General Purpose Sign/Verify programs For Use With Any Encryption/Decryption Programs

While my teammates implemented specific HMAC signing and verification schemes within their encrypt/decrypt code, I chose to generate general purpose sign/verify code that can be used with any file in hex format. In fact, these sign/verify programs can be used as they are to sign any type of file. The encrypt and decrypt routines remain the same as in part 1 of the project and use the aes128 counter mode of operation. The changes/additions for part 2 are as follows:

1. The ./keygen program has been modified to to accept one of the following 7 inputs to generate a key: md5, sha1, sha224, sha256, sha384, sha512, and aes128 (for my encode/decode). The invocation is as follows:

   ./keygen keytype

   As previously, the output goes to stdout and can be redirected to create a file, ie.:

   ./keygen sha256 >sha256key.txt

   The key types and the sizes generated for them are as follows:

   sha512 - 64 bytes
   sha384 - 48 bytes
   sha256 - 32 bytes
   sha224 - 28 bytes
   sha1 - 20 bytes
   md5 - 16 bytes
   aes128 - 16 bytes

   These definitions appear in the file attr.c

2. Two new programs, sign and verify, have been written and utilize HMACs of the following type: md5, sha1, sha224, sha256, sha384, sha512. The desired HMAC is specified by the user on the command line of the sign routine. sign and verify use a key generated by keygen. Program semantics for sign and verify are as follows

   The invocation for sign looks like:

   ./ sign hmackey hmactype file.txt signedfile.txt

   where:

   hmackey is the key for the desired HMAC type previously generated by keygen
   hmactype is one of the literals md5, sha1, sha224, sha256, sha384, or sha512
   file.txt is the file for which the HMAC tag is to be generated
   signedfile.txt is the signed output file

Note that no output goes to stdout

an example invocation is: ./sign sha256key.txt sha256 ciphertext.txt signedciphertext.txt
if timing statistics are desired, they are indicated by a -t parameter at the end of the command line. For example: ./sign sha512key.txt sha512 ciphertext.txt signedciphertext.txt -t

sign reads the input file in, generates a header containing the HMAC type, timestamp, and generated HMAC tag, and other parameters and writes it, followed by the input file contents (represented by file.txt in the example) to a signed file (represented by signedfile.txt in the example). The HMAC tag is generated on the contents of the input file (file.txt in the example) and the timestamp.

Note that **openssl** routines were used to generate the HMAC tag. **Also note** that if the key specified in the command line has a size that is less than or greater than that expected for the specified hmactype, then the openssl routines will pad it, reduce it to the expected size, or hash it, depending on the implementation of the openssl HMAC scheme invoked. **Also note** that the header produced is in hex format for easy viewing, should you wish to examine it. In production use, it would not. The header size is fixed and holds the maximum size HMAC output that can be generated by the openssl HMAC code. sha512 is the largest generated output size (64 bytes) and the header is sized to accommodate it.

Timing statistics are determined solely by the amount of time the openssl routines take to generate the HMAC. I/O is not taken into account. A 32K buffer is used to read the input file contents and a maximum 32k byte chunk is utilized in the calls to the openssl HMAC routines. Timing is obtained by calls to the builtin C timing libraries 'clock()' function. The clock is started immediately preceding the call to the openssl HMAC procedures and is stopped on its return. The resulting difference in time is accumulated by adding it to a variable that was zeroed on program startup. This process continues until all data for generating the HMAC tag is exhausted (input file contents and timestamp).

The invocation for verify looks like:

./verify hmackey signedfile.txt verifiedfile.txt
where:

hmackey is the key for the desired HMAC type previously generated by keygen and

is the same one used as the hmackey argument to the sign program when signedfile.txt is created. It is the shared key used for HMAC tag generation.

signedfile.txt is the signed input flle, output by the sign program, whose header contains the HMAC tag generated by the openssl routines.

verifiedfile.txt is the output file which is equivalent to the signedfile.txt with the header stripped off, if the verification process is successful. If not successful, the contents of this file are indeterminate and the file should not be used.

an example invocation is: ./verify sha256key.txt signedciphertext.txt verifiedciphertext.txt
if timing statistics are desired, they are indicated by a -t parameter at the end of the command line. For example: ./verify sha512key.txt signedciphertext.txt verifiedciphertext.txt -t

verify reads the signed file as input, and using the timestamp and HMAC tag header in it, verifies the file to be unaltered. **Note** Verify does not need an hmactype argument (ie, sha256). The HMAC type is carried in the header generated by sign.

verify prints out:
"OK" if the file is has not been tampered with and the current time is within 120 second of the time specified in the timestamp. The 120 second was arrived at to allow the grader sufficient time between the invocation of sing and the subsequent invocation of verify to check that the timestamp validation really works. **Note**: If **"OK"** is printed, then the contents of file.txt used in the sign program and the verifiedfile.txt output of the verify program are guaranteed to be identical.

"TIMESTAMP OUT OF TOLERANCE" if the difference between the timestamp and the current time is more than 120 seconds. The file should be considered as a replay of one previously sent and its contexts should not be utilized.

"TAG MISMATCH" if the calculated tag does not agree with the one in the file. If this message is output, the contents of the verifiedfile.txt argument is suspect, indeterminate, and should not be used.

Note as in the case of sign, the same **openssl** routines were used to generate the HMAC tag.

Also note that if the key specified in the command line has a size that is less than or greater than that expected for the hmactype specified in the header, then the openssl routines will pad it, reduce it to the expected size, or hash it, depending on the implementation of the openssl HMAC scheme invoked.

Note that the code used for generating the HMAC tag for comparison against that in the header is identical to that generating the tag in the sign program, hence the timing of these routines, if specified by the user, should be essentially the same. As in sign, Timing statistics are determined solely by the amount of time the openssl routines take to generate the HMAC. I/O is not taken into account. A 32K buffer is used to read the signed file contents and a maximum 32k byte chunk is utilized in the calls to the openssl HMAC routines. Timing is obtained by calls to the builtin C timing libraries 'clock()' function. The clock is started immediately preceding the call to the openssl HMAC procedures and is stopped on its return. The resulting difference in time is accumulated by adding it to a variable that was zeroed on program startup. This process continues until all data for generating the HMAC tag is exhausted (the complete signedfile,txt input contents and the timestamp have been utilized as input to the openssl HMAC tag generation code).

Note that no output goes to stdout. Only file input and output is utilized.

3. **MAJOR ASSUMPTIONS:**

   It is assumed that the signed file output by sign will be transmitted via some method to some ultimate destination via a communication medium. It is also assumed that both the transmitting and receiving computers have a mechanism for ensuring timing synchronization within some prescribed value. If this is not the case, then the effectiveness of the timestamp would be suspect.

   The encrypt and decrypt routines utilized in part 2 are those in part 1. Since their timing is independent of that of sign and verify, the timing information obtained for encrypt and decrypt is unchanged and there is no reason to perform timing studies on them here. They are utilized in the test sequences, however, to illustrate how a plaintext file is encrypted, signed, transmitted (assumed), received(assumed), verified, and decrypted.

4. Sequence of operations to execute the programs:

   A readme.txt file is included in the code submitted with instructions concerning program execution.

IMPORTANT INFORMATION -a shell file, test.sh has been included and goes through program execution step by step and dumps the contents of the files to the screen during the execution process. Invoke this shell : ./test.sh to see an execution example using the provided plaintext.txt test file.

A make file is include that will build the programs. To build them, employ the following sequence:

```
make clean

make
```

As previously stated, a test input file, plaintext.txt has been provided. The programs can be invoked as follows to generate a typical execution sequence using sha512 and this file:

```
./keygen aes128 > aes128key.txt

./keygen sha512 > sha512key.txt

./encrypt aes128key.txt plaintext.txt > ciphertext.txt

./sign sha512key.txt sha512 ciphertext.txt signedciphertext.txt

the file would be assumed to be transmitted and received here

./verify sha512key.txt signedciphertext.txt verifiedciphertext.txt

./decrypt aes128key.txt verifiedciphertext.txt > plaintextprime.txt
```

This sequence assumes that verify is invoked within 120 seconds of the execution of sign, and that no corruption of the signedciphertxt.txt file has occurred, hence "OK" has been printed out by verify. At this point, a diff between plaintext and plaintextprime.txt would indicate the files are identical.

Another test plaintext input file , test_16.hex.txt has been included to be employed by the grader, if desired, for texting purposes.

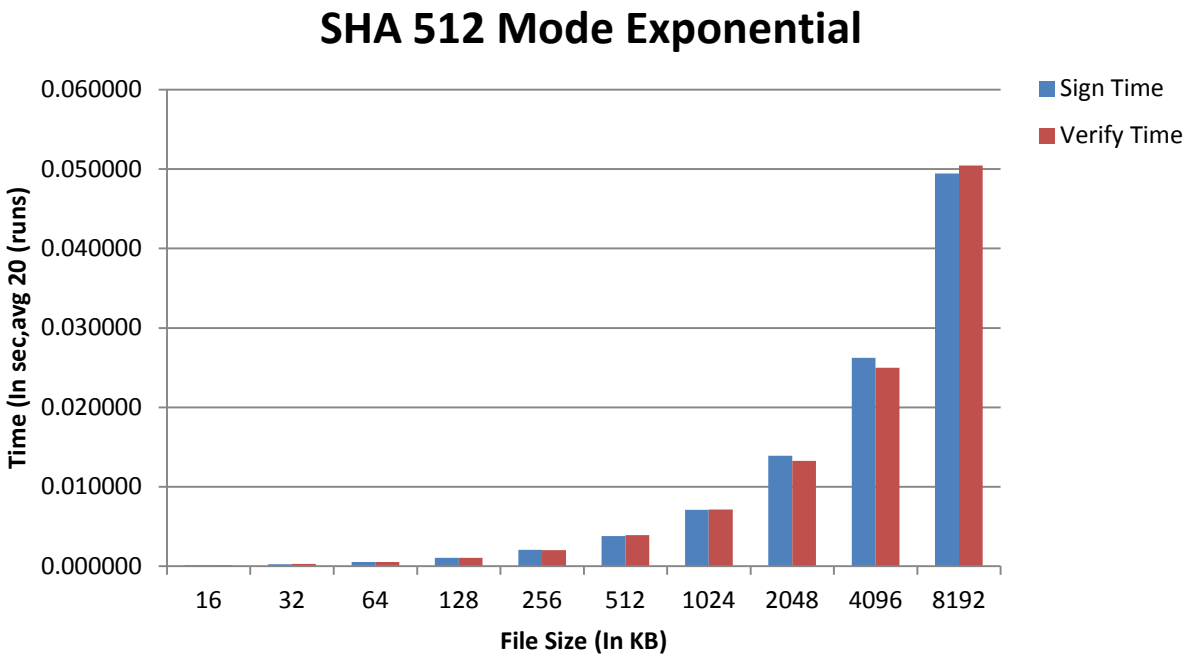
5. Timing studies for sign and verify were accomplished for all 6 HMAC tag generation options. Since sign and verify use the same code and timing sequences to generate and verify the HMAC tag, their values are expected to be the same in each case. In practice, this is not the case, as other tasks are running on the computer that may interfere with the timing studies. The machine was made as quiescent as possible by turning off wifi and all other unessential tasks.

The collected data indicates that there is some deviation due to the reason just stated, however, the effect is really not that profound.

6. The timing studies follow. In each set, the first table and graph indicate HMAC mode performance using exponentially growing file sizes, while the second table and graph measures performance using linearly growing file sizes. Twenty runs of each type were executed and time averaged. Based the performance data collected, we conclude that all the HMAC generation schemes run in linear time in the size of the file read in, that is O(n) where n is the number of bytes to be signed or verified.

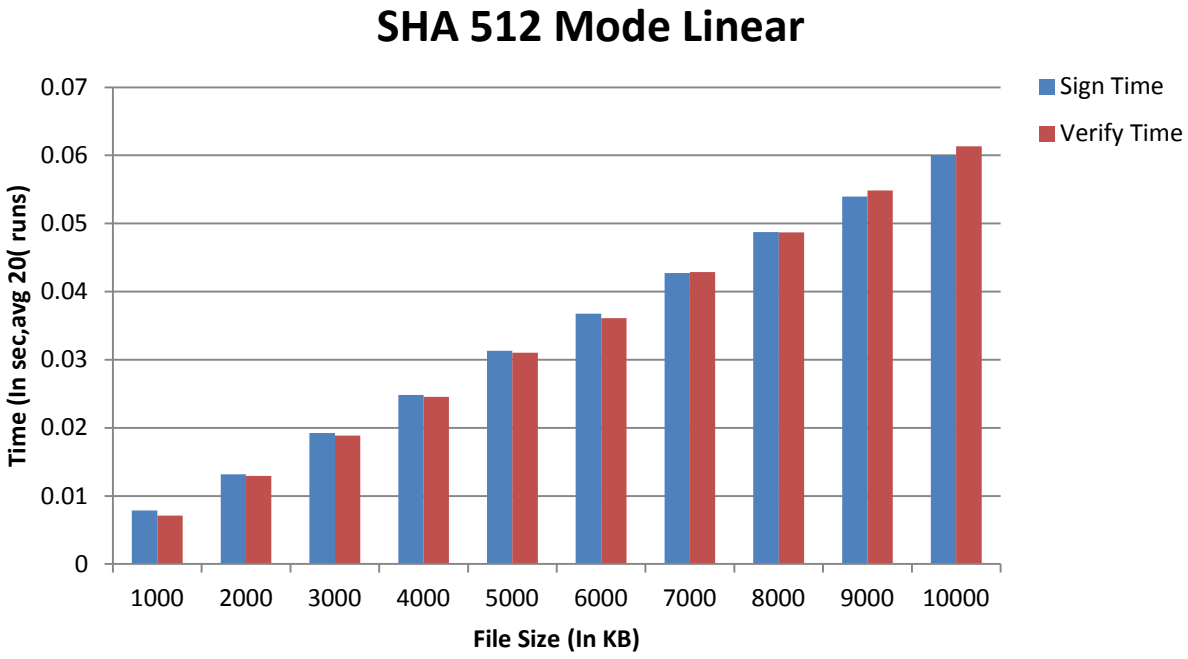
SHA512 - Exponentially Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
16	0.000140	0.000143	0.000132	0.000138
32	0.000272	0.000281	0.000269	0.000274
64	0.000541	0.000555	0.000525	0.000514
128	0.001066	0.001069	0.000985	0.000947
256	0.002051	0.002016	0.001752	0.001914
512	0.003803	0.003930	0.003315	0.003203
1024	0.007118	0.007133	0.006809	0.006135
2048	0.013927	0.013268	0.012290	0.011707
4096	0.026217	0.024975	0.023222	0.025456
8192	0.049439	0.050431		



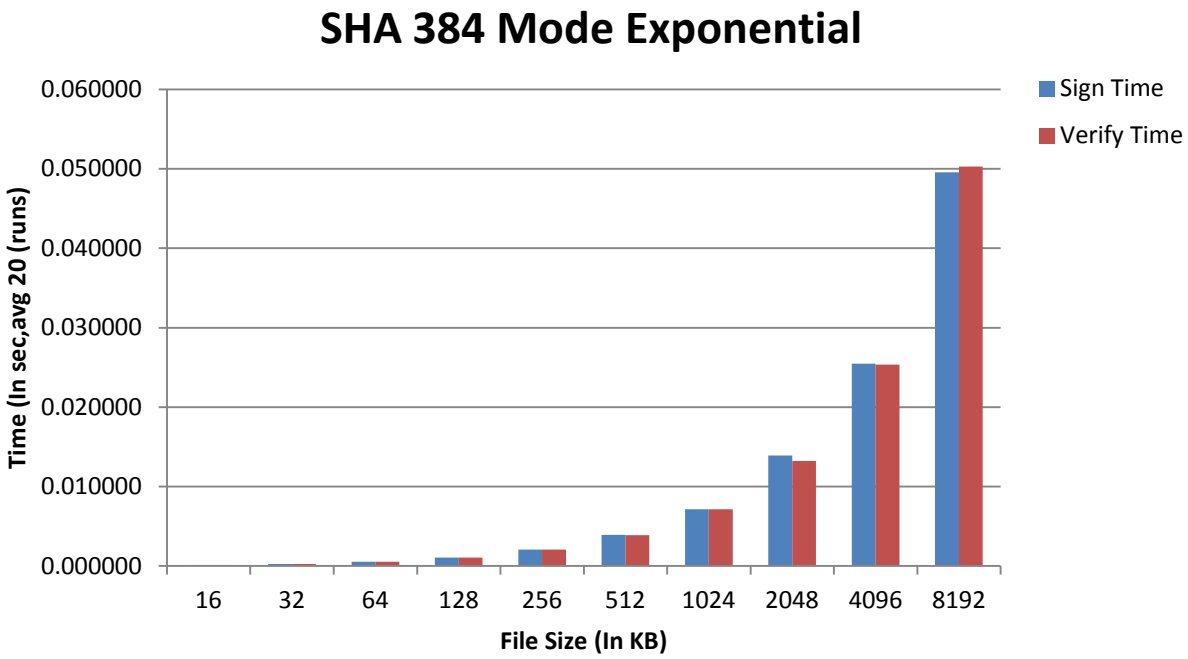
SHA512 - Linearly Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
1000	0.007847	0.007132	0.005326	0.005809
2000	0.013173	0.012941	0.006041	0.005940
3000	0.019214	0.018881	0.005599	0.005659
4000	0.024813	0.024540	0.006481	0.006493
5000	0.031294	0.031033	0.005477	0.005067
6000	0.036771	0.036100	0.005940	0.006748
7000	0.042711	0.042848	0.006040	0.005841
8000	0.048751	0.048689	0.005193	0.006167
9000	0.053944	0.054856	0.006094	0.006504
10000	0.060038	0.061360		



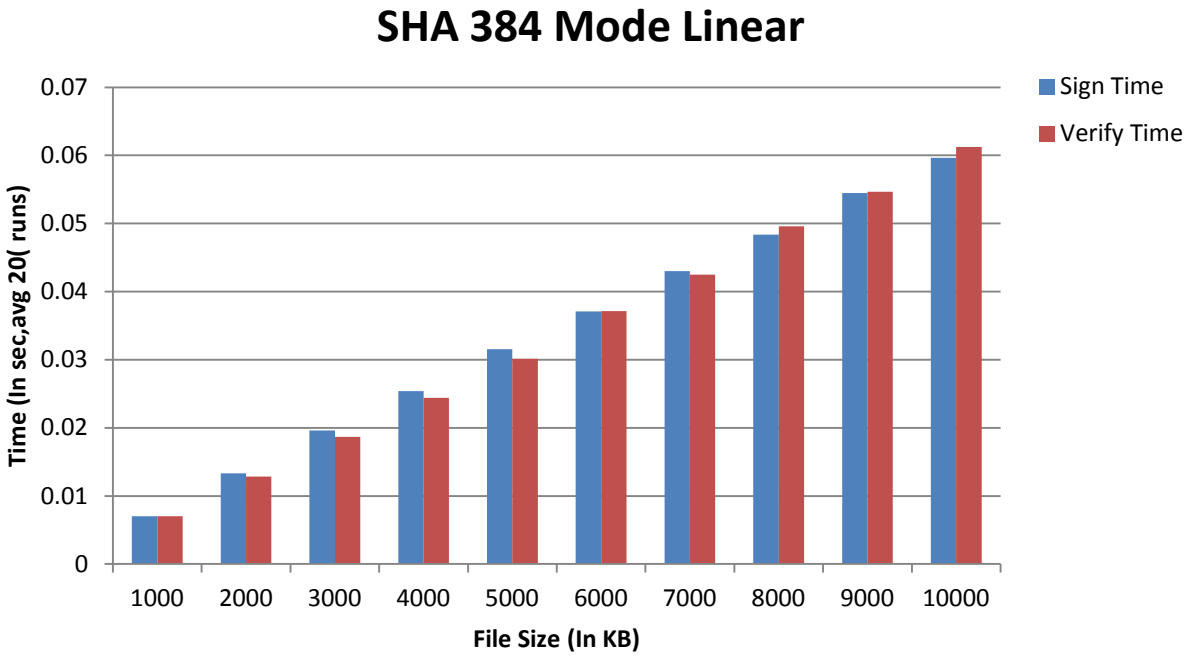
SHA384 - Exponentially Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
16	0.000138	0.000139	0.000133	0.000134
32	0.000271	0.000273	0.000264	0.000268
64	0.000535	0.000541	0.000531	0.000536
128	0.001066	0.001077	0.000985	0.000976
256	0.002051	0.002053	0.001854	0.001826
512	0.003905	0.003879	0.003258	0.003255
1024	0.007163	0.007134	0.006763	0.006091
2048	0.013926	0.013225	0.011544	0.012105
4096	0.025470	0.025330	0.024067	0.024935
8192	0.049537	0.050265		



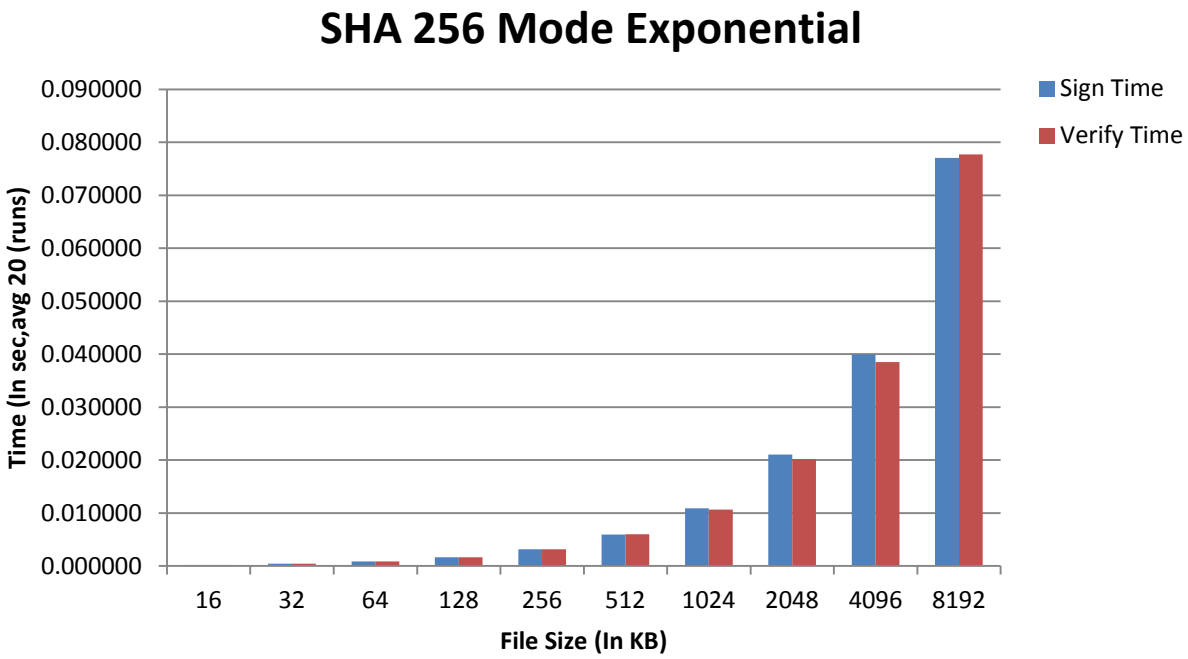
SHA384 - Linearly Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
1000	0.007036	0.007031	0.006264	0.005822
2000	0.013300	0.012853	0.006311	0.005830
3000	0.019611	0.018683	0.005775	0.005728
4000	0.025386	0.024411	0.006157	0.005718
5000	0.031543	0.030129	0.005551	0.006996
6000	0.037094	0.037125	0.005915	0.005366
7000	0.043009	0.042491	0.005342	0.007091
8000	0.048351	0.049582	0.006127	0.005065
9000	0.054478	0.054647	0.005183	0.006591
10000	0.059661	0.061238		



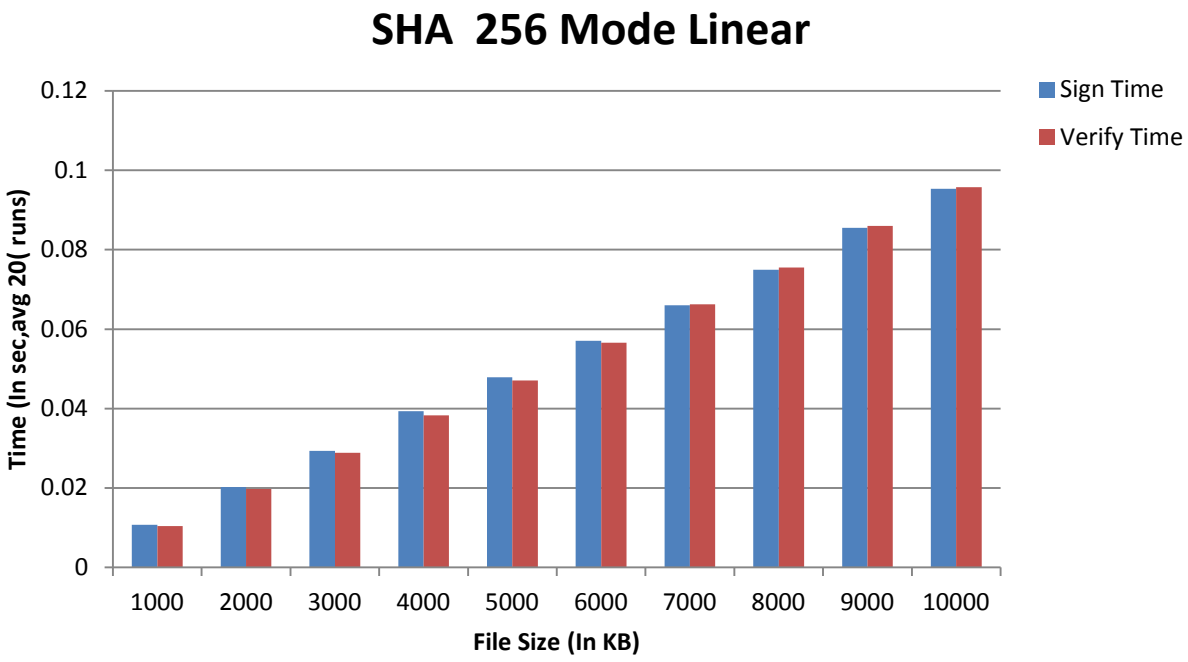
SHA256 - Exponentially Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
16	0.000215	0.000222	0.000212	0.000205
32	0.000427	0.000427	0.000419	0.008033
64	0.000846	0.000846	0.000782	-0.006814
128	0.001628	0.001646	0.001565	0.001521
256	0.003193	0.003167	0.002720	0.002849
512	0.005913	0.006016	0.005013	0.004656
1024	0.010926	0.010672	0.010120	0.009549
2048	0.021046	0.020221	0.018916	0.018290
4096	0.039962	0.038511	0.037091	0.039209
8192	0.077053	0.077720		



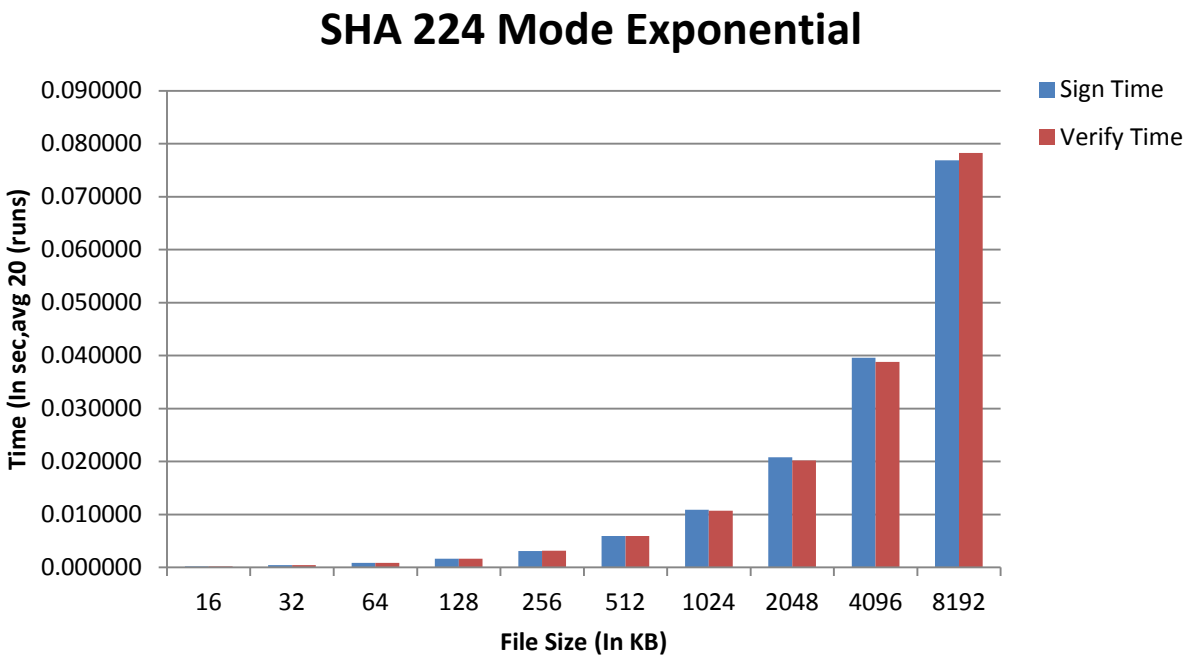
SHA256 - Linearly Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
1000	0.010737	0.010455	0.009494	0.009322
2000	0.020231	0.019777	0.009156	0.009056
3000	0.029387	0.028833	0.009919	0.009464
4000	0.039306	0.038297	0.008560	0.008754
5000	0.047866	0.047051	0.009157	0.009554
6000	0.057023	0.056605	0.008967	0.009619
7000	0.065990	0.066224	0.008971	0.009290
8000	0.074961	0.075514	0.010547	0.010504
9000	0.085508	0.086018	0.009844	0.009738
10000	0.095352	0.095756		



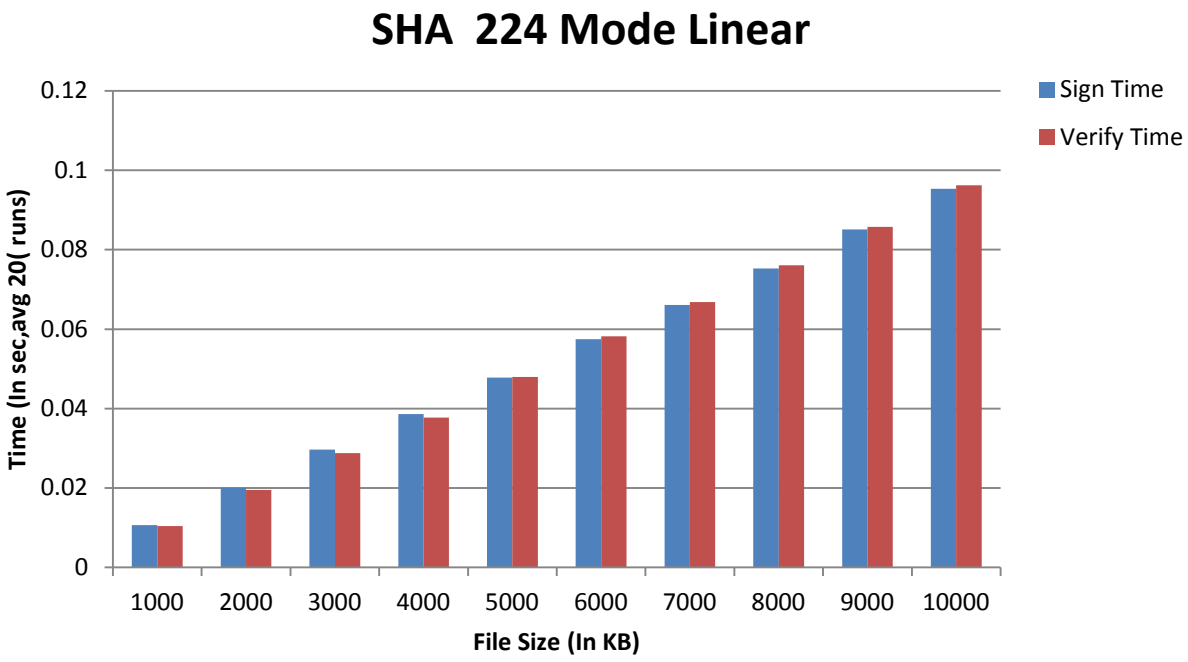
SHA224 - Exponentially Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
16	0.000213	0.000213	0.000207	0.000207
32	0.000420	0.000420	0.000422	0.000427
64	0.000842	0.000847	0.000812	0.000800
128	0.001654	0.001647	0.001474	0.001522
256	0.003128	0.003169	0.002833	0.002787
512	0.005961	0.005956	0.004926	0.004780
1024	0.010887	0.010736	0.009929	0.009490
2048	0.020816	0.020226	0.018767	0.018580
4096	0.039583	0.038806	0.037315	0.039452
8192	0.076898	0.078258		



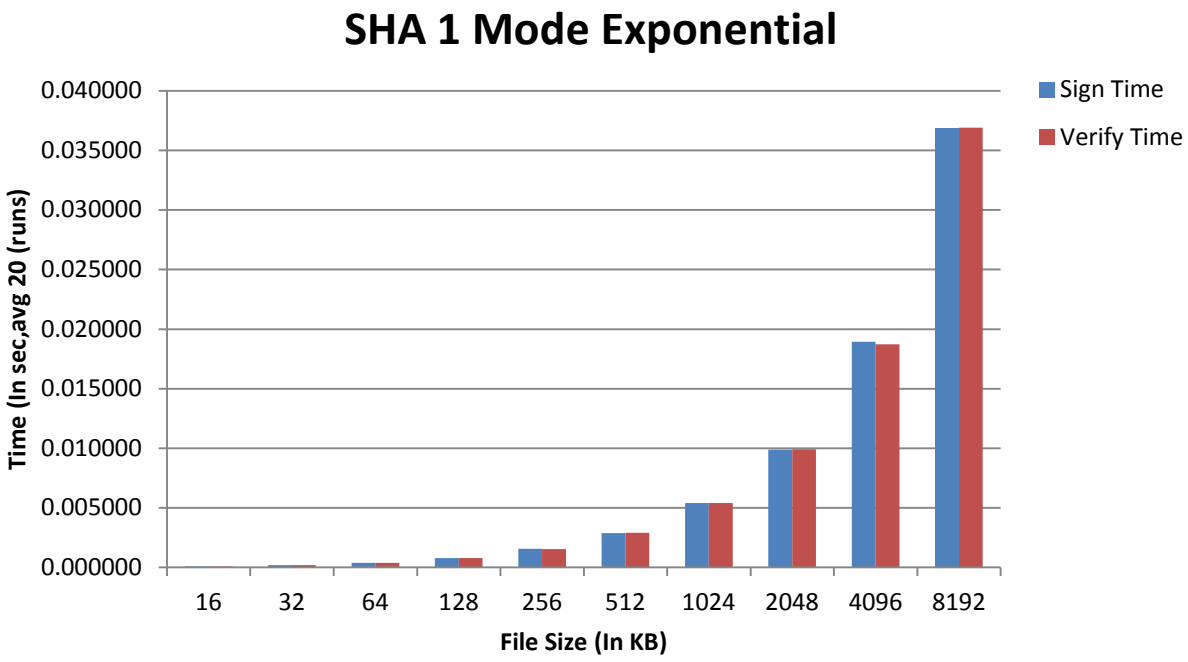
SHA224 - Linearly Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
1000	0.010681	0.010433	0.009432	0.009056
2000	0.020113	0.019489	0.009569	0.009305
3000	0.029682	0.028794	0.008915	0.008960
4000	0.038597	0.037754	0.009173	0.010199
5000	0.047770	0.047953	0.009673	0.010239
6000	0.057443	0.058192	0.008607	0.008618
7000	0.066050	0.066810	0.009192	0.009246
8000	0.075242	0.076056	0.009888	0.009691
9000	0.085130	0.085747	0.010230	0.010454
10000	0.095360	0.096201		



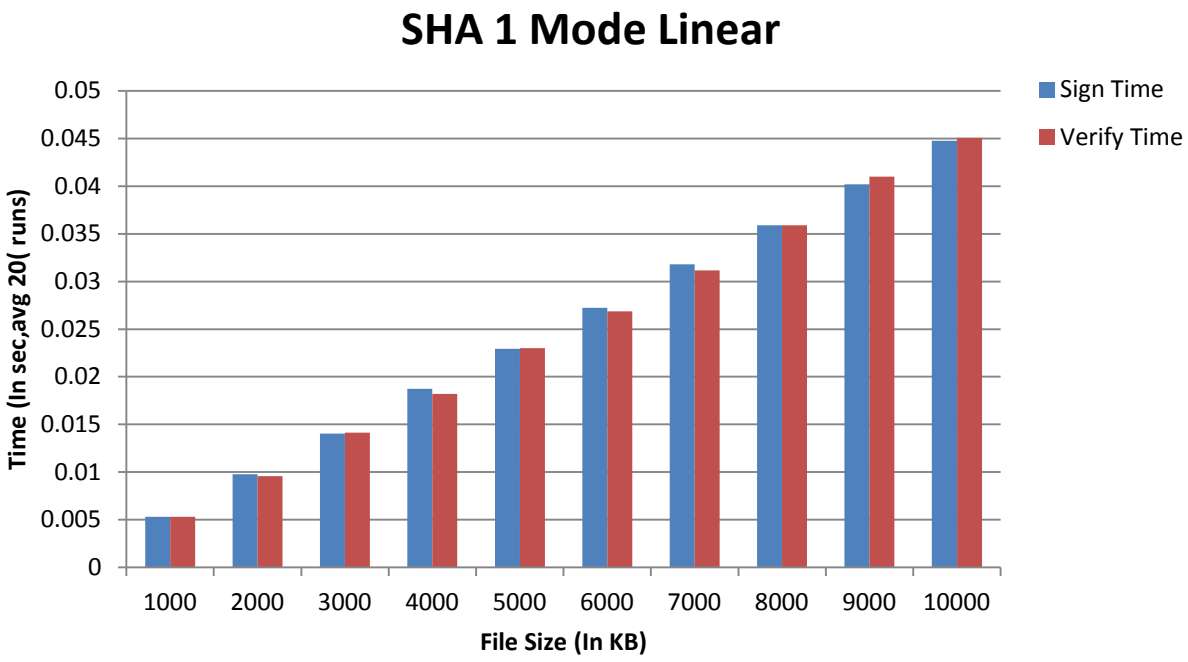
SHA1 - Exponentially Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
16	0.000100	0.000102	0.000101	0.000095
32	0.000201	0.000197	0.000196	0.000199
64	0.000397	0.000396	0.000392	0.000392
128	0.000789	0.000788	0.000770	0.000739
256	0.001559	0.001527	0.001326	0.001396
512	0.002885	0.002923	0.002525	0.002484
1024	0.005410	0.005407	0.004481	0.004511
2048	0.009891	0.009918	0.009039	0.008799
4096	0.018930	0.018717	0.017933	0.018173
8192	0.036863	0.036890		



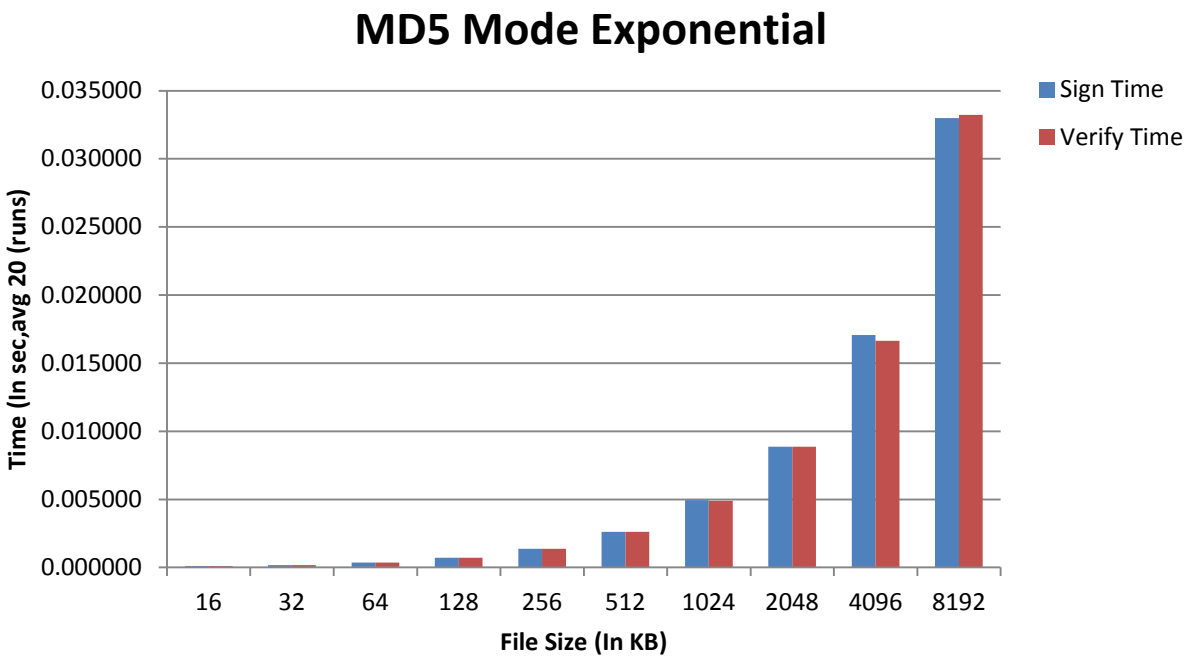
SHA1 - Linearly Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
1000	0.005327	0.005316	0.004457	0.004267
2000	0.009784	0.009583	0.004272	0.004544
3000	0.014056	0.014127	0.004671	0.004073
4000	0.018727	0.018200	0.004195	0.004806
5000	0.022922	0.023006	0.004302	0.003847
6000	0.027224	0.026853	0.004587	0.004304
7000	0.031811	0.031157	0.004087	0.004736
8000	0.035898	0.035893	0.004296	0.005094
9000	0.040194	0.040987	0.004542	0.004074
10000	0.044736	0.045061		



MD5 - Exponentially Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
16	0.000092	0.000092	0.000088	0.000087
32	0.000180	0.000179	0.000175	0.000176
64	0.000355	0.000355	0.000348	0.000363
128	0.000703	0.000718	0.000666	0.000657
256	0.001369	0.001375	0.001236	0.001253
512	0.002605	0.002628	0.002340	0.002272
1024	0.004945	0.004900	0.003927	0.003955
2048	0.008872	0.008855	0.008188	0.007799
4096	0.017060	0.016654	0.015928	0.016584
8192	0.032988	0.033238	-0.032988	-0.033238



MD5 - Linearly Growing File Size

File Size (in KB)	Sign Time	Verify Time	Ratio of Growth Sign	Ratio of Growth Verify
1000	0.004824	0.004818	0.003860	0.003913
2000	0.008684	0.008731	0.004115	0.003786
3000	0.012799	0.012517	0.004130	0.003841
4000	0.016929	0.016358	0.003782	0.003967
5000	0.020711	0.020325	0.003715	0.004056
6000	0.024426	0.024381	0.004265	0.003947
7000	0.028691	0.028328	0.003517	0.004009
8000	0.032208	0.032337	0.003606	0.004495
9000	0.035814	0.036832	0.003883	0.004501
10000	0.039697	0.041333