theoretically possible combinations, including the last one produced. None of the currently known PRNGs fulfills this condition, and this fact is generally admitted. This is the reason why PRNGs are considered to produce only finite sequences of random numbers. Even if it is hard to prove, our test-results and the conception of the AHS-RNG let us presume that it is most likely true that the AHS-RNG is able to produce infinitely distributed random numbers. Any possible proof falsifying this assumption is of course always welcome.

Binomial distribution

This distribution is very useful to check if the probability for smaller bit-strings, like 16 bits up to 80 bits, is in concordance with the probability theory for independent trials. To give an example, let's admit that you want to randomly distribute 100'000 times a dollar to 100'000 persons. If you always determine the person to get a dollar in a randomly and independent way, approximately 36'788 persons will stay with empty pockets, while a few ones will get six, seven or even eight bucks. This may appear very unfair, but such is live, and the laws of probability. The probability to get eight bucks is smaller than 1 to 100'000, so probably only in nine cases out of ten we will see one person getting eight bucks.
If you repeat this generosity infinite times, then of course one day there may be a case where one person will get the 100'000 bucks and 99'999 persons will stay with empty pockets. But by dealing with real world random numbers, the concept of infinity is very disturbing. Let's suppose in our example that you want to continue with your experiment until you get the case of one person receiving 25 bucks. The law of the binomial distribution tells us that with a very high probability your pockets (and bank accounts!) will be empty before you reach your goal, as the 50 % probability for this case is in the range of 200 billion billion trials.

In our tests the AHS-RNG performed well with trustable results, while the classical workhorse of the PRNGs, the LCG, was showing the known weakness, especially in the test of sorting 30 billion 64-bit strings. On the other hand the sophisticated PRNG named MT19937 performed well in this test.

Predictability, forward and backward

Except for specially designed pseudo-random bit generators for cryptographic applications, the PRNGs produce random numbers which are forward and backward predictable. As the random numbers are the result of a mathematical function, knowing one small sequence of the numbers allows you to calculate the previously produced number, as well as the next sequence to show up.
Due to the concept of the AHS-RNG, it is absolutely impossible to calculate, based on the knowledge of one part of the sequence, the unknown string before or behind the known part,

as long as the bit-fishing-table is secret.
If you know the BFT, but not the seed, you are not able to calculate the seed, but you would have to try out maybe all of the 2exp64 possible seeds to find the one used. If you know the seed and the BFT, but don't have the FAAP values, in case that we use randomly calculated FAAPs, then you would need to try out all the billions of billions of possible FAAPs.

The secrecy of the seed

The first strong recommendation for a limited use of some PRNGs in cryptography is to use only a seed based on an external random source, and to keep this seed secret. In case of the AHS-RNG nothing of the above applies. In cryptographic applications we may use the seeds 1, 2, 3 and so forth, and even show it to a potential malicious adversary. This means that we may send an e-mail to someone with whom we share a secret BFT, and are able to indicate in the subject line the seed used for encrypting the e-mail.

Periodicity

In principle, all of the PRNGs have a periodicity, after which they start to repeat the same random numbers. This is due to the mathematical function used in the PRNGs. For a classical 32-bit LCG we may prove this fact in practice in a few minutes on a modern desktop computer, while for a 48-bit or 64-bit version this task is more difficult, due to the much longer period. For the AHS-RNG we are not concerned with this problem. Every version of the BFT will guarantee a unique production of 2exp64 different strings of the length of 2exp64. As indicated in the description of the AHS-RNG, there exist different possibilities to exceed these values, e.g. run-time randomness and the change of the FAAPs.
The easiest way, if one day we might need to exceed the length of 2exp64 bits to produce, is the automatic new seeding with the old seed plus one. To illustrate the volume representing those 2exp128 random bits, we may calculate it in more common terms. Let's burn this volume of information on the new high-density DVDs with a supposed capacity of 50 Gigabytes per DVD. One DVD weighs 15.5 grams. After burning the DVDs, we stock them in railway wagons, 50 tons in a wagon of 10 meter length. In the end we would have produced some 1.318 * 10exp22 tons of DVDs, and they would fill a train of the length of 65'929 billion times the length of the equator. As we have this quantity of secret random numbers for every different BFT, it proves that periodicity is really not a problem for the AHS-RNG.

Deterministic function of the seed

"The outputs of a PRNG are typically deterministic functions of the seed; i.e., all true randomness is confined to seed generation. The deterministic nature of the process leads to the term "pseudorandom". Since each element of a pseudorandom sequence is reproducible from the seed, only the seed needs to be saved if reproduction or validation of the pseudorandom sequence is required" (point 1.1.4. paragraph 2 of the NIST Special Publication 800-22 A statistical test suite for random and pseudorandom number generators for cryptographic applications).

In the case of the AHS-RNG the seed plays a role, but not the major one, and absolutely not the only one. The main source of randomness is the bit-fishing-table (BFT). The seed, the FAAPs and the possible outside randomness introduced during the run of the generator by the run-time randomness are supplementary sources of randomness.

The deterministic nature of the process

In the literature one finds the opinion that computers, as deterministically working machines, are not be able to produce “true” random numbers, and thus always produce pseudo-random numbers. At the same time these authors support their statement by indicating the characteristics of the known PRNGs. It seems that historically the prefix “pseudo” was not introduced in order to distinguish between computer generated random numbers and those generated by means of other deterministic physical processes, but in order to indicate that the random numbers generated by known PRNGs like the LCG are missing some of the characteristics of true random numbers.
By the way, anyone who is familiar with the generation of random numbers by a physical process knows that there exists no physical process for simulating a “fair” coin with a statistically correct distribution of ones and zeros. There are some algorithms to correct this with mathematical functions, executed normally by means of a computer program. Therefore, one might wonder if it is correct to consider these numbers as “true” random numbers, as their final values are normally calculated by a deterministically working computer, using one out of more possible mathematical functions. Depending on the algorithm used, the same original sequence will result in different “true” random numbers.

As special case we may consider the random numbers generated by a physical quantum process. Who will disagree with the statement that for example flipping a coin is basically a physically deterministic process? But concerning the question of quantum processes, at this moment a large majority backs the thesis that the quantum effects are of “true” randomness. But regardless whether this thesis will stand for ever, random numbers produced this way always need some post-treatment to become useful for practical purposes.

We think that the real question is a more philosophical one. We have to decide if randomness is only a question of momentary events inter-depending in such a way that the outcome may not be calculated in advance, but may only be calculated or estimated with a certain probability. Or may we conclude that randomness is a product of historical and momentary events? In the case of the optical quantum generator, the historical process of manufacturing the product certainly plays a role, and as its output is declared true random, we cannot refute this second interpretation.

Applied to the AHS-RNG, we may conclude that the historical event of choosing a given BFT and the momentary event to choose a specific seed and/or FAAP, together form the randomness, and that the deterministic calculation of the computer is only the transformation of this intrinsic randomness, and doesn't influence the randomness in any way.

Just to remind: with an 8 KB BFT we have more than 10exp19'725 different possibilities, and with a 64 KB BFT there are more than 10exp157'823. The momentary event in form of seed and FAAP offers a supplementary randomness of 10exp39 possibilities. For all who think that this is not enough randomness, the possibility exists to add run-time randomness during the run of the generator (see run-time randomness in the glossary). Therefore we hope that you now understand why we refuse the prefix "pseudo" based on the sole fact that computers are deterministic machines.

TEST RESULTS / AHS-Random

First-bit and the number of repetitions

Let's begin with the starting bit of AHS-Random sequences. As we have a 50/50 probability for ones and zeros, the first bit has to be a zero in approximately half of the cases. But for the next bit, the same rule applies. As a result we will have only 25 % of a single zero followed by a one, and 25 % of a single zero followed by a zero. In those 25 % with the same bit on the first two places, again one half will have a third identical bit while the other half will have a different bit in third place. So the arithmetical series is: 25 % only a single identical bit, 12,5 % two, 6,25 % three, 3,125 % four etc. With the number of test-cases (i.e. different seedings) the probability will increase to get a long string of identical bits in the beginning. If someone tells you to discard a RNG because you found a sequence beginning with 35 zeros, don't trust him, he has not yet realized the spirit of randomness. First do a check with a big number of starts, and only if you find an obvious irregularity in the result you have to follow his recommendation. We have done 5 series with different 64 KB BFTs and always 200 billion seedings with seeds from 0 to 199'999'999'999. In appendix **B** you find the results of these 1000 billions cases.

Three long sequences of 100 Terabit each

We have generated three different 100'000 billion bit long sequences with BFTs of 8 KB, 16 KB and 64 KB, the whole sequence with one seeding. We counted the "1" bits per 1000 bits, per 1 million and per 1 billion. The results given are each time for 8 KB BFT / 16 KB / 64 KB.

The total "1"s for the whole sequences are 50'000'000'516'497 / 50'000'003'288'661 / 49'999'999'516'586. This seems ok as the standard deviation for this case is 5'000'000. With only three results we may not yet decide that the randomness is too low.
The ratio for the "fair" coin flipping is : surplus of one "1" per 193'611'966 bits / surplus of one "1" per 30'407'512 bits / one missing "1" per 206'862'027 bits.

The counting of "1"s per 1000 bits, three times 100 billion results, was the following:
Ratio between < 500 / > 500 (exactly 500 discarded) 1.00000184 / 0.99999355 / 1.00000233
Std.dev. total: 15.811375 / 15.811425 / 15.811367 (probability : 15.811388)
only left : 15.811342 / 15.811421 / 15.811373 "
only right: 15.811408 / 15.811429 / 15.811360 "

As the number of available results is very high, we checked, in addition to the standard deviation, the exact probability for every possible number of "1"s. The total of the surpluses resp. the missing ones, compared to the theoretical binomial distribution, was:

1'083'041 / 1'170'815 / 1'065'474 per 100'000'000'000
resulting in a percentage of 0.001083 % / 0.001171 % / 0.001065 %

The lowest encountered number of "1"s : 393 / 396 / 396
The highest encountered number of "1"s : 607 / 608 / 609

The counting of "1"s per million bits, three times 100 million results, was the following:
Ratio between < 500'000 / > 500'000 (exactly 500'000 discarded) :

1.00000536 / 0.99973038 / 0.99982747

Std.dev. total:	499.988568 / 500.011027 / 500.028362	(probability: 500)
only left:	499.975055 / 500.042811 / 500.096373	"
only right:	500.002080 / 499.979250 / 499.960354	"

The lowest encountered number of "1"s : 497280 / 497177 / 496935
The highest encountered number of "1"s : 502830 / 503060 / 502830

More than 99.99 % of the cases are in the range from:
498056 up to 501942 / 498052 up to 501943 / 498055 up to 501946

From the counting of "1"s per billion bits we got three times 100'000 values. If we claim infinite randomness, the variance, and so the standard deviation, has to follow the general law even at this sample-size. The results are the following :

Std.dev. total:	15820.177 / 15820.831 / 15863.822	(probability: 15811.388)
only left:	15788.772 / 15779.193 / 15861.905	"
only right:	15852.047 / 15862.733 / 15865.907	"

The maximum of the difference is 1/3 percent. It would be very interesting to have results from identical test-sequences from physical "true-random" generators.

Below 500'000'000 : 50041 / 50030 / 50126 samples
Above 500'000'000 : 49956 / 49968 / 49873 samples

Ten series with 240 Gigabyte AHS-Random 64 KB BFT compared to the MT19937

We generated ten times a sequence of 240 Gigabyte, 1920 Gigabits with the AHS-Random generator (different BFTs from 64 KB) and ten times 240 GB with the MT19937 with different seeds. In the results you find left side AHS / right side MT19937.
The results indicated concern the total of the 10 x 240 GB, so 2.4 Terabytes or 19.2 Terabits.

The total "1"s for the whole sequences are: 9'599'999'162'167 / 9'600'001'173'627
This seems ok as the standard deviation for this case is 2'190'890.23.
The ratio for the "fair" coin flipping is : one missing "1" per 22'916'261 bits / surplus of one "1" per 16'359'541 bits.

The counting of "1"s per 1000 bits, two times 19.2 billion results, was the following:
Ratio between < 500 / >500 (exactly 500 discarded) 1.00000958 / 0.99999277

Std.dev. total:	15.811473 / 15.811306	(probability : 15.811388)
Only left :	15.811459 / 15.811282	"
only right:	15.811486 / 15.811330	"

As the number of results available is very high, we checked, in addition to the standard deviation, the exact probability for every possible number of "1"s.
The total of the surpluses resp. the missing ones, compared to the theoretical binomial distribution was:

501'654 / 481'026 per 19'200'000'000
resulting in a percentage of 0.002613 % / 0.002505 %

The counting of "1"s per million bits, two times 19.2 million results, was the following:

Ratio between < 500'000 / > 500'000 (exactly 500'000 discarded) :
1.00022457 / 1.00037307

Std.dev. total:	500.109484 / 500.160676	(probability: 500)
only left:	500.153327 / 499.993487	"
only right:	500.065627 / 500.327871	"

The lowest encountered number of "1"s : 497307 / 497208
The highest encountered number of "1"s : 502754 / 502749

More than 99.99 % of the cases are in the range from:
498056 up to 501946 / 498054 up to 501952

From the counting of "1"s per billion bits we got two times 19'200 values. If we claim infinite randomness, the variance, and so the standard deviation, has to follow the general law even at this sample-size. The results are the following :

Std.dev. total:	15805.438 / 15890.881	(probability: 15811.388)
only left:	15780.538 / 15834.707	"
only right:	15831.251 / 15947.123	"

Below 500'000'000 : 9624 / 9552 samples
Above 500'000'000 : 9575 / 9647 samples

In this test we counted the distribution per byte-value (8 bit unsigned integer).
The average value par byte: 127.50001600 / 127.50007013 (Theor. 127.50)
The standard deviation on the numbers per value:
97'361.260 / 92'790.131 (Theor. 96'635.288)

We counted also the number of strings of identical bits per length.
The longest string for the AHS was 44 x "1" (probability 0.272)
and was 47 x "0" for the MT19937 (probability 0.034).

The difference theoretical / counted for all cases was the following:
"0" abs. 4889103 / 3545984 in percent 0.00010186 % / 0.00007387 %
"1" abs. 2021919 / 2763252 in percent 0.00004212 % / 0.00005757 %

As the number of changes from "0" to "1" and from "1" to "0" is linked arithmetically to the previous values, it may not surprise that they are very close to the theoretical value of 50 %.
In absolute figures: 9'600'001'606'248 / 9'600'000'517'286
The standard deviation is again 2'190'890.23, as the probability for a change is the same as the probability for a "1" or "0".

The diversity of the first 256 bits

In order to check if the first 256 bits produced are really random if using the same BFT continuously seeded (new seed = old seed + 1), and for different BFTs seeded with always the

same seed, we have done the following two tests:
<u>5'000'000 keys of 256 bit, same BFT, seeds from 0 to 4999999</u>

The test consist of a systematic check of every key of 256 bit against all the others, and to find out the number of diverging bits. Once we have tested A against B, there's no need to also check B against A, as the differences are the same. So we got a total of 12'499'997'500'000 tested pairs. The results from this test:
For random keys the average result should be 128. Counted: 127.999999974
Ratio between less than 128 bit difference and more than 128 bit difference : 1.000000148
Total std. dev. (Theoretical 8) : 8.00000103
Only left side: 8.00000044
Only right side: 8.00000161
Lowest number of different bits : 69 Highest : 186
More than 99.9999 percent of the cases are in the range from 89 up to 167.

<u>5'000'000 keys of 256 bits, different BFTs, seed always 0</u>

The test consists of a systematic check of every key of 256 bit against all the others, and to find out the number of diverging bits. Once we have tested A against B, there's no need to also check B against A, as the differences are the same. So we got a total of 12'499'997'500'000 tested pairs. The results from this test:
For random keys the average result should be 128. Counted: 128.000000955
Ratio between less than 128 bit difference and more than 128 bit difference : 0.999999928
Total std. dev. (Theoretical 8) : 8.00000217
Only left side: 8.00000147
Only right side: 8.00000288
Lowest number of different bits : 71 Highest : 187
More than 99.9999 percent of the cases are in the range from 89 up to 167.

Sorting ten samples of 30 billion 8-Byte strings (64 bits)

The previous test was designed to check if the number of different bits follows the theoretic probability. The number of test-pairs, in the region of 10exp13, and the length of the tested keys was nearly excluding an identical pair. Based on the binomial law we are able to calculate e.g. the probable number of identical values in a big set of small random bit-strings. In 30 billion 64-bit random samples the theoretical probability is 24.39454884. But comparing a set of 30 billion strings each one against all the others is definitely an impossible task if you don't have a grid of one million computers at your disposal. The number of pairs to check is 4.5 * 10exp20.

We can solve this problem by first sorting the 30 billion strings and then checking how many identical strings we find, as same strings will show up as neighbors in the sorted set.
During the sort-process we are also able to determine the number of strings identical on the last 16 bits, the last 20, 24, ... up to the whole string of 64 bits.
An infinitely distributed random sample will follow very closely the theoretic probability.
In appendix **C** you find the sum of the values for the ten tests with 30 billion strings, up from the last 32 bit to 64 bit (the tables from 16 bit to 28 bit are too long!).
It is very interesting how close the number of identical 64 bit strings is, compared to the theoretic value. We think that this is only by pure coincidence, as the details for the different

tests are: 17, 35, 33, 24, 19, 22, 28, 28, 21, 17.
The theoretic expectation being 24.39454884, we may compare the resulting theoretic standard deviation of 4.939083 with the one calculated from the 10 test-cases, which is 6.069599. Theses values convince us that the same result for the sum and the theoretic probability is not due to a what-so-ever regular pattern in the randomness.

Simulating ten times 3.6 Billion "birthday-paradox" / 23 persons

The only simulation test we have done is the simulation of having 23 persons in the same room and to calculate how many times at least two persons share the same birthday. By random numbers we attributed a given birthday to each person, under the presumption of 365 days per year and an equal probability for every day.
As you can recognize from the test-result in appendix **E**, if you want to do this well-known bet some day, don't forget to insist on the term "at least", as otherwise you risk to lose your bet if your partner insists on excluding the cases with three or more identical birthdays!

Seeing the big number of possibilities for having "at least" two identical birthdays, one might easily understand that the trick to calculate the probability is to only calculate the probability q for not having two identical birthdays, and to calculate p = 1 - q. Not having two identical birthdays is only the line of "cases with 23 unique birthdays".

We must apologize for not yet having calculated the probability of the different variants, as these probability values would increase the usefulness of this test. Did we hear someone say he will take over this challenge ??
The test consists of filling 3.6 billion times a room with 23 persons, to attribute a random birthday to every person, and to first calculate the number of times having at least two persons sharing identical birthdays. On the subtotals per 30 million cycles, we did some calculations on the randomness.
Using the binomial distribution, it is possible to calculate the probability to have unique birthdays, 2 identical and so on. So we calculated these values for the whole test-sample as well. As last analysis we sorted the different positive cases by the encountered variations of identical cases.
We ran this complete test forty times: ten times with AHS-Random taking the last 9 bits from 16 bit integers / ten times with MT19937 taking the last 9 bits from 32 bit integers / ten times with the lrand48 from the SVID, giving an 31 bit unsigned integer from which we took the last 9 bits / ten times a 64 bit LCG running with the same parameters as we use in the AHS, from which we used the bits 33 to 39.
Here are the results (counted cases minus theoretically probable cases) of every test for the four generators:

test-number	AHS	MT19937	lrand48	LCG 64
1	8178	44989	-1546513	-38315
2	-4740	78179	-1546111	-23011
3	3004	-5435	-1547643	12066
4	52061	13665	-1546088	37922
5	3027	-8233	-1546184	23499
6	3873	-35661	-1545867	21777
7	54684	13462	-1545978	5507
8	-30046	-1059	-1545604	20416
9	7284	79064	-1546583	27019
10	-6741	-2669	-1544686	-2669

Average	9058.4	17630.2	-1546125.7	8421.1

As indicated before, we may calculate the expected number of cases with unique birthdays, 2 identical etc. For the AHS and the MT19937 the differences theoretical-counted are as follows (total of 23 x 36'000'000'000 = 828'000'000'000 persons) :

Category	Expected cases	Diff. AHS	Diff. MT19937
unique	779'502'942'681.12	-257'429.12	-188'675.12
2 ident.	23'556'407'608.49	116'038.51	168'181.51
3 ident.	453'007'838.62	6'417.38	-47'222.62
4 ident.	6'222'635.14	1'353.86	-1'257.14
5 ident.	64'961.57	127.43	-164.57
6 ident.	535.39	8.61	-27.39
7 ident.	3.57	-0.57	-0.57

Conclusion of the tests

By presenting these "first-light" results from a completely new type of random number generators, the first software "flipping a coin" and built on the principle of "give chance a chance", we hope to have convinced you that this method is worth a serious consideration. Concerning the potential use in cryptography it seems that the usefulness for different applications is indisputable. In the field of scientific simulations the tests have shown that it may enter the racecourse without wrong modesty; not in order to become the most powerful horse, but maybe to become the best horse for replacing by software the physical random generators in simulations.

RPP – OTP

Randomly Permuted Positions - One-Time Pad

Introduction

The invention of the one-time pad is considered a combined work of Gilbert Vernam of AT&T and Captain Joseph Mauborgne. To summarize we may retain as characteristic principle that the one-time pad is the method to add to a given plaintext a string of randomly chosen characters of the same length as the original plaintext. The resulting ciphertext is considered to be cryptographically secure (proved mathematically by Claude Shanon). This is only true if we use a different string of random characters for every new encryption, in order to avoid possible attacks based on statistics comparing collected ciphertexts. Thus the name "one-time".

There exist some objections against the use of the OTP in modern cryptography. The first ones concern the difficulties for safely distributing the enormous volumes of random data needed and the secure storing of this random data, while the second group of objections concerns the possibility of altering the message during the transmission over unsecured transmission channels (e.g. the public Internet).

The first objections may be considered as solved with the presentation of the AHS-random number generator, which allows to store virtually on a smartcard or a USB-Stick 2exp64 different secret random strings of a length of 2exp64 bits each, in a secured (password protected) and user-friendly way. To solve the second problem, the use of one of the modern message-authentication algorithms seems to be the logical solution. Although this combination alone guarantees unbreakable security, the method presented here as RPP-OTP will increase the protection and the number of possible fields of application.

The Randomly Permuted Positions - One-Time Pad

In order to allow not only the secure transmission of files, but also to serve in a full-duplex live communication like e-banking, the RPP-OTP method breaks the messages into 1000 bytes blocks. Every block is then transformed in a 1024 bytes long cipher-text block. This block contains information about the block-number (of the file or the session), the length of the text stored in the datagram and a message-authentication information. In our test-implementation we use a derived work from the MD 5 algorithm.

The term "randomly permuted positions" indicates that during the encryption every byte of the original text has changed its position in the ciphertext in a random way, i.e. the byte 5 may be in the first block on position 844, in block two on the position 45, and so on. The motivation to do the encryption this way is based on the fact that a lot of communication messages (like e-banking) very often use the same standard and known small text pieces in all messages. The resulting cipher-text in which every byte has randomly changed its position and every bit has twice been XOR-ed with different random bits, leaves any possible attacker with a bitstring of 8192 perfect random bits.
Even the positions of the message-authentication code (16 byte = 128 bit) are unknown, so no attack on this information is possible.

The encryption goes as follow:

a) we produce 8192 random bits, the base, in a memory area organized in bytes
b) we prepare an empty list of 1024 flags and a memory space of 1024 bytes to store the cypher-text
c) we store on 64 bits (8 bytes) the block-number (54 bits) and the length of the datagram (10 bits)
d) we store these 8 bytes to a 1024 byte block, followed with up to 1000 bytes plain-text and 16 zero bytes (if the plain-text is smaller than 1000, the remaining bytes are zero)
e) we calculate the message digest as authentication and store the resulting 16 bytes to the last 16 zero bytes, so that we now have the 1024 bytes original text to encrypt
f) we store zero to the cipher-position-counter and the original-text-counter
g) we add, from the base, the lowest 5 bits (0 to 31) from the byte indicated by the position counter to the position counter, taking care that all additions to this counter have to be followed by a subtraction of 1024 if the result is 1024 or above
h) we check if the flag corresponding to the position counter is empty, and if not we increase the position counter by one until we find an empty position
i) we store the result of the byte from the original text referenced by the original-text-counter XOR-ed with the byte of the base referenced by the cipher-position-counter to the same byte-position in the cipher-block
j) we flag the same position in the flag-list as occupied
k) we increase by one the original-text-counter and the position counter
l) we repeat 1023 times the steps g) - k)
m) we produce the next 8192 random bits
n) we XOR the cipher-block with this 8192 bit-string, giving us the cipher-text to send

The decryption on the receiver side is done in the opposite direction:

a) we XOR the cipher-text with the second block of random bits
b) we use the first block, the base, of 8192 random bits to find back the positions of the original text, and we XOR the cipher with the corresponding byte from the base to give us the original text
c) after completing the 1024 bytes of the original text, we check the block-number and the length of the datagram, and are able to detect an error
d) we save the last 16 bytes, the message authentication and replace these bytes with zeros
e) we recalculate the message-digest and compare the result with the saved 16 bytes, allowing us to detect any alteration by any means, e.g. by transmission error or by an attacker

If you feel that this method seems to be a lengthy process, don't forget that computers are in charge to encrypt and decrypt the messages. A Pentium IV with EM64T running at 3 Ghz is able to process around 6000 blocks/sec, generation of the AHS-random numbers included, giving a throughput of 6 Megabytes/sec.
Of course this method is not the solution for 10 Gigabit/sec links, but the combination of a symmetric encryption like AES combined with RPP-OTP for the key-exchange will guarantee the same high-security as the use of the so-called quantum cryptography, and offer as benefits the low costs and the possibility to easily link long distances, like Europe with Australia.
The small physical supports containing the MPU with protected memory may be programmed

in pairs with a secret BFT of the AHS-random. Copying during the physical transportation to the distant partner is impossible, thus avoiding a possible source of information leak in case that the random numbers for a one-time pad would have to be exchanged using a CD or tape. If we use the AHS-random number generator in the MPU, the generation of new keys may be executed as well.

The security of the RPP-OTP method becomes more evident if we think of small messages for authentication and login for example. As they are now always hidden in 8192 random bits it becomes impossible for an attacker to cryptanalyse the messages transmitted.

If we use the AHS-Random generator, it is also very practical that we may openly indicate (in the subject of an e-mail or in the synchronization of a full-duplex channel) the seed to use for decrypting. In full duplex the responder may use the first seed plus one in order to avoid the usage of the same random numbers twice.

To extend the usage over point-to-point communications, there exist two possibilities. Where appropriate we may create a trusted and secured post-office which shares a different BFT with every participant. Now every member of the group has the possibility to send and receive secret messages from any other member, as the post-office internally decrypts the received message (with authentication), and sends it, after a new encryption with the appropriate BFT, to the addressee. This seems to be the best solution for local authorities, bigger corporations and so on.

A different solution exists for smaller closed groups to share a common BFT. Every encryption unit has the authorization to only use a restricted number-space of the 2exp64 possible seeds. The first encrypted RPP-OTP block includes the one or more addressees to whom one wants to send the message. The decryption program of the units of other members will then refuse to decrypt the message if its own member-number is not included in the addressee-list of the first block.

Other applications may concern the distribution of secret papers inside an organization, by including a decrypting unit between the computer and the printer. The addressee, by using his own smartcard with password-protection, is able to print out the document. The plaintext never shows up inside the company's IT-network.

In the same way one may organize a trusted company-wide computer-network where all sensitive computers are shielded by an RPP-OTP system, nevertheless allowing all internal communication over the Internet. This may apply to lawyer offices, patent attorneys, bankers etc.

Test results / RPP - OTP

We checked the encryption with the RPP-OTP method in combination with the AHS-RNG. For the first test we encrypted the same text, with a length of almost 1000 bytes, one million times using the same BFT of 64 KB and seeds from 0 to 999'999.
For the second test we encrypted the same text using one million different BFTs from 64 KB, but always with the same seed of 0.

```
Test type: 1'000'000 RPP-OTP encrypted blocks with the same original
           text are tested (bit-difference-count), each against all
           the others.

Tested block size: 8192 bits = 1 RPP-OTP block
Theoretical values in brackets
========================================================================

First test
----------

Total tested pairs:  499'999'500'000

Total bits different:  2'047'997'927'056'690
Average bits per pair:  4095.999950113330   (4096)

Total pairs with less than 4096 bits:  247796100936
Total pairs with more than 4096 bits:  247795914900
Ratio between 'less' and 'more':  1.000000750762   (1)

Total std.dev.:   45.254863069189    (45.254833995939)
Std.dev. left :   45.254930841773    (45.254833995939)
Std.dev. right:   45.254795296453    (45.254833995939)

Lowest value in the test:  3784  =  46.191 % of 8192
Highest value in the test: 4416  =  53.906 % of 8192
99.9999 percent of the pairs are in the range from  3875 up to  4317
=====================================================================

Second test
-----------

Total tested pairs:  499.999.500.000

Total bits different :  2.047.997.888.606.412
Average bits per pair:  4095.999873212697   (4096)

Total pairs with less than 4096 bits:  247796661872
Total pairs with more than 4096 bits:  247795300887
Ratio between 'less' and 'more':  1.000005492376   (1)

Total std.dev.:   45.254797068507    (45.254833995939)
Std.dev. left :   45.254817823404    (45.254833995939)
Std.dev. right:   45.254776313488    (45.254833995939)

Lowest value in the test:  3776  =  46.093 % of 8192
Highest value in the test: 4405  =  53.771 % of 8192
99.9999 percent of the pairs are in the range from  3875 up to 4317
====================================================================
```

Glossary of terms used in AHS-Random

BFT Bit-Fishing-Table

One-dimensional bit-table filled randomly with an equal number of “0” and “1” bits. The dimension (total of the random bits) has to be an exponent of two. Depending on the type of application the exponent may vary from 16 to a technical limit of 32.
The technical limit refers to a 32-bit processor architecture. Thus the number of bits may be 65'536 bits (8 KB), 131'072 bits (16 KB), 262'144 bits (32 KB), 524'288 bits (64 KB), and so on.
The random bits in the BFT form the basis for the “endless” variations and the unpredictability of the produced random number output of AHS-RNG.
For crypto applications the 8KB, 16KB, 32KB and 64KB versions are of special interest, as they easily fit into the secured memories of smartcards or USB sticks.
Considering that the BFT, and not the seed, is the only element to be held secret in crypto applications, one might wonder how a “secret” of only 8KB may be sufficient for top secured applications.

From the 2.003529 * 10exp19'728 possible tables of 8KB (2exp65'536), there are 6.244451 * 10exp19'725 different tables with 32'768 zero-bits and 32'768 one-bits.
If we have a given BFT of 8 Kbyte, we would need to produce approximately 10exp141 tables before we would find one with 55 % or more identical bits compared to the original. To find one with 60 % or more identical bits, we need to produce approximately 10exp572 tables, and to come up to 70 % identical bits, we need to produce 10exp2'341 tables, and so on. If we suppose that the population on earth will reach 100 billion people, and that everybody will need one table per second, the odds are very, very strong that in 1'000 years, we will not see two tables having 55% or more identical bit-positions. Indeed, as the probability is 1 to 10exp141 to get a table with 55% or more bit-positions identical to a given table, we need more than 10exp70 tables to find two tables with more than 55% identical bits, by applying the so-called “birthday-paradox” (which, by the way, is not a paradox, but a fact that can easily be explained by means of the principles of the theory of probability).
Considering that the total number of produced tables in 1'000 years would sum up to 3.153 * 10exp21, we have to admit that the odds are very, very strong.

The new invented method in AHS-Random efficiently transforms this enormous potential of a simple 8 KByte secret table into billions of billions of unpredictable and well distributed random numbers.
Every time we double the size of the BFT, the exponents indicated in the example doubles as well.
To find a 64 Kbyte (524'288 bits) BFT-table with 55% or more identical bits compared to a given table, we need to produce approximately 10exp1'128 BFT-tables.

BM Basic Modifier

A table of 32-bit unsigned integers with a recommended dimension of 16 elements.
During the seeding process the values of these integers are calculated and they are not modified until the next seeding.
The recommended method for the calculation of these values is to alternately fill blocks of two values, one block with the LCG 64-bit random number XORed with randomly fished bits of the BFT to guarantee the best secrecy of these values, and the next block with unmodified LCG random numbers to guarantee the uniqueness for any given seed out of the 2exp64 possible ones.

These values interfere cyclically in the production of the FBMs. Every time the requested random number of 8, 16, 24 or 32 bits is terminated, the FBM3 is transferred to the FBM4, the FBM2 to the FBM3, and a new FBM is created by XORing the next BM with the last 32 bits generated. After using the BM16, the next cycle begins again with the BM1.

FBM FeedBack Modifier

A table of four 32-bit values (FBM1 to FBM4) used for the creation of the BRVs. The creation of the FBMs is described in the last paragraph of BM. The feedback of the last 32 bits produced influences the value of the generated FBM, but does not determine the value itself. Nevertheless we use the expression “feedback” to indicate that the last bits generated influence the calculation of these values. In appendix **D** you find an illustration of the effect of this feedback.

BRV Basic Randomness Value

A table of four 32-bit values (BRV1 to BRV4).
After the production of one bit, a new LCG 64-bit random number is generated, and one of the BRV, in a cyclic way, is newly calculated, by XORing the upper 32 bits of the LCG random number with one of the FBMs. The number of the FBM to be used is also determined in a cyclic way. The role of the BRVs is to deliver during 4 cycles every time one fourth of the bits needed to assemble the address of the bit to be “fished” from the BFT. It is very important to use every BRV-bit only once, in order to guarantee a well distributed random number production.

FAAP Final Address Assembling Parameter

16 special values (4 groups of 4) used to assemble the final BFT-address for "fishing" the next bit. To easily understand the specialty of these values, we may imagine every bitposition of the BFT-address as a separate mini-chessboard of 4 x 4 squares. The columns are numbered A, B, C, D, the rows 1, 2, 3, 4.
We now have to put four queens on this miniboard in such a way that one queen will be in every row and in every line. One possibility is to put them in the squares A1, B2, C3 and D4.

As we may now permute the rows or the columns, we will find that there exist 24 (4!) possibilities to arrange the 4 queens. Every square will have a total of 6 queens on all the 24 different miniboards. Let us suppose that we need a 19-bit address-space for a 64 Kbyte BFT. In a first step we randomly discard 4 miniboards having together one queen on every square. There exist 24 possibilities to define the subgroup to discard. From the remaining 20 boards we randomly decide the order to take 19 boards to calculate the FAAPs. Each FAAP value corresponds to a given square on the chessboard. From the first board selected we write a zero bit to the first bitposition of every FAAP value if in the corresponding square there is no queen, and a one bit if there is a queen. After we have processed board 19, the FAAP values are calculated. In every column and in every row we will have 3 values with 5 bits and one with 4 bits.
Don’t be afraid, in practice we do not have to play with 24 mini-chessboards, because on a 3 Ghz Intel Pentium IV we are able to calculate randomly about 100’000 different FAAP tables per second. The total number of different tables we can find this way is 58’389’648’196’239’360’000 for a 19-bit BFT-table (64 Kbyte).

To assemble the FA (final address) in order to determine the next bit to be extracted from the BFT, we use the four parameters from one line (A1, A2, A3, A4 , the next cycle from the B line, then C and D). The correctly calculated FAAP guarantees that every bit of the address is assembled properly.

As we have seen in the description of the BRV, every BRV will participate in the address-assembling with different bits during 4 cycles. It is very important that we never use the same bit twice, otherwise the good distribution of the random numbers produced will be in danger.
Having one bit per position in the 4 parameters per row fulfills this requirement.

We may arbitrarily choose a given table (that looks "very random") and include it in the program instructions, or we may randomly generate tables in case we want to use a random FAAP table as secret session key in crypto applications .
As illustration of the described method, please find in appendix A two examples of randomly generated tables for a 19 bit BFT (64 Kbyte).

FA Final Address

The address (number) of the bit to be extracted from the BFT in order to form a bit position in the random number to be produced

The final address is an assembling of different bit positions from the four BRVs. In order to get well distributed random numbers, the FAs produced also have to be randomly distributed over the whole address space of the BFT.
Tests of the number of accesses per bit in the BFT have confirmed that the invented AHS-Random method to determine the FA is extremely close to the theoretical value, if we calculate the variance and the standard deviation for the number of accesses to the different bits.

Run-time randomness

The invented method of the AHS-RNG allows a unique way to increase the randomness during the production of the random numbers. As explained, you have the choice to ask the function to produce 8, 16, 24 or 32 bits per function call. The function updates the FBMs before returning the bits produced. This property allows the user to influence the production of the next whole string by asking two times 16 bits instead of one time 32 bits. The application-program may use a simple source of some randomness, like the clock() variable, to decide between these two possibilities. A few lines to illustrate the explication:

<u>At program-start</u>

```
clock_t x;
x = clock()%997;
```

<u>For producing the 32 bits</u>

```
if (x != 0) {
   r32 = ahsrnd(4);
   x--;
  }
else {
   r16hbits = ahsrnd(2);
   r16lbits = ahsrnd(2);
   x = clock()%997;
  }
```

By doing so the produced random string will be unique and unreproducible after a few thousand function calls, but nevertheless well distributed and statistically correct.

This method is completely different to a possible random re-seeding of a LCG, as that would not produce new random numbers, but only random numbers from a different part of the periodic cycle.

APPENDIX A

Per Column			Per Row		
a1.	0010110000000000110	90118	a1.	0010110000000000110	90118
a2.	0000000110000011000	3096	b1.	1000000011001000000	263744
a3.	0001000001101000001	33601	c1.	0000001000110010001	4497
a4.	1100001000010100000	397472	d1.	0101000100000101000	165928
total:	1111111111111111111	524287		1111111111111111111	524287
b1.	1000000011001000000	263744	a2.	0000000110000011000	3096
b2.	0001001000000100011	36899	b2.	0001001000000100011	36899
b3.	0100100000010001100	147596	c2.	1110000000001000100	458820
b4.	0010010100100010000	76048	d2.	0000110001110000000	25472
total:	1111111111111111111	524287		1111111111111111111	524287
c1.	0000001000110010001	4497	a3.	0001000001101000001	33601
c2.	1110000000001000100	458820	b3.	0100100000010001100	147596
c3.	0000010110000100010	11298	c3.	0000010110000100010	11298
c4.	0001100001000001000	49672	d3.	1010001000000010000	331792
total:	1111111111111111111	524287		1111111111111111111	524287
d1.	0101000100000101000	165928	a4.	1100001000010100000	397472
d2.	0000110001110000000	25472	b4.	0010010100100010000	76048
d3.	1010001000000010000	331792	c4.	0001100001000001000	49672
d4.	0000000010001000111	1095	d4.	0000000010001000111	1095
total:	1111111111111111111	524287		1111111111111111111	524287

a1.	0010000001000101000	66088	a1.	0010000001000101000	66088
a2.	0000011000010000101	12421	b1.	0001100000010010100	49300
a3.	0100100010101000000	148800	c1.	1000000100101000001	264513
a4.	1001000100000010010	296978	d1.	0100011010000000010	144386
total:	1111111111111111111	524287		1111111111111111111	524287
b1.	0001100000010010100	49300	a2.	0000011000010000101	12421
b2.	0110000100001100000	198752	b2.	0110000100001100000	198752
b3.	1000010001000000011	270851	c2.	0000100010000010010	17426
b4.	0000001010100001000	5384	d2.	1001000001100001000	295688
total:	1111111111111111111	524287		1111111111111111111	524287
c1.	1000000100101000001	264513	a3.	0100100010101000000	148800
c2.	0000100010000010010	17426	b3.	1000010001000000011	270851
c3.	0001001000000101100	36908	c3.	0001001000000101100	36908
c4.	0110010001010000000	205440	d3.	0010000100010010000	67728
total:	1111111111111111111	524287		1111111111111111111	524287
d1.	0100011010000000010	144386	a4.	1001000100000010010	296978
d2.	1001000001100001000	295688	b4.	0000001010100001000	5384
d3.	0010000100010010000	67728	c4.	0110010001010000000	205440
d4.	0000100000001100101	16485	d4.	0000100000001100101	16485
total:	1111111111111111111	524287		1111111111111111111	524287

APPENDIX B

First-bit counting from 1000 Billions seedings

Bits ident.	EXPECTED	ZEROS	ONES
1	250000000000.00	250005340837	249993933978
2	125000000000.00	125003907692	124996597854
3	62500000000.00	62501990060	62498320818
4	31250000000.00	31250972089	31248918322
5	15625000000.00	15625670577	15624468360
6	7812500000.00	7812827960	7812235392
7	3906250000.00	3906222126	3906060268
8	1953125000.00	1953192065	1953067529
9	976562500.00	976619544	976507281
10	488281250.00	488310505	488249523
11	244140625.00	244130974	244151810
12	122070312.50	122073726	122072053
13	61035156.25	61058159	61028351
14	30517578.13	30519528	30510457
15	15258789.06	15259016	15264620
16	7629394.53	7629631	7628766
17	3814697.27	3816690	3817860
18	1907348.63	1903552	1906014
19	953674.32	953198	953784
20	476837.16	477444	476184
21	238418.58	238924	238854
22	119209.29	118687	120261
23	59604.64	59475	59866
24	29802.32	29651	29854
25	14901.16	15124	14921
26	7450.58	7510	7438
27	3725.29	3658	3850
28	1862.65	1817	1785
29	931.32	908	962
30	465.66	459	459
31	232.83	208	259
32	116.42	116	125
33	58.21	47	75
34	29.10	34	31
35	14.55	11	14
36	7.28	7	1
37	3.64	3	3
38	1.82	3	3
TOTAL:		500013352015	499986647985

APPENDIX C

10 sorts of 30 billion 64 bit strings

Missing pattern at 32 bit:

Counted 39772588
Expected 39760143.96

	IDENT.	COUNTED	EXPECTED
32 bit	1	277734693	277721397.37
	2	969903481	969930775.78
	3	2258306894	2258296068.61
	4	3943590374	3943503954.64
	5	5509001250	5509011384.62
	6	6413306296	6413333333.31
	7	6399535311	6399516548.61
	8	5587482335	5587513339.07
	9	4336383172	4336481090.39
	10	3028945745	3028997050.20
	11	1923379493	1923390907.51
	12	1119626869	1119560857.33
	13	601569978	601541711.49
	14	300112963	300122879.66
	15	139757103	139755606.46
	16	61027067	61011352.12
	17	25081813	25068198.32
	18	9726682	9727741.25
	19	3575488	3576183.56
	20	1250002	1248967.68
	21	415151	415425.64
	22	131821	131896.12
	23	40278	40055.81
	24	11776	11657.78
	25	3214	3257.15
	26	809	875.03
	27	232	226.37
	28	66	56.47
	29	15	13.60
	30	0	3.17
	31	1	0.71
36 bit	1	193876938039	193877209474.11
	2	42319275732	42319270753.90
	3	6158345881	6158264405.05
	4	672102539	672109061.76
	5	58689936	58682844.54
	6	4270988	4269738.89
	7	266433	266283.76
	8	14444	14531.02
	9	677	704.85
	10	37	30.77
	11	4	1.22
40 bit	1	291925126067	291925208330.05
	2	3982613535	3982566454.22
	3	36217211	36221230.89
	4	247079	247072.63
	5	1378	1348.27
	6	4	6.13
44 bit	1	299488801420	299488845190.95
	2	255380589	255359548.06
	3	145738	145155.10
	4	47	61.88
48 bit	1	299968031219	299968027280.77
	2	15983549	15985507.71
	3	561	567.92
52 bit	1	299997999372	299998001605.21
	2	1000311	999194.07
	3	2	2.22
56 bit	1	299999875468	299999875099.94
	2	62266	62450.02
60 bit	1	299999992250	299999992193.75
	2	3875	3903.13
64 bit	1	299999999512	299999999512.11
	2	244	243.95

APPENDIX D

The influence of the FBM (Feedback modifier)

We produced 2 different bit-strings with the same seed.
Both times we used the same BFT of 64 KB, except that we exchanged 2 bits in the BFT on an arbitrarily selected position ("01" to "10").

From the two bit-strings we produced a new one by XOR-ing the bits from the two strings to get a string with a "1" on the bit-positions differing, and a "0" if both bits are identical.
As long as we have not yet "fished" one of the modified bits, the output must be identical. But when we encounter one of the two bits, two different reactions are possible, depending on the position of this bit in the register "last 32 bits produced" when we end the production-cycle of 8, 16, 24 or 32 bits.

The following dump illustrates the two possibilities:

alain@linux:/ux> od -tx1 arbS4aa|more

```
0000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
0104300 00 00 00 00 00 80 00 00 00 00 00 00 00 00 00 00
0104320 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
0204740 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0204760 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
0317500 00 01 00 00 00 20 20 00 08 08 80 88 14 67 69 20
0317520 51 08 20 69 c9 d1 49 49 db 54 c7 60 34 3b 20 8f
0317540 43 13 24 2b 74 6e 65 d6 fd 26 44 cd 1d 60 e8 80
0317560 9c 25 62 74 3f f7 34 0a 80 42 5c d0 f8 6b b4 82
0317600 46 2c 03 58 eb 59 81 60 87 9e 24 ec 54 47 e5 48
0317620 6f ef db b0 b9 20 cd c5 69 4f db c3 64 c3 f9 bb
0317640 b1 7f 4c 9d 45 68 ec 4f 42 e8 1f 8d 19 d1 68 5f
```

<u>One bit different in the first and second case:</u>
The position in the file (x00800000, x08000000) in 32 bit blocs indicate that the different bits are not in the range of the relevant last 19 bits (for 64 KB BFT) for XOR-ing the FBM for the calculation of the next BRVs, so no influence in the next rounds.

<u>One bit different in the third case:</u>
The dynamic of the third case is very interesting. In the production of the 32 bits, the difference occurred on the bit position 17, producing an effect on the next cycle of 32 bits.
This cycle now shows a difference of 2 bits within the interval of 8 bits.
The next cycle produces a difference of 5 bits in the production of 32 bits, then 12, 9, 11, 15, 13, 11, 18, 17, 11, 14, 18 ....
and the separation in 2 different random bit-streams is accomplished!

APPENDIX E

```
=========================================================================
RESULTS FROM 3.6 BILLION TEST-CASES WITH 23 PERSONS / RNG: AHS-RNG 64 KB
=========================================================================
Theoretical number of cases by probability calculation:    1826270044
Total counted cases with at least two identical birthdays: 1826278222
                                                           ----------
                 Difference between theoretical / counted:       8178
                                                           ----------
                 Percentage of the difference : 0.00044780 %

Theoretical probability :    0.507297234
Counted probability :        0.507299506
                             -----------
Difference theor./counted:  +0.000002272

======   RECAP OF THE PROBABILITY PER SUB-TOTAL OF 30 MILLION TEST-CASES EACH   =======
-----------------------------------------------------------------------------------------
 0.507279   0.507382   0.507260   0.507335   0.507238   0.507176   0.507168   0.507446
 0.507212   0.507215   0.507357   0.507471   0.507379   0.507377   0.507227   0.507335
 0.507390   0.507461   0.507349   0.507321   0.507414   0.507267   0.507195   0.507342
 0.507455   0.507368   0.507170   0.507333   0.507401   0.507211   0.507515   0.507246
 0.507274   0.507374   0.507385   0.507284   0.507245   0.507301   0.507276   0.507282
 0.507244   0.507332   0.507279   0.507292   0.507427   0.507288   0.507273   0.507246
 0.507213   0.507328   0.507190   0.507183   0.507054   0.507414   0.507217   0.507320
 0.507358   0.507376   0.507355   0.507178   0.507366   0.507306   0.507122   0.507240
 0.507298   0.507182   0.507152   0.507407   0.507211   0.507128   0.507238   0.507285
 0.507223   0.507398   0.507408   0.507439   0.507403   0.507465   0.507247   0.507464
 0.507298   0.507217   0.507335   0.507215   0.507202   0.507420   0.507399   0.507304
 0.507217   0.507370   0.507256   0.507421   0.507048   0.507344   0.507374   0.507224
 0.507330   0.507272   0.507229   0.507156   0.507403   0.507394   0.507147   0.507321
 0.507222   0.507324   0.507157   0.507360   0.507280   0.507341   0.507343   0.507317
 0.507258   0.507224   0.507423   0.507363   0.507383   0.507216   0.507190   0.507375
-----------------------------------------------------------------------------------------
aver. per 30 Mio.: theor. 15218917.030 / counted 15218985.183 / diff.      +68.154
std.dev. of proba: theor.     2738.321 / counted     2816.567 / diff.      +78.246
std.dev.real val.: theor.     2738.321 / counted     2815.742 / diff.      +77.422
Number of sub-totals below theor. value : 58   /  above theor. value : 62
=========================================================================================
RECAP OF THE NUMBER OF DIFFERENT OCCURENCES PER TEST-CASE
----------------------------------------------------------
              1  case  with   6  unique birthdays
              4  cases with   7  unique birthdays
             24  cases with   8  unique birthdays
            372  cases with   9  unique birthdays
            964  cases with  10  unique birthdays
          15421  cases with  11  unique birthdays
          24459  cases with  12  unique birthdays
         403916  cases with  13  unique birthdays
         374779  cases with  14  unique birthdays
        6572043  cases with  15  unique birthdays
        3247479  cases with  16  unique birthdays
       66304347  cases with  17  unique birthdays
       14666963  cases with  18  unique birthdays
      399734188  cases with  19  unique birthdays
       26622667  cases with  20  unique birthdays
     1308310595  cases with  21  unique birthdays
     1773721778  cases with  23  unique birthdays
-----------------------------------------------------------------------------
binom.prob.: 77950294268.112  Counted: 77950220010.000  Diff:  -74258.112
-----------------------------------------------------------------------------
     1323216633  cases with   1  time    2  ident. birthdays
      402636576  cases with   2  times   2  ident. birthdays
       66364064  cases with   3  times   2  ident. birthdays
        6509441  cases with   4  times   2  ident. birthdays
         393399  cases with   5  times   2  ident. birthdays
          14684  cases with   6  times   2  ident. birthdays
            345  cases with   7  times   2  ident. birthdays
              3  cases with   8  times   2  ident. birthdays
-----------------------------------------------------------------------------
binom.prob.:  2355640760.849  Counted:  2355677279.000  Diff:   36518.151
-----------------------------------------------------------------------------
       44930558  cases with   1  time    3  ident. birthdays
         185140  cases with   2  times   3  ident. birthdays
```

```
             333  cases with   3  times   3  ident. birthdays
---------------------------------------------------------------------------
binom.prob.:    45300783.862  Counted:    45301837.000  Diff:    1053.138
---------------------------------------------------------------------------
          621674  cases with   1  time    4  ident. birthdays
              24  cases with   2  times   4  ident. birthdays
---------------------------------------------------------------------------
binom.prob.:      622263.514  Counted:      621722.000  Diff:    -541.514
---------------------------------------------------------------------------
            6531  cases with   1  time    5  ident. birthdays
---------------------------------------------------------------------------
binom.prob.:        6496.157  Counted:        6531.000  Diff:      34.843
---------------------------------------------------------------------------
              63  cases with   1  time    6  ident. birthdays
---------------------------------------------------------------------------
binom.prob.:          53.539  Counted:          63.000  Diff:       9.461
===========================================================================
ANALYSIS OF THE VARIATIONS OF THE ENCOUNTERED 'AT LEAST 2 IDENTICAL'
-----------------------------------------------------------------------
        16 times    1 x 6 ident. +  1 x 2 ident.
        47 times    1 x 6 ident.

         2 times    1 x 5 ident. +  1 x 3 ident. +  2 x 2 ident.
        12 times    1 x 5 ident. +  1 x 3 ident. +  1 x 2 ident.
        29 times    1 x 5 ident. +  1 x 3 ident.
         2 times    1 x 5 ident. +  4 x 2 ident.
        25 times    1 x 5 ident. +  3 x 2 ident.
       302 times    1 x 5 ident. +  2 x 2 ident.
      1899 times    1 x 5 ident. +  1 x 2 ident.
      4260 times    1 x 5 ident.

         2 times    2 x 4 ident. +  2 x 2 ident.
         3 times    2 x 4 ident. +  1 x 2 ident.
        19 times    2 x 4 ident.
         1 time     1 x 4 ident. +  2 x 3 ident. +  1 x 2 ident.
        10 times    1 x 4 ident. +  2 x 3 ident.
         3 times    1 x 4 ident. +  1 x 3 ident. +  4 x 2 ident.
        12 times    1 x 4 ident. +  1 x 3 ident. +  3 x 2 ident.
       137 times    1 x 4 ident. +  1 x 3 ident. +  2 x 2 ident.
      1085 times    1 x 4 ident. +  1 x 3 ident. +  1 x 2 ident.
      3116 times    1 x 4 ident. +  1 x 3 ident.
         5 times    1 x 4 ident. +  5 x 2 ident.
       197 times    1 x 4 ident. +  4 x 2 ident.
      3760 times    1 x 4 ident. +  3 x 2 ident.
     37294 times    1 x 4 ident. +  2 x 2 ident.
    190553 times    1 x 4 ident. +  1 x 2 ident.
    385501 times    1 x 4 ident.

         4 times    3 x 3 ident. +  2 x 2 ident.
        76 times    3 x 3 ident. +  1 x 2 ident.
       253 times    3 x 3 ident.
         1 time     2 x 3 ident. +  5 x 2 ident.
        23 times    2 x 3 ident. +  4 x 2 ident.
       556 times    2 x 3 ident. +  3 x 2 ident.
      7684 times    2 x 3 ident. +  2 x 2 ident.
     49690 times    2 x 3 ident. +  1 x 2 ident.
    127175 times    2 x 3 ident.
         1 time     1 x 3 ident. +  7 x 2 ident.
        21 times    1 x 3 ident. +  6 x 2 ident.
       946 times    1 x 3 ident. +  5 x 2 ident.
     24221 times    1 x 3 ident. +  4 x 2 ident.
    373139 times    1 x 3 ident. +  3 x 2 ident.
   3242464 times    1 x 3 ident. +  2 x 2 ident.
  14662703 times    1 x 3 ident. +  1 x 2 ident.
  26622667 times    1 x 3 ident.

         3 times    8 x 2 ident.
       344 times    7 x 2 ident.
     14663 times    6 x 2 ident.
    392447 times    5 x 2 ident.
   6484995 times    4 x 2 ident.
  65986572 times    3 x 2 ident.
 399348687 times    2 x 2 ident.
1308310595 times    1 x 2 ident.
----------
1826278222 cases
```