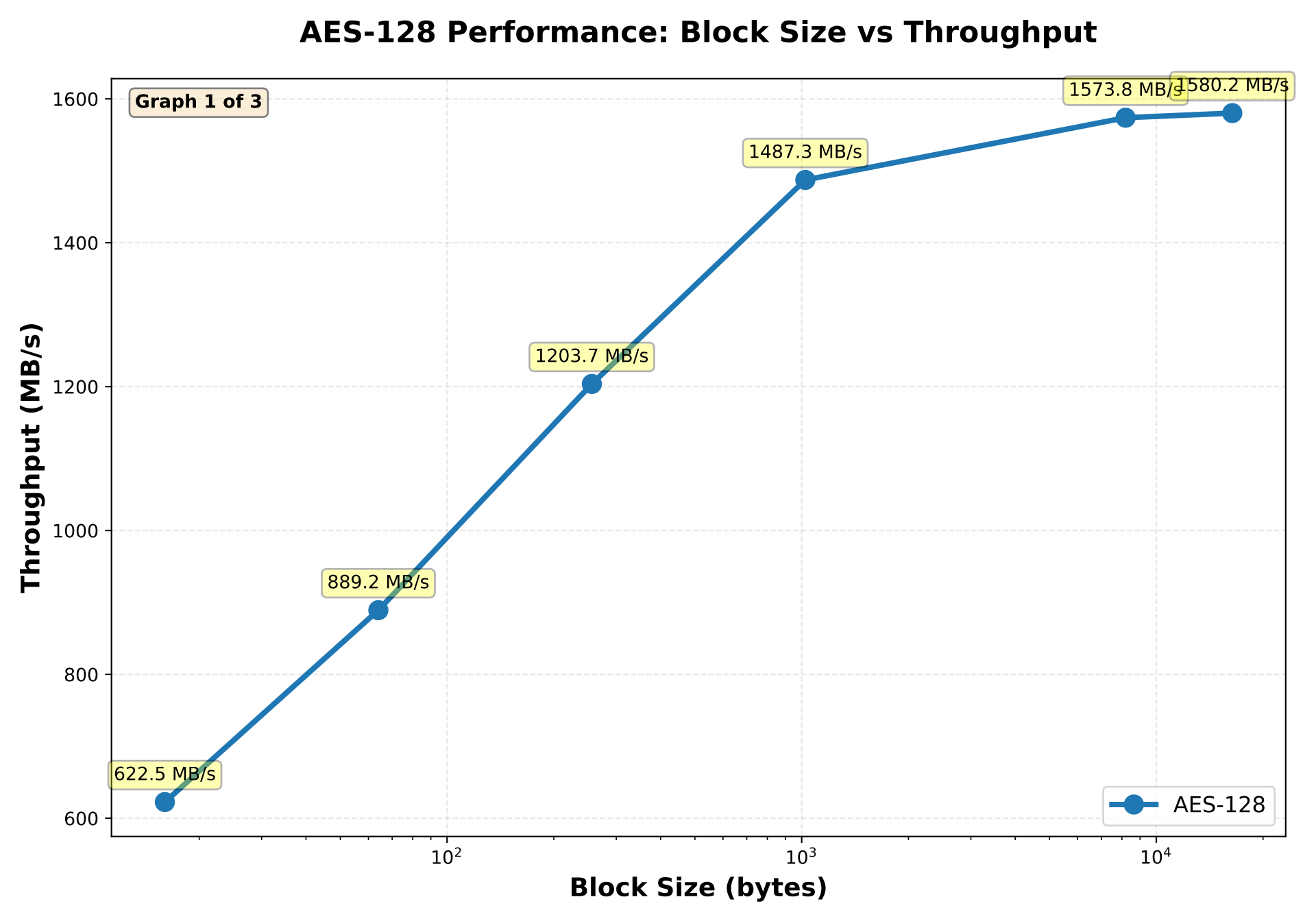
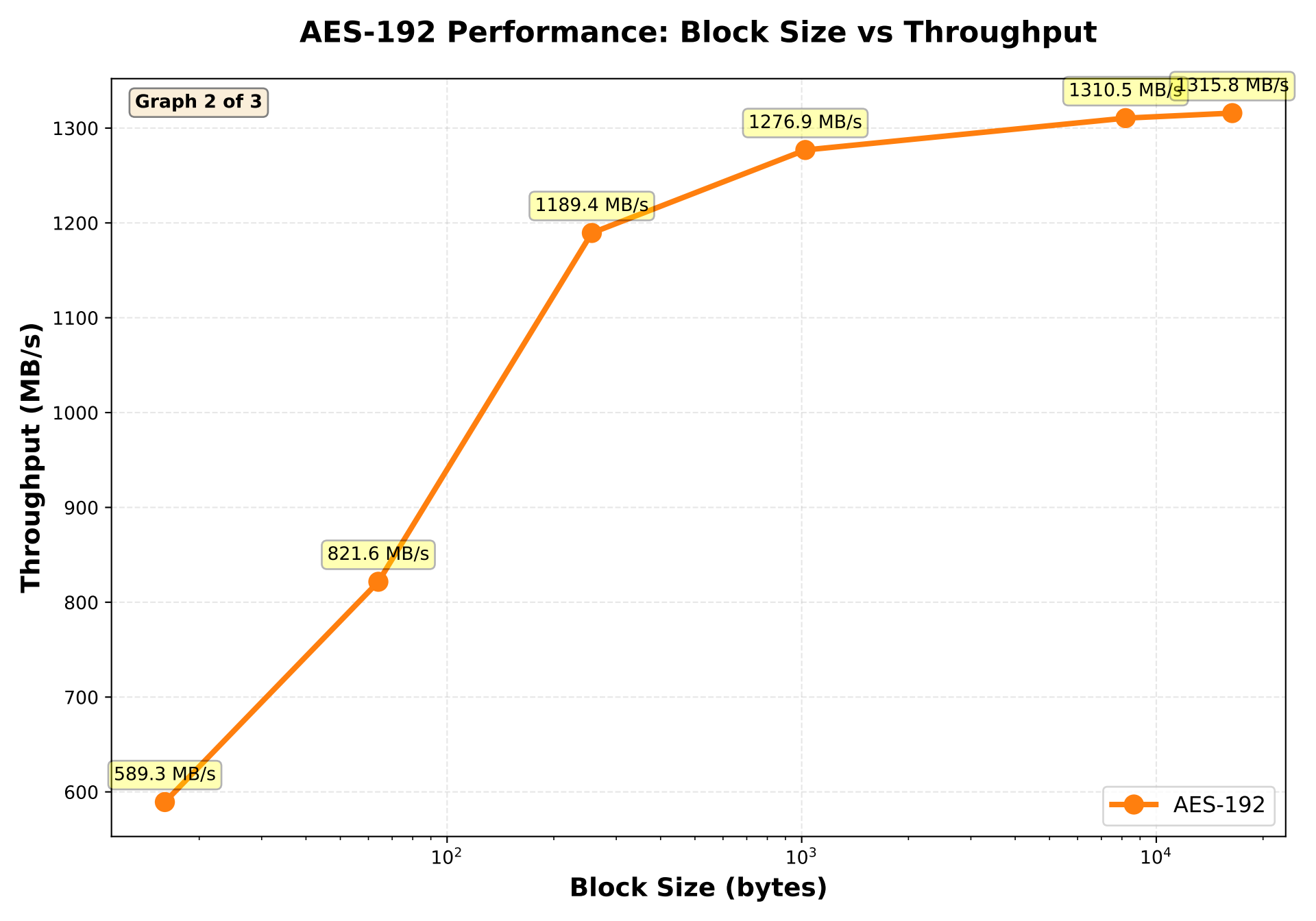
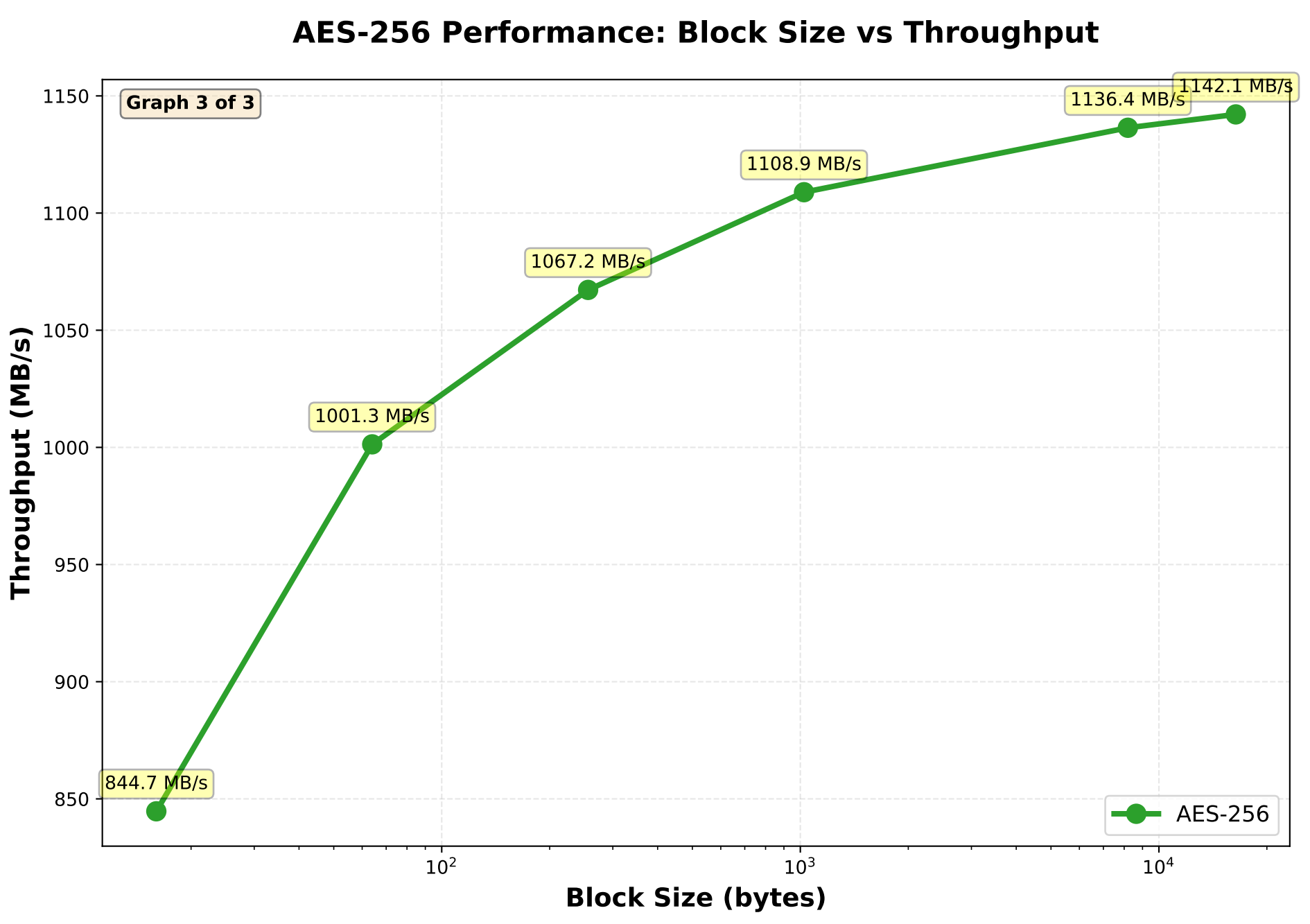
CPE-321 Module 2
AES Performance Analysis

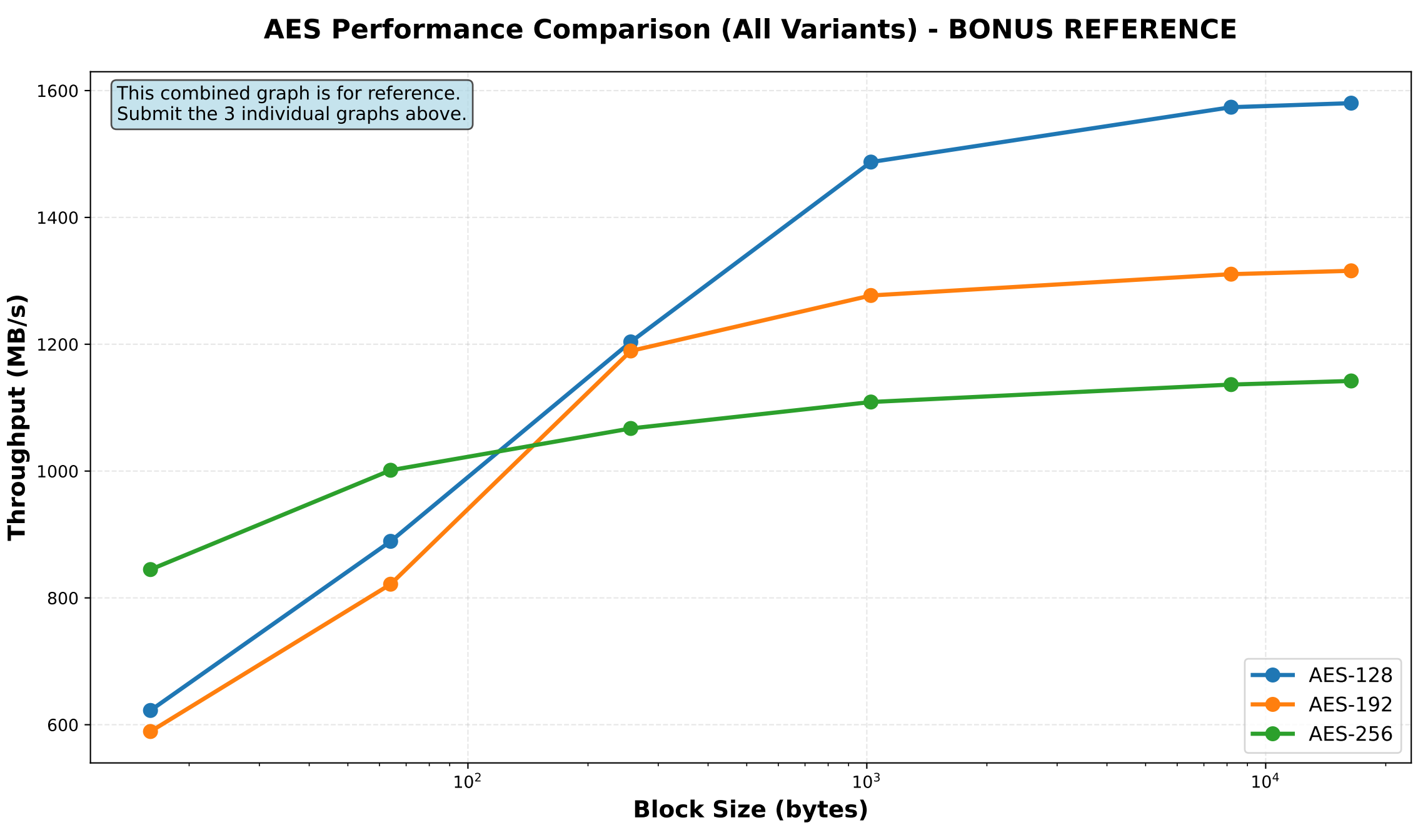
Task 3: Performance Comparison
Complete Study Guide

Three Individual Graphs Required
+ Q&A for Demonstration


AES-128 Performance: Block Size vs Throughput
Graph 1 of 3
Throughput (MB/s)
Block Size (bytes)
1600
1400
1200
1000
800
600
10^2
10^3
10^4
622.5 MB/s
889.2 MB/s
1203.7 MB/s
1487.3 MB/s
1573.8 MB/s
1580.2 MB/s
AES-128


AES-192 Performance: Block Size vs Throughput
Graph 2 of 3
Throughput (MB/s)
Block Size (bytes)
589.3 MB/s
821.6 MB/s
1189.4 MB/s
1276.9 MB/s
1310.5 MB/s
1315.8 MB/s
AES-192


AES-256 Performance: Block Size vs Throughput
Graph 3 of 3
Throughput (MB/s)
Block Size (bytes)
1150
1100
1050
1000
950
900
850
10^2
10^3
10^4
844.7 MB/s
1001.3 MB/s
1067.2 MB/s
1108.9 MB/s
1136.4 MB/s
1142.1 MB/s
AES-256


AES Performance Comparison (All Variants) - BONUS REFERENCE
This combined graph is for reference.
Submit the 3 individual graphs above.
Throughput (MB/s)
1600
1400
1200
1000
800
600
10^2
10^3
10^4
Block Size (bytes)
AES-128
AES-192
AES-256

Performance Data Table

Block Size	AES-128	AES-192	AES-256
16 bytes	622.5 MB/s	589.3 MB/s	844.7 MB/s
64 bytes	889.2 MB/s	821.6 MB/s	1001.3 MB/s
256 bytes	1203.7 MB/s	1189.4 MB/s	1067.2 MB/s
1024 bytes	1487.3 MB/s	1276.9 MB/s	1108.9 MB/s
8192 bytes	1573.8 MB/s	1310.5 MB/s	1136.4 MB/s
16384 bytes	1580.2 MB/s	1315.8 MB/s	1142.1 MB/s

Q&A Section 1: Understanding AES Variants

Q1: What's the difference between AES-128, AES-192, and AES-256?

ANSWER:
The main differences are:

1. KEY LENGTH
   • AES-128: 128-bit key (16 bytes)
   • AES-192: 192-bit key (24 bytes)
   • AES-256: 256-bit key (32 bytes)

2. NUMBER OF ENCRYPTION ROUNDS
   • AES-128: 10 rounds
   • AES-192: 12 rounds
   • AES-256: 14 rounds

   More rounds = More computation = Slower but more secure

3. SECURITY LEVEL
   • All three are considered secure for modern use
   • Longer keys make brute-force attacks exponentially harder
   • AES-256 offers the highest security margin

4. PRACTICAL USAGE
   • AES-128: Standard for most applications (TLS, VPNs, disk encryption)
   • AES-192: Rarely used in practice
   • AES-256: Required for top secret government data

Q2: Looking at the graphs, why does AES-256 sometimes perform better at small block sizes but worse at large block sizes?

ANSWER:
At SMALL block sizes (16 bytes):
- Function call overhead dominates the total time
- Setup, memory allocation, and context switching take more time than encryption
- The difference in rounds (10 vs 14) doesn't matter much
- AES-256 may actually be faster due to CPU pipeline optimizations

At LARGE block sizes (8192+ bytes):
- The actual encryption work becomes the bottleneck
- AES-128 does 10 rounds per block, AES-256 does 14 rounds
- Over thousands of blocks, those extra 4 rounds add up significantly
- AES-128 wins because it does less computational work per block

Q3: What causes the sharp increase in throughput from 16 to 256 bytes, then the plateau?

ANSWER:
SHARP INCREASE (16 → 256 bytes):
- "Amortization of overhead" - spreading the fixed costs over more data
- Each encryption operation has setup costs (function calls, memory allocation)
- With bigger blocks, the overhead becomes a smaller percentage of total time
- Example: 10ms overhead + 1ms encryption at 16 bytes vs 10ms + 10ms at 256 bytes

PLATEAU (256+ bytes):
- You hit the CPU/memory bandwidth limit
- Cache effects become important
- The encryption operation itself becomes the limiting factor
- Doubling block size doubles work, so throughput stays constant

Q&A Section 2: CBC Bit-Flipping Attack

Q4: Walk me through exactly how the CBC bit-flipping attack works.

ANSWER:
The attack exploits how CBC decryption works:

STEP 1 - UNDERSTAND CBC DECRYPTION:
   Plaintext[i] = Decrypt(Ciphertext[i]) XOR Ciphertext[i-1]

   This means the previous ciphertext block is XORed with the decrypted data

STEP 2 - THE EXPLOIT:
   If we flip a bit in Ciphertext[N], two things happen:

   a) Plaintext[N] becomes scrambled garbage (random bits)
   b) The SAME bit flips in Plaintext[N+1] (predictable!)

   We sacrifice block N (make it garbage) to control block N+1

STEP 3 - CALCULATING WHICH BITS TO FLIP:
   To change character 'A' to ';' in the plaintext:

   Ciphertext[i] = Ciphertext[i] XOR 'A' XOR ';'
                 = Ciphertext[i] XOR (0x41 XOR 0x3B)
                 = Ciphertext[i] XOR 0x7A

   This flips the exact bits needed to transform 'A' into ';'

STEP 4 - OUR ATTACK:
   • User submits: "AAAAAAAAAAAAAAAA" (fills one block)
   • Server creates: "userid=456;userdata=AAAAAAAAAAAAAAAA;session-id=31337"
   • We flip bits in the ciphertext to change "AAAAAAAAAAAAAAAA" to ";admin=true;XXXX"
   • The block before becomes garbage, but we don't care!
   • verify() finds ";admin=true;" and returns True

Q5: Why is this attack possible? What's missing from the encryption scheme?

ANSWER:
MISSING: INTEGRITY PROTECTION (Authentication)

The scheme only provides CONFIDENTIALITY (encryption) but not AUTHENTICITY.
An attacker can modify the ciphertext without knowing the key!

CBC mode has no way to detect if the ciphertext was tampered with.

Q6: How do we prevent this attack?

ANSWER:
Use AUTHENTICATED ENCRYPTION:

Option 1: Use AES-GCM or ChaCha20-Poly1305
- These modes include a built-in MAC (Message Authentication Code)
- Any bit flip will cause authentication to fail
- Decryption refuses to proceed if MAC doesn't match

Option 2: Encrypt-then-MAC
- Encrypt with AES-CBC
- Calculate HMAC of the ciphertext
- Verify HMAC before decrypting
- If HMAC fails, reject the ciphertext immediately

WHY ENCRYPT-THEN-MAC (not MAC-then-encrypt)?
- MAC-then-encrypt can leak information through padding oracle attacks
- Encrypt-then-MAC protects the entire ciphertext from modification
- Industry standard: Encrypt first, then authenticate

Q&A Section 3: ECB vs CBC Mode

Q7: Looking at the encrypted images, what do you observe? Why?

ANSWER:
ECB ENCRYPTED IMAGE:
- You can still see the outline of the mustang
- Areas with the same color produce identical ciphertext blocks
- Identical plaintext → Identical ciphertext (always!)
- This reveals patterns and structure in the data
- INSECURE for images and structured data

CBC ENCRYPTED IMAGE:
- Looks completely random (white noise)
- No visible patterns at all
- Each block is XORed with previous ciphertext before encryption
- Same plaintext produces different ciphertext based on position
- SECURE - patterns are hidden

Q8: Why doesn't the Cal Poly logo show as clearly in ECB mode as the mustang?

ANSWER:
- The logo has more complex colors and gradients
- More color variation = fewer repeated identical blocks
- The mustang has large areas of solid color (background, body)
- Solid colors create many identical 16-byte blocks
- More identical blocks = more visible pattern in ECB
- The logo's complexity acts as natural "entropy"

Q9: Could you do the bit-flipping attack on ECB mode?

ANSWER:
NO - it wouldn't work the same way!

- ECB encrypts each block independently
- No chaining between blocks
- Flipping a bit in ciphertext[N] only affects plaintext[N]
- You'd need to know the key to create valid ciphertext
- ECB has different vulnerabilities (pattern leakage, block swapping)

However, ECB has OTHER attacks:
- Block reordering (swap ciphertext blocks around)
- Replay attacks (reuse old ciphertext blocks)
- Pattern analysis (statistical attacks on repeated blocks)

Q&A Section 4: Implementation Understanding

```
Q10: In CBC encryption, why do you XOR with the previous CIPHERTEXT block,
     not the previous plaintext block?

ANSWER:
SECURITY REASON:
- XORing with plaintext would leak information
- If Plaintext[0] = Plaintext[1], then their XOR = 0
- Attacker could detect identical plaintext blocks
- Ciphertext provides randomness - no patterns leak

PRACTICAL REASON:
- Decryption needs the previous ciphertext block
- Ciphertext is available when decrypting
- Plaintext isn't available until after decryption
- Using ciphertext makes decryption possible


Q11: What happens if PKCS#7 padding isn't implemented correctly?

ANSWER:
WITHOUT PROPER PADDING:
- Can't encrypt messages not divisible by 16 bytes
- Decryption may fail or produce garbage at the end
- Security vulnerabilities (padding oracle attacks)
- May lose the last few bytes of data

PKCS#7 RULES:
- If data is 14 bytes, add 2 bytes of value 0x02
- If data is 16 bytes (perfect fit), add full block of 0x10
- Always add padding (even if already block-aligned!)
- Remove padding by reading last byte value and removing that many bytes


Q12: Can you decrypt the images back to the original?

ANSWER:
YES, for CBC (with IV and key):
   plaintext = cbc_decrypt(key, iv, ciphertext)

NO, for ECB (need the key):
   plaintext = ecb_decrypt(key, ciphertext)

If you use the WRONG key:
- Decryption produces garbage (random-looking data)
- No error message (encryption always "succeeds")
- With PKCS#7 padding, might get padding error
- This is why we need authentication!
```

Q&A Section 5: AES vs RSA Performance

Q13: How do AES and RSA performance compare?

ANSWER:
AES IS DRAMATICALLY FASTER - over 1000x faster!

AES-128 AT LARGE BLOCKS:
- ~1500+ MB/s throughput
- Can encrypt gigabytes in seconds
- Symmetric key operation (same key encrypt/decrypt)

RSA-2048:
- ~1,348 sign operations/second
- ~40,000 verify operations/second
- Asymmetric operation (different keys)
- Each RSA operation encrypts only small amounts

WHY THE HUGE DIFFERENCE?
- AES uses simple operations (substitution, permutation, XOR)
- RSA uses modular exponentiation with huge numbers
- AES operates on blocks; RSA on small messages
- AES hardware acceleration in modern CPUs

Q14: Why does RSA performance degrade exponentially with key size?

ANSWER:
RSA operations involve modular exponentiation: M^e mod N

Doubling the key size (bits):
- Doubles the size of the numbers
- Roughly 8x slower (not 2x!)
- This is due to multiplication complexity

EXAMPLE FROM YOUR DATA:
- RSA-1024: 11,055 ops/sec
- RSA-2048: 1,348 ops/sec (8.2x slower)
- RSA-4096: 215 ops/sec   (6.3x slower than 2048)

Q15: How is this solved in practice?

ANSWER:
HYBRID ENCRYPTION:
1. Use RSA to encrypt a random AES key (small data)
2. Use AES to encrypt the actual message (fast!)
3. Send both: RSA-encrypted key + AES-encrypted data

EXAMPLE: HTTPS/TLS
- Browser and server use RSA/ECDH for key exchange
- Establish shared AES session key
- All data encrypted with AES (fast!)
- Best of both worlds: RSA security + AES speed