An Analysis of the AES-NI Extension to the x86 Instruction Set

Shawn D. Wilkinson

***Abstract*—The AES-NI extension to the x86 instruction set used by Intel and AMD microprocessors greatly enhances the performance of cryptographic operations relying on the Advanced Encryption Standard (AES). Cipher modes of operation than can be run in parallel see improvements in encryption time of 600% to 1400%. Cipher modes of operation than do not run in parallel see a 200% to 300% improvement in encryption time. Other ciphers are using elements of AES supported by the AES-NI instruction set to realize performance gains from their algorithms.**

I. Purpose of the AES-NI Extension

The Advanced Encryption Standard (AES) was adopted by the U.S. government as a federal encryption standard on May 26, 2002, following final approval by the Secretary of Commerce. The algorithm, based on the Rijndael cipher created by Joan Daemen and Vincent Rijmen, was selected by the U.S. National Institute of Standards and Technology (NIST) after a five year selection process designed to replace the aging and less secure Data Encryption Standard (DES). The AES cipher, defined in FIPS PUB 197 and ISO/IEC 18033-3, has become one of the most implemented ciphers in use today. As AES has become more and more widespread, CPU manufacturers recognized the need to implement acceleration methods for AES operations inside the CPUs themselves. Intel proposed the development of the AES-NI instruction set extension in 2008, and after collaboration with AMD, both companies released CPUs able to take advantage of these instructions in 2010. The instruction set extension allows programmers to utilize hardware more efficiently when performing AES related cryptographic operations, rather than rely on strictly software implementations performed by a general purpose processor.

AES is used in numerous applications, and since the release of AES-NI, most of these applications have been updated to take advantage of the new instructions. AES-NI enabled programs perform whole disk encryption, data authentication, random number generation, implement authenticated ciphers, and many other tasks.

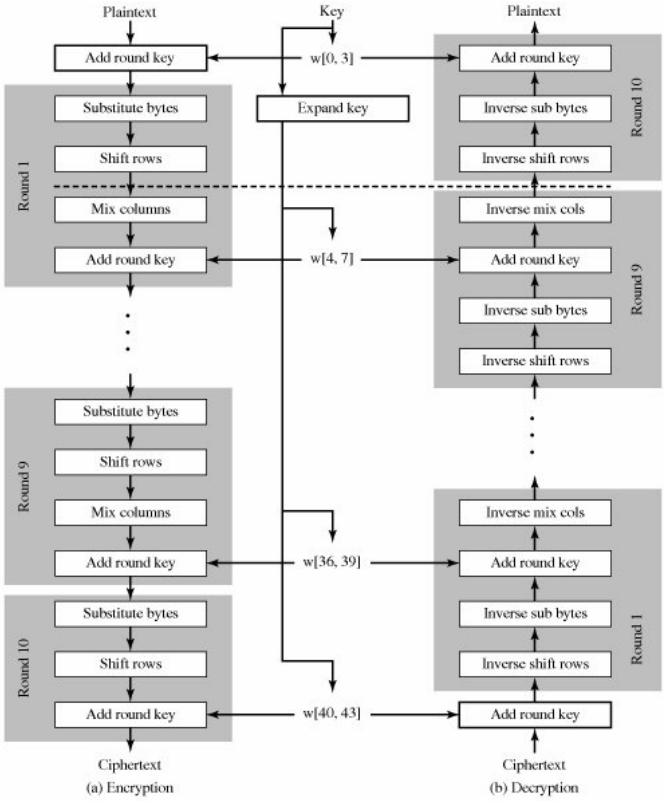
II. Motivation

There have been numerous benchmarks conducted on the AES-NI extensions. Many of the early performance metrics were performed when AES-NI capable CPUs first entered the market in 2010 and 2011. I found during my research that many studies were not performed correctly, such as using one version of a cryptographic library to benchmark AES-NI, and then using another version to benchmark the same operations without the benefit of AES-NI. In some cases, an entirely different CPU was used for comparison purposes. I conducted the tests detailed in this paper using the same algorithms on the same hardware and believe the results are as accurate as possible.

III. AES Fundamentals

AES is approved using three different key sizes: 128 bit, 192 bit, and 256 bit. The specification also states that blocks of data to be encrypted will be 128 bits, although the original Rijndael algorithm does allow for any multiple of 32 bits to be used as a block size. The various key sizes change the algorithm with regards to how many rounds of operations must be performed on the data to encrypt it. A key size of 128 bits require 10 rounds of the AES transformations, while a 192 bit key requires 12 rounds and a 256 bit key requires 14 rounds. The following diagram illustrates how AES encrypts and decrypts data.

This paper was produced to meet the project requirement for ECE646 at George Mason University. This paper was produced in December, 2015.



The first step in encrypting or decrypting data is to perform a key expansion. This process breaks the n bit key (128, 192, or 256 bits) into a 16 byte matrix of 4 rows and 4 columns for each round of transformations (10, 12, or 14 rounds dependent on key size). This matrix is called a round key. Typically, all round keys required for the entire operation will be computed at once, and their values will be stored in lookup tables. The second step in the algorithm is called the "Substitute bytes" transformation. This process involves using an S-box (substitution box) to replace values on the input text with known values from a lookup table. The third step is called the "Shift rows" transformation, and the fourth step is called the "Mix columns" transformation. These transformations change the position of the data in a register as it is being processed. The final step performed in each round is the "Add round key" transformation. This process XORs the round key with the data output from the previous steps. Each step, excluding round key expansion (assuming all round keys were initially calculated and stored), is then performed again for as many rounds as necessary.

IV. The New Instructions

Six instructions were added to the x86 instruction set as part of the AES-NI extension. AESENC performs one round of an AES encryption flow. AESENCLAST performs the last round of an AES encryption flow. AESDEC performs one round of an AES decryption flow. AESDECLAST performs the last round of an AES decryption flow. AESKEYGENASSIST assists in AES round key generation. AESIMC assists in AES Inverse Mix Columns. In addition to those commands, a seventh instruction, used heavily by the AES-NI process, is PCLMULQDQ which performs a carryless multiply.

The format for using the commands include specifying the register where the data input is and the register where the key input is. For example, the command AESENC xmm1, xmm2/m128, tells the processor where to retrieve the initial text input and the round key, and then after completing the round of AES encryption stores the result where the initial text input was stored. The instructions perform the following tasks within the processor.

```
AESENC xmm1, xmm2/m128
Tmp := xmm1
RoundKey :=xmm2/m128
Tmp := ShiftRows (Tmp)
Tmp := SubBytes (Tmp)
Tmp := MixColumns (Tmp)
xmm1:= Tmp xor RoundKey

AESENCLAST xmm1, xmm2/m128
Tmp := xmm1
RoundKey :=xmm2/m128
Tmp := ShiftRows (Tmp)
Tmp := SubBytes (Tmp)
xmm1:= Tmp xor RoundKey

AESDEC xmm1, xmm2/m128
Tmp := xmm1
RoundKey :=xmm2/m128
Tmp := InvShiftRows (Tmp)
Tmp := InvSubBytes (Tmp)
Tmp := InvMixColumns (Tmp)
xmm1:= Tmp xor RoundKey

AESDECLAST xmm1, xmm2/m128
Tmp := xmm1
RoundKey :=xmm2/m128
   Tmp := InvShiftRows (Tmp)
Tmp := InvSubBytes (Tmp)
xmm1:= Tmp xor RoundKey

AESKEYGENASSIST xmm1, xmm2/m128, imm8
Tmp := xmm2/m128
RCON[31-8] := 0; RCON[7-0] := imm8;
X3[31-0] := Tmp[127-96]; X2[31-0] := Tmp[95-64];
X1[31-0] := Tmp[63-32]; X0[31-0] := Tmp[31-0];
xmm1 := [RotWord (SubWord (X3)) XOR RCON, SubWord
(X3), RotWord (SubWord (X1)) XOR RCON, SubWord (X1)]

AESIMC xmm1, xmm2/m128
RoundKey := xmm2/m128;
xmm1 := InvMixColumns (RoundKey)
```

AES-NI allows the programmer to type a single command for each round of encryption, instead of having to manually code each step in the AES process. These commands save programmers a great deal of effort, in addition to providing performance benefits due to the AES operations taking place

all within the CPU. The bottleneck with an all software implementation of AES is having to continuously read and write lookup tables to memory.

V. Security benefits

One advantage to using AES-NI is the mitigation of side-channel attacks. One side-channel attack that has been demonstrated to work takes advantage of the lookup table used for key expansion being stored in memory. An attacker can create a rogue program that runs alongside the AES service. It continuously writes data to memory, filling up the cache. As the AES service needs to perform a key expansion, it re-writes the key data to memory. The rogue program, over time, can map what specific locations in memory the AES service is writing to and be able to decipher the key. AES-NI does not use lookup tables stored in memory for key expansion. All key expansion operations happen in the CPU without lookup tables, mitigating this attack.

VI. Power consumption

Using fewer CPU cycles to complete an operation results in less power consumption. A typical home user may not notice these effects, however looking at cases involving much larger computers or much smaller computers yields tangible benefits. A server farm or large website host may be able to calculate actual cost savings from requiring less electricity to perform the same tasks. A mobile device user on a battery may be able to see longer battery life due to less processing required.

VII. Testing platform

The first testing platform used was a desktop computer equipped with an Intel i7-4770K processor running at 3.5 GHz. The processor was not overclocked and hyper-threading was disabled in the system BIOS. This processor supports the AES-NI instruction set. This processor contains four physical cores.

The second testing platform used was a desktop computer equipped with an Intel i5-4670K processor running at 3.4 GHz. The processor was not overclocked and hyper-threading was disabled in the system BIOS. This processor supports the AES-NI instruction set. This processor contains four physical cores.

The third testing platform used was a laptop computer equipped with an Intel i5-3317U processor running at 1.7 GHz. The processor was not overclocked and hyper-threading was disabled in the system BIOS. This processor supports the AES-NI instruction set. This processor contains two physical cores.

Processor	Frequency	# of Physical Cores	Hyper-threading	AES-NI Capable
Intel i7-4770K	3.5 GHz	4	Disabled	Yes
Intel i5-4670K	3.4 GHz	4	Disabled	Yes
Intel i5-3317U	1.7 GHz	2	Disabled	Yes

The exact same operating system and cryptographic libraries were used during final testing on each platform. In an effort to have a software platform that can easily be reproduced for further research, the following software configuration was used. I performed a clean installation of Ubuntu Linux 14.04 LTS (Long Term Support indicating it will be supported for at least five years). I downloaded the cryptographic library OpenSSL version 1.0.2e (released on December 2, 2015). I extracted the file 'openssl-1.0.2.e.tar.gz' to the folder 'openssl-1.0.2e' in the home directory (an earlier version of OpenSSL comes pre-installed with most Linux distributions, and I wanted to segregate the two to avoid confusion). In the terminal, after navigating to the directory you extracted OpenSSL to, I typed the command "./config && make".

Operating System	Crypto Library
Ubuntu 14.04 LTS	OpenSSL 1.0.2e

One of the reasons that OpenSSL was selected for this testing process was that it has an ability to benchmark cryptographic algorithms in a variety of ways. I was able to select the algorithm to be used and the key size for each test, and was able to select a duration of time during which to complete as many encryptions as possible on different size files.

I performed tests on data sets that were 16, 64, 128, 256, 1024, and 8192 bytes in length. The data length used for most of the graphs in this paper was 8192 bytes. This length of data allowed the processor to spend more time performing transformations enabled by AES-NI and less time performing overhead operations.

The command used for testing was openssl speed –elapsed –evp algorithm. Openssl is the name of the executable file. The speed option indicates that it should encode blocks of data as many times as possible within a certain time limit (3 seconds is the default and the time length that I used). The elapsed option tells the program that everything should be referenced in real time as opposed to CPU time. The evp option tells the program that we want to enable AES-NI instructions once a check is performed to see if the CPU supports them. Algorithm is replaced by the specific cryptographic algorithm requested. For this test, all inputs were in the form 'cipher – key size – mode of operation', such as aes-256-cbc.

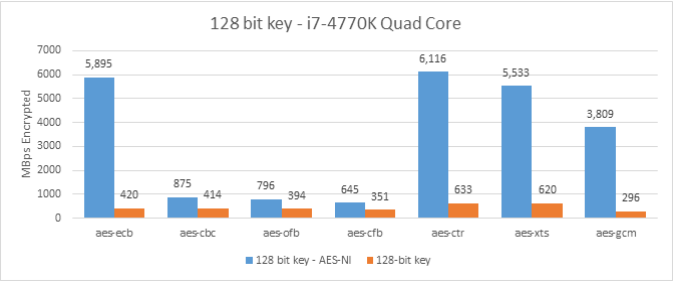
The OpenSSL program will check the processor to see if it supports AES-NI when using the command above. In order to run tests where AES-NI is not used, it is necessary to override the flags set by the operating system when performing this test. In order to inform OpenSSL that the CPU does not support AES-NI (even though it does), it is necessary to preface the openssl command with OPENSSL_ia32cap="~0x200000200000000", which alters the IA-32 processor capabilities vector.

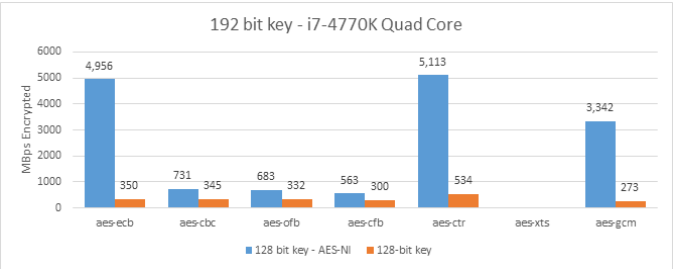
I wrote a script to automate the tests conducted for this paper, and a portion of that script is posted below. The full script is added as an appendix to this paper to aid any future research into this subject.

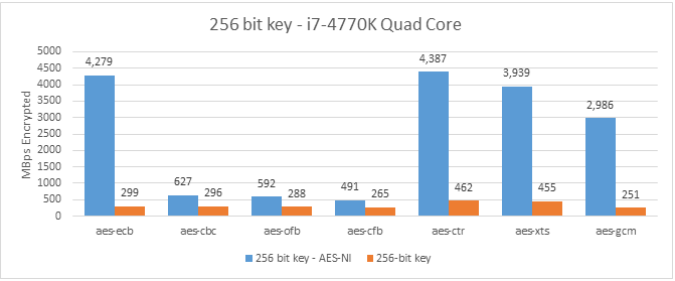
```
echo "------------------------------------------------" > output.txt

echo "128-bit AES operations AES-NI
Enabled" >> output.txt

echo "------------------------------------------------" >>
output.txt

./apps/openssl speed -elapsed -evp aes-128-ecb >> output.txt

sleep 10

./apps/openssl speed -elapsed -evp aes-128-cbc >> output.txt

sleep 10

echo "------------------------------------------------" >>
output.txt

echo "128-bit AES operations AES-NI Disabled" >>
output.txt

echo "------------------------------------------------" >>
output.txt

OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-128-ecb >> output.txt

sleep 10

OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-128-cbc >> output.txt

sleep 10
```

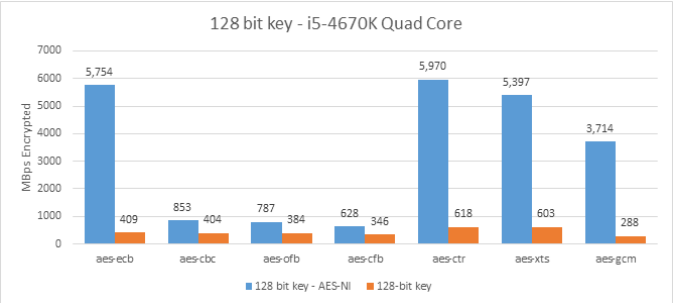
VIII. Performance

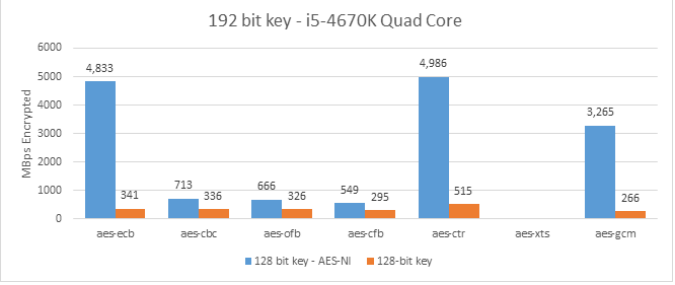
Intel claimed performance gains of two to three times for modes of operation that can't be run in parallel. Intel claimed performance gains of six to eight times for modes of operation that can be run in parallel. The following charts show the performance measured using the testing platforms and commands detailed above. Note that the XTS mode of operation is not supported using a 192-bit key size, and is omitted from the testing.

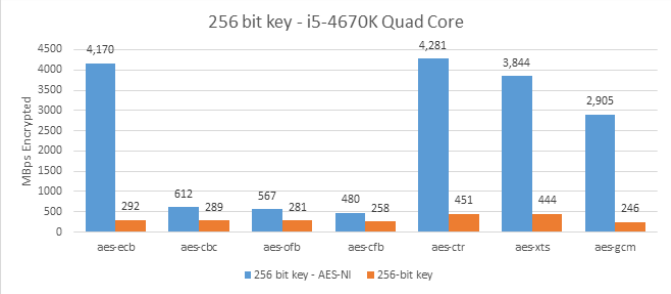


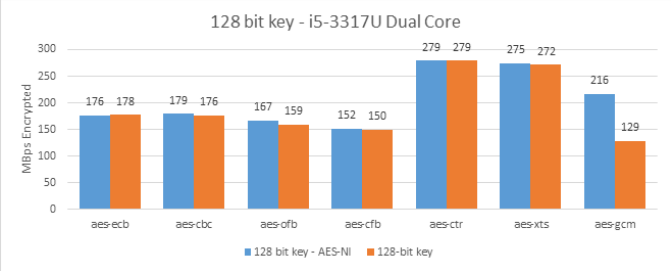


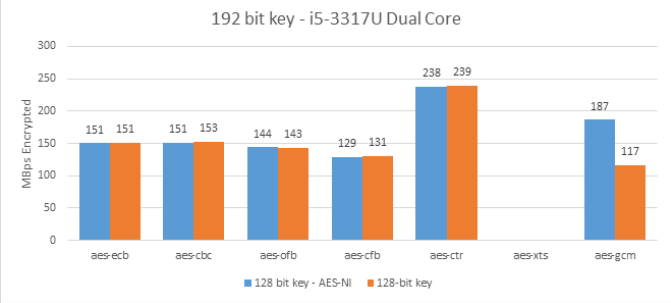


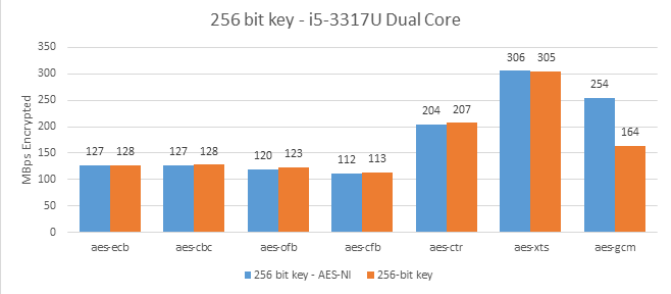












IX. Analysis

Modes of operation that can be run in parallel benefit in performance gains ranging from 7-14x. This is higher than Intel originally claimed, and is likely attributed to significantly faster processors being produced over the last five years. Modes of operation that can't be run in parallel did see performance improvements of 2-3x.

The outlier in my data set is the i5-3317U CPU. I do not understand why it did not show the same performance gains as the other processors, and believe this merits further research. I performed several tests using various libraries and all results were similar.

X. Future Testing

Other algorithms in addition to AES have begun using instructions from AES-NI to improve performance. Due to Intel and AMD processors being saturated throughout the market, other cipher designers know if they can take advantage of certain pieces of AES-NI their cipher will perform better. Ciphers currently using AES-NI in the NIST Hash competition include: LANE, SHAMATA, SHAvite-3, ECHO, GrOstl, Lesamnta, and Vortex. It merits further research to understand how these ciphers are performing against their peers that are not using AES-NI.

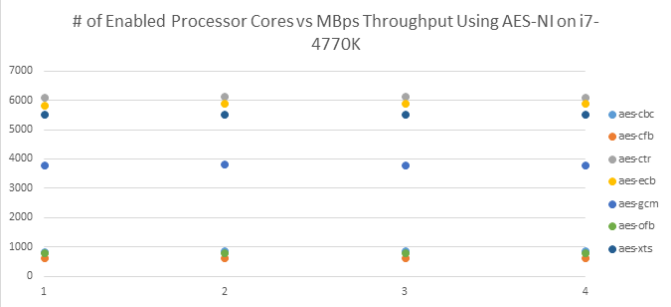
I would like to conduct further research into special microprocessor extensions for other platforms, such as ARM. ARMv8 and some earlier versions include a NEON cryptographic extension that I would like to understand further.

XI. Physical Cores, Pipelining, and Hyperthreading

Hyper-threading provides no benefit towards operations utilizing AES-NI. The processes occurring within the CPU are already optimized in hardware as much as possible. Hyper-threading may enhance a pure software implementation of AES, but is not relevant to this topic. Hyper-threading was disabled for all tests as a precaution.

Pipelining is a large benefit to AES processes, especially those that can be run in parallel. When the block being processed does not require the output from the previous block, the CPU can place commands on every clock cycle to optimize how many blocks are processed simultaneously. Even though a single AES-NI instruction performs all steps for a round of encryption and decryption, it still takes numerous clock cycles to complete a block. Pipelining allows the use of those spare clock cycles for the next block.

Multiple cores enable a single processor to theoretically perform n times more calculations simultaneously than without multiple cores. I attempted to prove this through benchmarking tests, but was unable to provide evidence. Initial guesses as to why I couldn't see these performance gains point towards multiple cores attempting to share the same registers inside the CPU or, more likely, that I was not successfully disabling cores when performing these tests. I attempted to disable cores in my system BIOS on the machine with the i7-4770K processor. I ran tests with one, two, three, and all four cores enabled, and came up with the following results.



As the graphic indicates, all modes of operation using AES-NI resulted in very similar throughput. I believe that this merits further research and could be included in a future expansion of this research paper.

XII. References

[1] AES Instruction Set. (n.d.). *Wikipedia*. Available: https://en.wikipedia.org/wiki/AES_instruction_set. Accessed Sep. 22, 2015.

[2] S. Gueron. (2009). Intel's New AES Instructions for Enhanced Performance and Security. LNCS, vol. 5665, pp. 51-66. Available: https://www.iacr.org/archive/fse2009/56650054/56650054.pdf

[3] J. Bos. (2011) Efficient hashing using the AES instruction set. LNCS, vol. 6917, pp. 507-522. Available: http://eprint.iacr.org/2010/576.pdf

[4] S. Gueron, "AES-GCM software performance on the current high end CPUs as a performance baseline for CAESAR competition," presented at DIAC Conf., Chicago, IL, 2013. Available: http://2013.diac.cr.yp.to/slides/gueron.pdf

[5] R. Benadjila, "Use of the AES instruction set," presented at Ecrypt II Conf., Bruges, Belgium, 2012. Available: https://www.cosic.esat.kuleuven.be/ecrypt/AESday/slides/Use_of_the_AES_Instruction_Set.pdf

[6] How exactly does AES-NI work?. (2014, Oct.). Stack Exchange. [Online]. Available: http://crypto.stackexchange.com/questions/19544/how-exactly-does-aes-ni-work

[7] ECHO Hash function. (n.d.). France Telecom. [Online]. Available: http://maketheconnection.net/conditions/ptsd2. Accessed Sep. 22, 2015.

[8] G. Hofemeier. (2012, Jul. 26). Introduction to Intel AES-NI and Intel Secure Key Instructions. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-aes-ni-and-intel-secure-key-instructions

[9] Some small suggestions for the Intel instruction set. (2014, May 17). The cr.yp.to blog. [Online]. Available: http://blog.cr.yp.to/20140517-insns.html

[10] AES-NI SSL Performance. (2015, Oct 20). Calomel.org [Online]. Available: https://calomel.org/aesni_ssl_performance.html

[10] AES-NI SSL Performance. (2015, Oct 20). Calomel.org [Online]. Available: https://calomel.org/aesni_ssl_performance.html

XIII. Appendix: OpenSSL Script For Testing

```
echo "----------------------------------------------------" > output.txt
echo "128-bit AES operations AES-NI Enabled" >> output.txt
echo "----------------------------------------------------" >>
output.txt
./apps/openssl speed -elapsed -evp aes-128-ecb >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-128-cbc >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-128-ofb >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-128-cfb >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-128-ctr >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-128-xts >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-128-gcm >> output.txt
sleep 10
echo "----------------------------------------------------" >>
output.txt
echo "192-bit AES operations AES-NI Enabled" >> output.txt
echo "----------------------------------------------------" >>
output.txt
./apps/openssl speed -elapsed -evp aes-192-ecb >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-192-cbc >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-192-ofb >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-192-cfb >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-192-ctr >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-192-xts >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-192-gcm >> output.txt
sleep 10
echo "----------------------------------------------------" >>
output.txt
echo "256-bit AES operations AES-NI Enabled" >> output.txt
echo "----------------------------------------------------" >>
output.txt
./apps/openssl speed -elapsed -evp aes-256-ecb >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-256-cbc >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-256-ofb >> output.txt
```

```
sleep 10
./apps/openssl speed -elapsed -evp aes-256-cfb >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-256-ctr >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-256-xts >> output.txt
sleep 10
./apps/openssl speed -elapsed -evp aes-256-gcm >> output.txt
sleep 10
echo "----------------------------------------------------" >>
output.txt
echo "128-bit AES operations AES-NI Disabled" >>
output.txt
echo "----------------------------------------------------" >>
output.txt
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-128-ecb >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-128-cbc >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-128-ofb >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-128-cfb >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-128-ctr >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-128-xts >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-128-gcm >> output.txt
sleep 10
echo "----------------------------------------------------" >>
output.txt
echo "192-bit AES operations AES-NI Disabled" >>
output.txt
echo "----------------------------------------------------" >>
output.txt
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-192-ecb >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-192-cbc >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-192-ofb >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-192-cfb >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-192-ctr >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-192-gcm >> output.txt
sleep 10
echo "----------------------------------------------------" >>
output.txt
echo "256-bit AES operations AES-NI Disabled" >>
output.txt
echo "----------------------------------------------------" >>
output.txt
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-256-ecb >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-256-cbc >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-256-ofb >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-256-cfb >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-256-ctr >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-256-xts >> output.txt
sleep 10
OPENSSL_ia32cap="~0x200000200000000" ./apps/openssl
speed -elapsed -evp aes-256-gcm >> output.txt
sleep 10
```