VIKA Zero Candidate A
An Informal Paper

By stringzzz, Ghostwarez Co.
02-16-2025

Dedicated to Life’s Little Victories

Introduction & Warning

Firstly, the goal of this project is to have an encryption system to work with as I acquire more tools to be able to properly analyze it and make better design choices in response. Candidate A will serve as a base for this, where I can fix problems as I discover them and create newer, improved candidates. As for this candidate, this was all done as an amateur cryptologist, so it's important to keep this in mind and just assume the tests and design choices were chosen with limited knowledge of cryptology and the mathematics needed for them, so they should be taken lightly. Also, much of the code in the original tests were highly redundant, so they were refactored into functions with later tests. This, along with some other details, were kept as is with the test code that will be found with this paper, simply to avoid introducing any errors or changing something that actually changes the test itself.

This system is based on the idea of using a key-dependent S-Box and P-Box as opposed to fixed boxes. Each box is built with a unique set of key schedule bits by a shuffling function, which correspond to their own separate portion of the original key. The testing started with a more broad set of cipher specs, then it was narrowed to to more and more specific ones until the final tests were done with a single setup that became Candidate A.

The name VIKA Zero stands for "Vanishing Illusion Key Automator" Zero. It hints at that the plaintext and key vanish into the ciphertext like an illusion. The "Key Automator" is due to there being a built-in key generator in the system. There is one in there now, but it's purely for testing purposes. As part of this project, one other goal is to produce a decent, working Cryptographic PRNG to use for the key and IV (CBC-Mode). Lastly, I will note that this system is currently only set up with 128-bit mode. Scaling up the system seems to be not that difficult, but I will wait to do so after learning more of what I need to be able to properly test and compare the results of those different sizes.

REVISED NOTICE: A mistake was found in the key schedule where it was producing many more bytes than needed, and the bytes were distributed for the S-Box and P-Box as 4 times the needed size. All of the affected tests were fixed and ran again to produce the new data, so the revisions are mostly updating that data. Here are the new, correct specifications:

Key Size (bits)	Key Size (Bytes)	Cycles	KS Bytes	S-Box KS Bytes	S-Box Shuffles	P-Box KS Bytes	P-Box Shuffles	KS XOR Bytes
128	16	16	2304	1024	32	1024	256	256

S-Box Test

The purpose of this test was to produce a number of S-Boxes according to a different set of specs, the number of bit-dependent, re-creatable shuffles for the box and the corresponding amount of bits for that process, then compare all possible inputs to their outputs to see how many bits of difference there were.

10,000 S-Boxes of 8x8 bits were generated for each set of 11 box shuffling specifications using the program whose source code is called “sbox_test2B.cpp”. The following table shows the specs of each test and the results gathered from them.

Shuffles	KS bytes used for shuffling	Mean Differences between input and output of S-Box (/2048)	Range of differences	Score	Score (Without Shuffles)
1	256	1023.93	208	208	208
2	512	1023.94	192	193	192
4	1024	1024.29	174	177	174
8	2048	1024.36	176	183	176
16	4096	1023.81	182	197	182
32	8192	1023.58	170	201	170
64	16384	1024.12	184	247	184
128	32768	1023.53	172	299	172
256	65536	1023.98	180	435	180
512	131072	1024.29	170	681	170
1024	262144	1023.77	162	1185	162

The score was determined as follows:
Score = 0 + range + abs(1 - N_SHUFFLES[n_specs]) + abs(1024 - final_average)
Lowest (Best): 0 + 0 + abs(1 - 1) + abs(1024 - 1024) = 0
Highest (Worst): 0 + 2048 + abs(1 - 1024) + abs(1024 - 0) = 4095

Note that 256 shuffles produced the closest to half mean average, though the best score was for the greatest number of shuffles, 1024. After later tests, 32 shuffles for the S-Box and 256 for the P-Box were chosen to keep the number of key schedule bytes needed to a minimum. Had 1024 shuffles been used, it would require 262,144 key schedule bytes, which is unrealistic considering they would have to be produced from just 16 bytes of the original key.

P-Box Tests

Test 4A

The purpose if this test was to examine the properties of the S-Box and P-Box given different specifications, such as the number of shuffles to produce them and the number of encryption cycles. The encryption function itself was simply:

1. Apply S-Box
2. If cycles is a multiple of ‘n_cycles_per_pbox’, apply nybble P-Box (Break to nybbles, use P-Box, then reassemble to bytes afterward)

The source code file for this test was named “pbox_test4A”. 500 original plaintexts were produced, along with 128 copies, all with a single, different bit flipped. For each of the 24 cipher specifications, 250 different pairs of S-Boxes and P-Boxes were set up. For each box setup, each original plaintext was encrypted to produce the original ciphertext. Then, for each flipped plaintext, they were encrypted to produce the flipped ciphertexts. Finally, all the original ciphertext and the corresponding flipped ciphertexts were compared to find the number of different bits, and produce the mean of these differences. The results of this are shown in the following table, sorted by lowest (best) score first:

S-Box Shuffles	P-Box Shuffles	KS Bits Per Box	Cycles Per P-Box	# of Cycles	Mean Differences (/128)	Highest Difference	Lowest Difference	Range	Score
1024	8192	262144	2	16	61.86980	94	1	93	2.13017
512	4096	131072	2	16	61.72430	93	1	92	2.27570
64	512	16384	2	16	61.66000	94	1	93	2.34003
128	1024	32768	2	16	61.58390	93	1	92	2.41607
256	2048	65536	2	16	61.55120	93	1	92	2.44877
32	256	8192	2	16	61.49220	92	1	91	2.50782
512	4096	131072	2	14	57.73760	92	1	91	6.26242
64	512	16384	2	14	57.69930	92	1	91	6.30068
128	1024	32768	2	14	57.51430	94	1	93	6.48566
256	2048	65536	2	14	57.49810	92	1	91	6.50189
32	256	8192	2	14	57.35950	92	1	91	6.64053
1024	8192	262144	2	14	57.19440	92	1	91	6.80561
512	4096	131072	2	12	48.88600	91	1	90	15.11400
256	2048	65536	2	12	48.62390	93	1	92	15.37610
64	512	16384	2	12	48.53500	90	1	89	15.46500
128	1024	32768	2	12	48.45410	90	1	89	15.54590
1024	8192	262144	2	12	48.41010	91	1	90	15.58990
32	256	8192	2	12	48.22710	91	1	90	15.77290
32	256	8192	2	10	35.96130	82	1	81	28.03870
128	1024	32768	2	10	35.90060	81	1	80	28.09940
256	2048	65536	2	10	35.83480	82	1	81	28.16520

512	4096	131072	2	10	35.82540	79	1	78	28.17460
64	512	16384	2	10	35.71750	81	1	80	28.28250
1024	8192	262144	2	10	35.65190	79	1	78	28.34810

The score was calculated by simply taking |64 – Mean Difference|. It's no surprise that the setups with the largest number of cycles had the best scores, while the lower number of cycles were at the bottom. Due to this, the number of cycles were narrowed down to 16 for the remainder of the P-Box tests.

P-Box Test 4B, 4C, & 4D

The following tests used the exact same setup as "pbox_test4A", only the number of cycles is now always 16, and also 1000 original plaintexts were produced to be used with 1000 different boxes instead of the original amount. The only part differing from each of these three tests is the encryption function. Here is the description of each encryption function, along with the results sorted by lowest (best) score first.

4B Encrypt (Exactly the same encryption function as 4A):

S-Box Shuffles	P-Box Shuffles	KS bits Per Box	Cycles Per P-Box	# of Cycles	Mean Differences	Highest Difference	Lowest Difference	Range	Score
128	1024	32768	2	16	61.69430	94	1	93	2.30570
64	512	16384	2	16	61.66650	94	1	93	2.33353
32	256	8192	2	16	61.66210	96	1	95	2.33790
1024	8192	262144	2	16	61.58510	95	1	94	2.41491
512	4096	131072	2	16	61.58030	94	1	93	2.41969
256	2048	65536	2	16	61.57420	96	1	95	2.42575

The top 2 seem to be the best, but considering the amount of key schedule bits needed for them, number 3 seems like a good candidate. I will make an important note, however, that at this point the number of operations required to break the plaintext into nybbles, use the P-Box on them, and then reassemble the nybbles back into bytes is quite expensive. I wanted to figure out a way to propagate the changes throughout the block without having to use that many operations. This is where I came up the the "Diffusion Slide". In some loose sense, it's almost like having a mini-CBC Mode inside of the encryption function, using XOR from left to right. This was a much cheaper alternative to the P-Box, but instead of removing that altogether it was just used less often. To be exact, the P-Box is applied every even cycle that's NOT a multiple of 4, while the diffusion slide is applied to every cycle that's a multiple of 4. Here it is along with the results:

4C Encrypt:
1. Apply S-Box
2. If cycles is even but not a multiple of 4, apply P-Box
3. If cycles is a multiple of 4, apply Diffusion Slide.

S-Box Shuffles	P-Box Shuffles	KS bits Per Box	Cycles Per P-Box	# of Cycles	Mean Differences	Highest Difference	Lowest Difference	Range	Score
512	4096	131072	2	16	63.97980	108	1	107	0.02016
256	2048	65536	2	16	63.96510	108	1	107	0.03487
128	1024	32768	2	16	63.96460	100	1	99	0.03541
32	256	8192	2	16	63.96240	111	1	110	0.03761
64	512	16384	2	16	63.95530	120	1	119	0.04471
1024	8192	262144	2	16	63.93710	116	1	115	0.06288

I was actually surprised by the results of this test. Not only did this setup seem more efficient speed-wise (Speed test shown soon), but the mean differences were all almost exactly in the middle. The problem I recognized, however, is that there is not enough going on as is. It is pretty standard to XOR in some key schedule bytes, so while the next test, 4D, is almost exactly the same as the previous, the one new addition is an XOR of some key schedule bytes at the end of each cycle.

4D Encrypt (Exact same as 4C but with and additional XORing of the block bytes with key schedule bytes at the end of each cycle.):

S-Box Shuffles	P-Box Shuffles	KS bits Per Box	Cycles Per P-Box	# of Cycles	Mean Differences	Highest Difference	Lowest Difference	Range	Score
1024	8192	262144	2	16	63.96760	109	1	108	0.03240
128	1024	32768	2	16	63.96190	103	1	102	0.03806
32	256	8192	2	16	63.95450	112	1	111	0.04553
256	2048	65536	2	16	63.95140	109	1	108	0.04856
512	4096	131072	2	16	63.95140	109	1	108	0.04860
64	512	16384	2	16	63.94720	114	1	113	0.05279

The scores on this test were slightly worse, however I thought that the extra key schedule bytes being XORed in was an important trade-off.

Next is the complexity and speed comparisons of the three previous encryption setups. 3 Identical sets of 10,000,000 plaintexts were produced by PRNG, then each was encrypted by the three 4B, 4C, & 4D encryption functions while recording the time with the “chrono” library to get the microseconds elapsed in the process. The results and complexity of the encryption functions are shown in this table:

Test	Total Operations Per Block	Elapsed Time (Microseconds)	Time Per 1 Block (Microseconds)
4B	5993	42710115	4.27101
4C	4309	30423554	3.04236
4D	6117	33969906	3.39699

While 4D had more operations then 4B, it was noticeably faster. 4C was the fastest, but 4D was ultimately chosen in favor over it due to the extra security of the key schedule byte XORing. So far, the key schedule bytes were generated purely by PRNG, next is testing an actual key schedule.

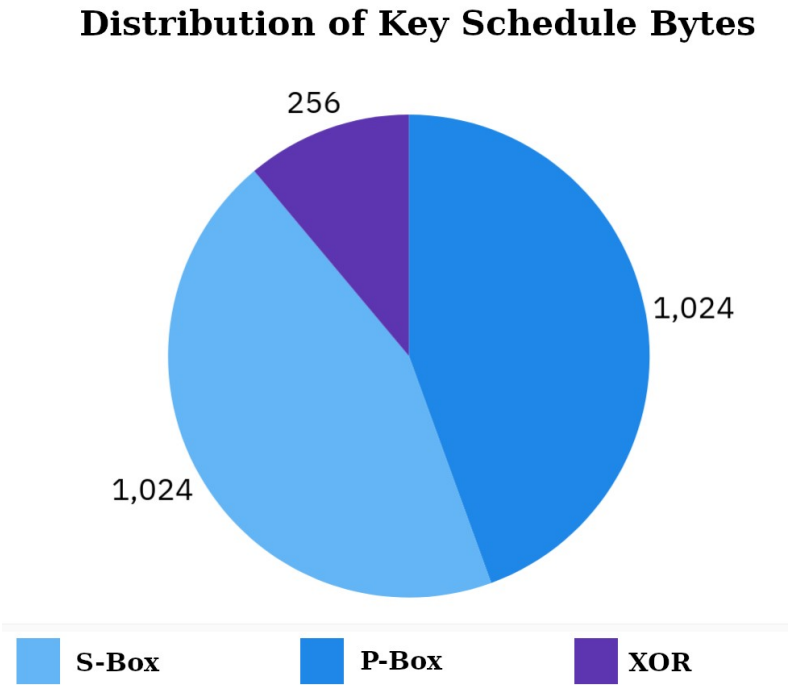
Key Schedule Tests

Direct Key Schedule Test

The following specifications were chosen for the key schedule:

Key Size (bits)	Key Size (Bytes)	Cycles	KS Bytes	S-Box KS Bytes	S-Box Shuffles	P-Box KS Bytes	P-Box Shuffles	KS XOR Bytes
128	16	16	2304	1024	32	1024	256	256

The distribution of the key schedule is also shown here:



So, the key schedule bytes for shuffling the boxes corresponds to almost 44.44% of the original key each, while the XOR bytes correspond to about 11.11%. All of the key schedule bytes are used exclusively, never overlapping or reused.

The algorithm for the key schedule is as follows:

1. Make 16 copies of the original key, each with its bytes shifted over by 0-15 positions.

2. Produce 273 more blocks based off of the first 16 by flipping one bit in the originals (Also keep original 16).

3. Separate half of these blocks to be used to set up a temporary key schedule (2304 bytes) , plus an additional 16 byte block.

4. Using the temporary key schedule, set up a temporary S-Box, P-Box, and XOR bytes.

5. Using the additional block as an IV, CBC-Mode encrypt the remaining half of the blocks (The ones not used for the temporary key schedule).

6. Output the resulting final key schedule.

For this initial test of the key schedule, the goal was to generate an original key schedule for each of 50,000 original keys, along with another set of key schedules produced with copies of the original key but with 1 bit flipped (128 per original key). Then, the resulting “flipped” key schedule bits were compared with the original to determine the number of different bits, and a pop count of the 1 bits of each was also gathered. Finally, the averages of these pop counts and differences were calculated. The results of this are shown here:

Highest Pop Count	Lowest Pop Count	Average Pop Count Range	Average Pop Count	Highest Difference (/18432)	Lowest Difference (/18432)	Average Difference Range	Average Difference (/18432)
52.1213%	47.819%	352.114	49.9999%	9588	8876	352.122	9216.02

The average pop count is very close to half, and the average difference is also close to the half of 9216. In recognition, there are many more statistical tests that should be done here, but as mentioned at the beginning of this paper, I am not equipped with those skills (yet).

Plaintext Avalanche Effect

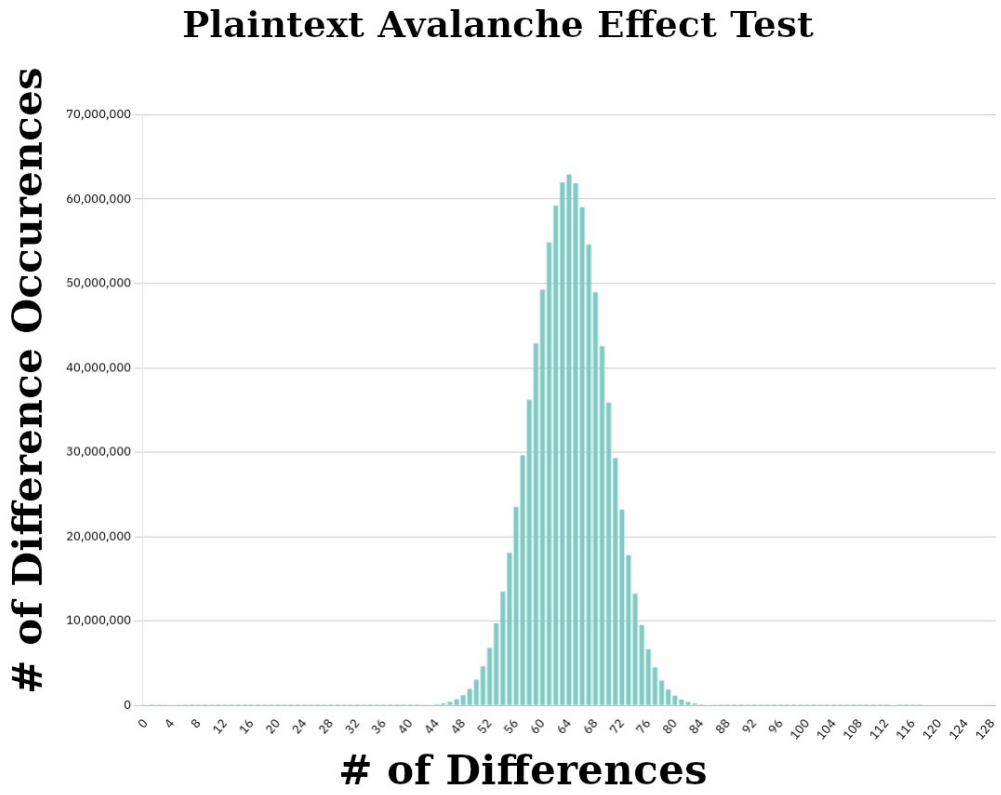
The code for this test is found under the name “keyschedule1A_SPAC_test2.cpp”, though I believe calling it “SPAC” (Strict Plaintext Avalanche Criterion” may have been a mistake. It seems that doing such a test does require statistics, while what this test is actually looking at may be the “Plaintext Avalanche Effect” in general. To save explanation, the same is true for the SKAC and KAE (K = Key) names used in the next test after this one.

For this PAE Test, 2000 plaintext blocks were generated, and just like previous tests the 128 “flipped” plaintext were generated from these original plaintext. Then, for each 3500 keys, the key schedule was produced and the boxes and XOR bytes were set up. This encryption setup was used for the original plaintexts and the flipped plaintexts, and the original ciphertext and flipped ciphertexts were compared for the differences. A count of the numbers of all of these differences were gathered in the process as well.

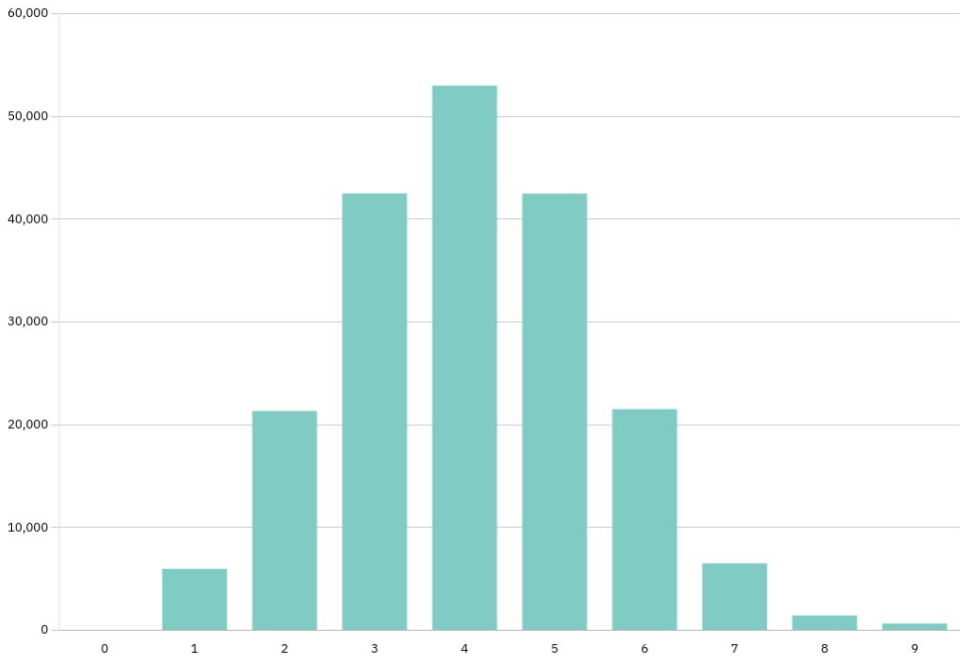
This is the result of the above test:

Highest Difference:	117/128
Lowest Difference:	1/128
Range:	116
Average Difference:	63.955/128

Here is a bar graph of the distribution of these differences from the original ciphertext to the flipped ciphertext:



It appears to be flat on both ends, however, if we zoom in on the left we see this behavior (Next Page):



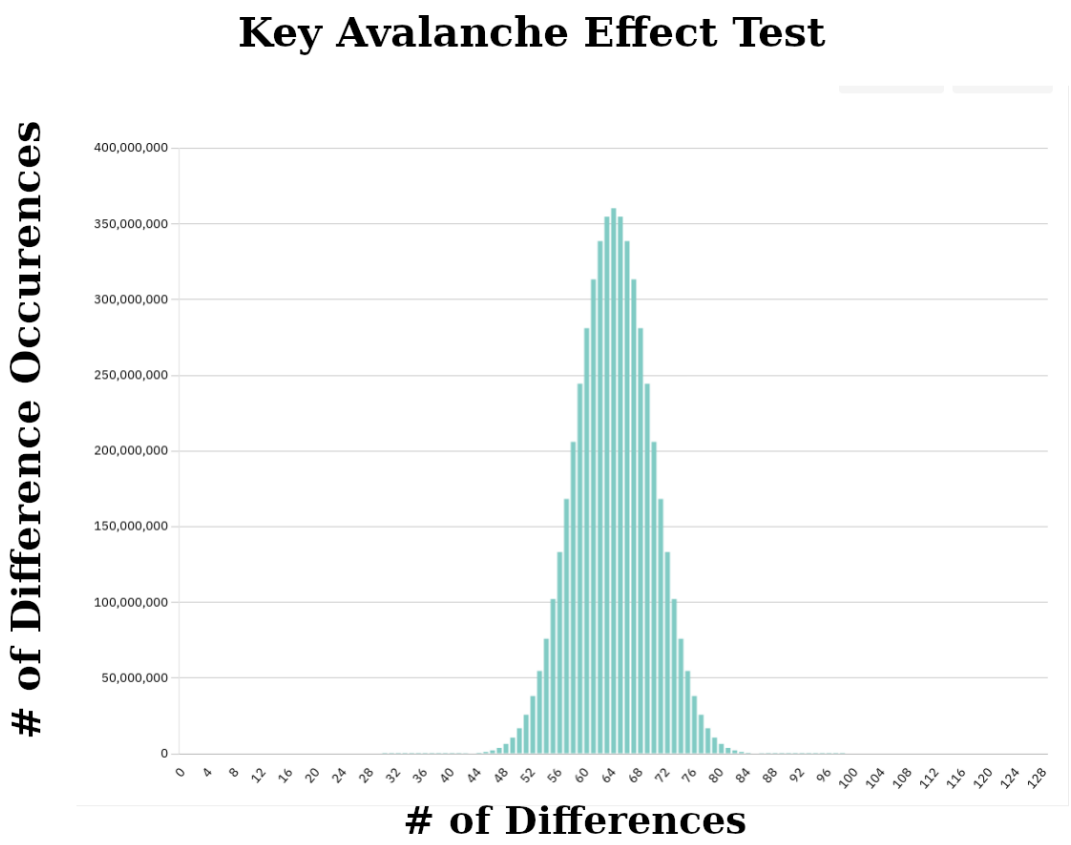
It isn’t shown, but if we zoom in on the right of the 1st graph, it is completely flat. Unfortunately, I don’t know enough of what I need to give a clear explanation of this, but it seems possible this is a sign of bias. That, or just flipping one bit may likely produce a change of only a few bits and that is something perfectly reasonable. Also to note, as far as this zoomed in graph portion goes, the highest occurrence is a difference of 4 bits, a nybble worth. It seems possible this may be related to the fact that the P-Box works on nybbles. Aside from this, due to my lack of knowledge about this, I won’t make any further claim about it.

Key Avalanche Effect

This test is highly similar to the previous, except the plaintext stays as is, and the keys are bit flipped instead. First, 10,000 plaintexts are generated to be used with every single test key set. Then, for each of 4,000 a key is generated along with 128 bit flipped copies of the original key. The key schedule is produced for each one and used to encrypt the plaintext. The resulting ciphertext produced from the original key is compared to the ciphertext produced by each flipped key to get the difference in bits between them. Here are the results of the test:

Highest Difference: 98/128
Lowest Difference: 30/128
Average Difference Range: 52.511
Average Difference: 64/128

Along with the distribution of differences in graph form:



This time, as the range shows, both ends are flat. Again, this statement is to be taken lightly, but it seems on top of having the average difference be as close to half as possible, it also would be a desirable property for the range to be as narrow as possible.

Key Generator

The following section will be very brief. Given that the only property of randomness I know how to test is the pop count of bits over the whole collection of generated keys, there is not much to show here. The “keygeneratortest1B.cpp” system simply produces a PRNG key with “random_device”, then it does the following 4 times, lastly outputting the resulting key as the generated key:

1. Use the current temporary key to produce a key schedule.
2. XOR the entire key schedule into the temporary key block by block.
3. Inject more PRNG bytes into the temporary key bytes with XOR. (Except last loop)

This was done for 1,000,000 generated keys that were collected, then all was converted into bits to get a pop count of them. Here are those results for this test:

Generated key: 1000000/1000000
Pop Count: 63992900/128000000 (49.9944%)

There really isn’t much to interpret here, the pop count is close to 50%, but without being able to do further tests this key generator shouldn’t really be used. Its only purpose is to make testing easier, for generating keys and also for generating the IV in CBC mode. I will gladly expand on it or completely replace it when I am more capable of doing so.

Speed Comparison

The purpose of this test was to compare VIKA Zero Candidate A’s speed performance to my previous encryption system, ANGELITA128. I will not be describing that former system in any detail, aside from it definitely being more complex to due its P-Box working on 2-bit pieces rather than nybbles. For this test, the file “speed_test_file_100MB.txt” was generated with PRNG bytes (104,857,600 to be exact) using a Python script to generate a 100MB text file. Two copies of this file were made, then each system in turn was used to encrypt and decrypt their respective copies in ECB mode while recording their time in microseconds. The following table shows these results:

System	Encrypt/Decrypt	Total Time (Microseconds)	Time Per 1 Block (Microseconds)
ANGELITA128	Encrypt	93743628	14
ANGELITA128	Decrypt	89844511	13
VIKA_Zero Candidate A	Encrypt	19132866	2
VIKA_Zero Candidate A	Decrypt	20119000	3

Comparing the two sets of results yields this:

Encrypt: 93743628 / 19132866 * 100 = 489.9612426% (Roughly 5x faster)
Decrypt: 89844511 / 20119000 * 100 = 446.5654903% (Roughly 4.5x faster)

This might not seem like much, but one of my biggest problems in past encryption system projects was making a system that was horribly inefficient, so these results are really exciting to me that I am actually making progress, at least a bit.

Conclusion

While it is clear that there is much left to be desired from all of these tests, I feel this is a fine foundation to build off of in making the future candidate VIKA Zero systems. One problem I've had that I've just chosen to ignore instead of finding a solution for is that none of these tests really keep track of all the data produced, only the final results. Though all of the test code will be provided with this system to be able to allow others to reproduce similar results, it would be a lot better to have a whole set of produced data to work off of as well. The problem I found with this is dealing with the huge amount of data these tests would produce if actually saving that data instead of just pulling parts like averages off of them. It does seem like the tests did show some desirable properties of an encryption system, but its highly possible that I'm misinterpreting these results in my ignorance.

So, instead of drawing a clear conclusion, I will state what I need to do in order to make other candidates of this system:

1. Learn Number Theory, Abstract Algebra, & Statistics
2. Using the above, study existing cryptosystems, whether they are provably secure or found to have problems in them.
3. Expand on the previous tests and examine the system more deeply.
4. Given what was found in those newer tests, find ways to patch up problems on Candidate A, or start almost from scratch if it turns out to be too flawed.
5. Create Candidate N from the above results, try to work on a decent CPRNG for its key generator.
6. Write increasingly formal papers on each new candidate.
7. Repeat steps 3-6 several times, producing a new candidate each time.
8. After learning even more of what is needed, analyze each candidate further until a final candidate is chosen.
9. Introduce VIKA_Zero, attempt to show the work done on the system myself to hopefully get others to do their own analysis on the system (Peer review process).

As stated at the beginning, this is a very informal paper, so the conclusion is really unconventional and sounds more like my personal chore list. I just wanted to note where I fall short currently in this field, and that I intend to improve by using the above process to the point where I can produce some system that's worthy of study by others in this field. Until then, progress is progress, even if it is only incremental.

--stringzzz
Ghostwarez Co.
02-16-2025