LAB 3 MANUAL

Name : keerthan pv
SRN : PES2UG23CS272

In every screenshot make sure your SRN is present

This lab needs to be executed in your VM or Linux machine as /dev/random or /dev/urandom are Unix/Linux-specific device files

Task 1

- Provide both the completed code and an output screenshot with the corresponding SRN clearly displayed.
- Ensure that all sub-questions are answered comprehensively.


```
[09/09/25]seed@keerthanpv:~/Documents$ python3 pes2ug23cs272_task1.py
Current timestamp: 1757476762
Generated key (hex): ec82bf72e56475611e095d26e0e80a48
--------------------------------------------------
Current timestamp: 1757476763
Generated key (hex): 01afcfad80cfa608912bb969b14aae84
--------------------------------------------------
Current timestamp: 1757476764
Generated key (hex): f59310400571704310212758 6f057eb4
--------------------------------------------------
```

1. Why do the keys repeat when generated multiple times close together?

The random number generator is being seeded with the current time in seconds. Since the seed only changes once every second, if you generate two keys within the same second, the program starts from the same seed and

produces the exact same sequence of numbers. That's why the keys sometimes repeat when created too close together.

---

2. What is the security risk if an attacker knows the rough time of key generation?

If someone has a good guess of when the key was generated (for example, the attacker knows you ran the program around 3:15:06 PM), they don't need to try all 2^{128} possible keys. Instead, they can just try all the seeds around that second — maybe a few hundred or thousand guesses at most. This makes the key much easier to crack.

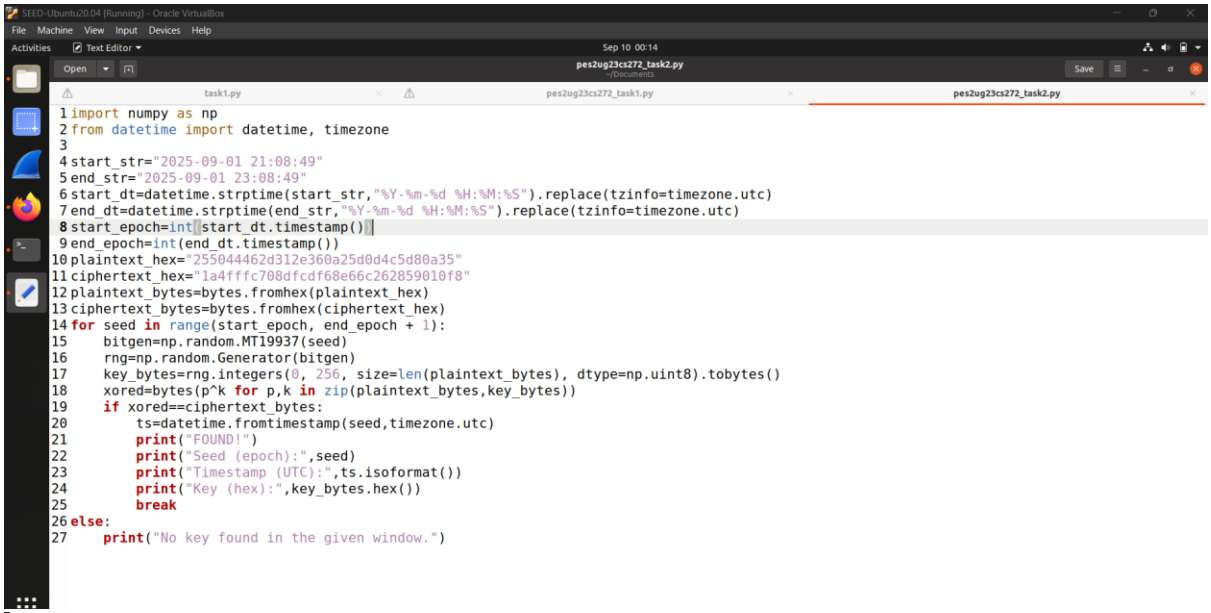
---

3. How does seeding with the timestamp waste the 128-bit potential and fail to maximize randomness?

A true 128-bit key has an enormous number of possibilities — about 3.4×10^{38}. That's practically impossible to brute-force. But when the seed comes from the system time, the possibilities are limited to the number of seconds since 1970 — only around 2^{32} different values. So instead of using the full strength of 128-bit randomness, the actual randomness comes from just a few billion possibilities. That's tiny in comparison, and it makes the keys predictable and insecure.

Task 2

- Complete the given code and provide both the completed code and an output screenshot with the corresponding SRN clearly displayed.
- Ensure that all sub-questions are answered comprehensively.




```
[09/10/25]seed@keerthanpv:~/Documents$ python3 pes2ug23cs272_task2.py
FOUND!
Seed (epoch): 1756761334
Timestamp (UTC): 2025-09-01T21:15:34+00:00
Key (hex): 3f1fbb8125eee3c0844312b640481acd
[09/10/25]seed@keerthanpv:~/Documents$
```

Task 3

- Complete the given code and provide both the completed code and an output screenshot with the corresponding SRN clearly displayed.
- Ensure that all sub-questions are answered comprehensively.

1. Which method is usually the slowest, and why doesn't it always act that way?

/dev/random is usually the slowest because it can stop (block) and wait for real randomness from things like keyboard presses or disk activity. But if the system already has lots of randomness stored, it won't need to wait, so it can sometimes be just as fast.

---

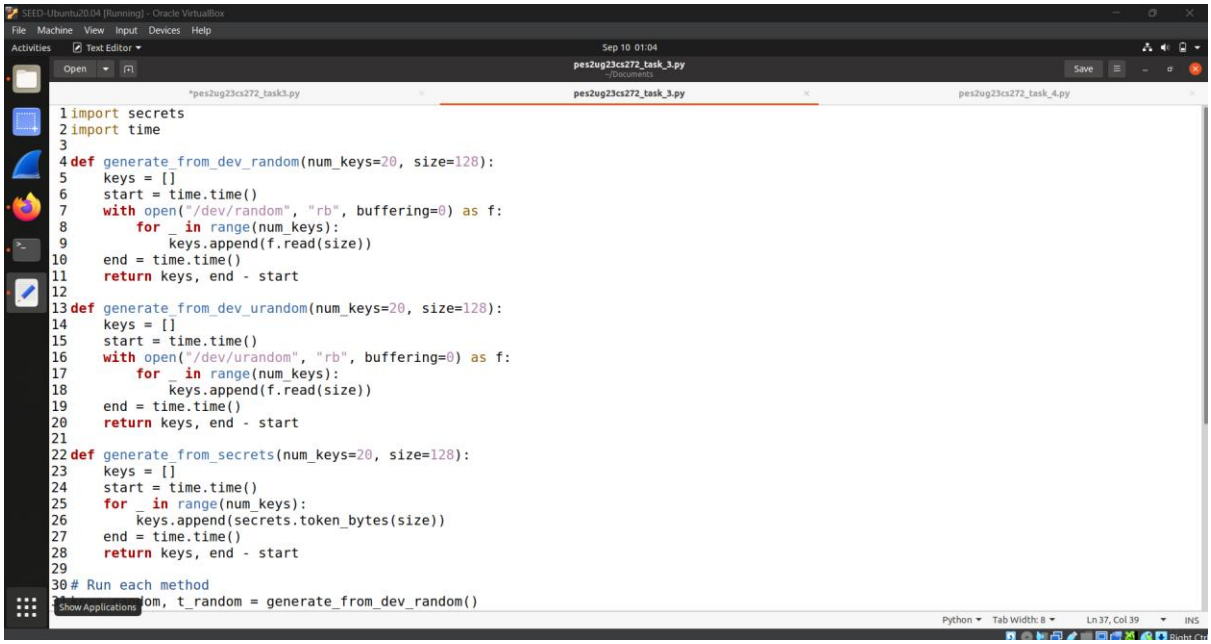
2. What's the main difference between /dev/random and /dev/urandom?

- /dev/random waits until it has "fresh" randomness before giving you data.
- /dev/urandom never waits — it uses a strong algorithm to stretch existing randomness so it can always give you output immediately.

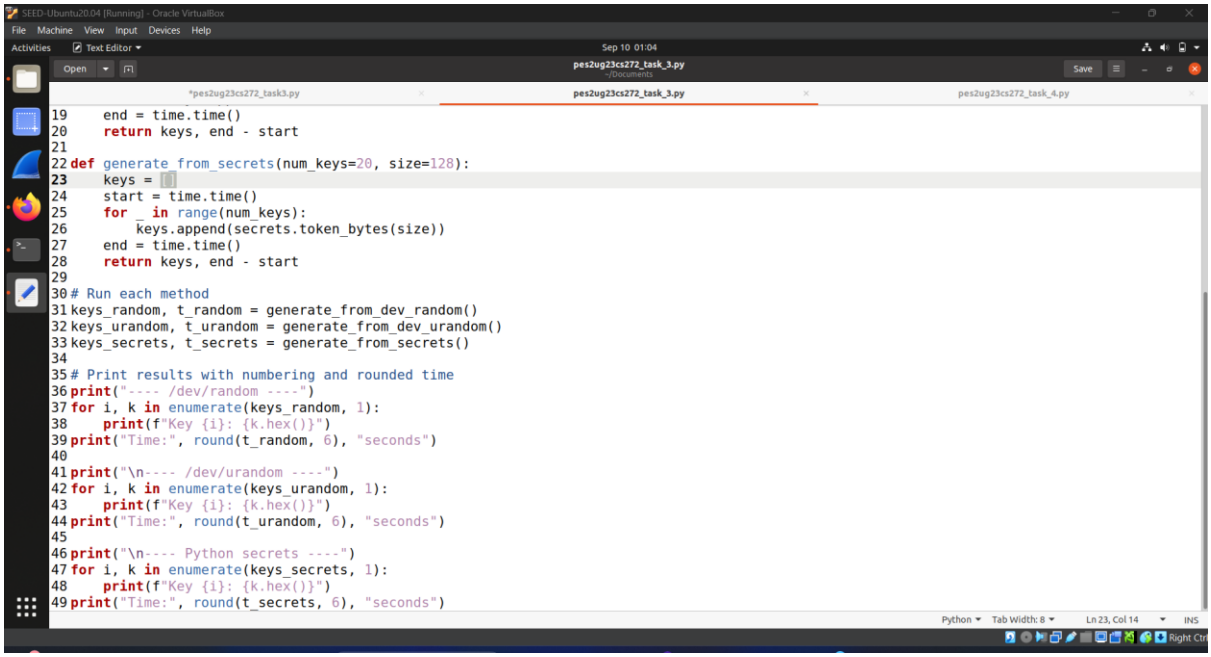
---

3. Which method does Python's secrets module use?

The secrets module uses whatever secure random source the operating system provides. On Linux, this is /dev/urandom. On Windows, it uses the system's secure random functions.

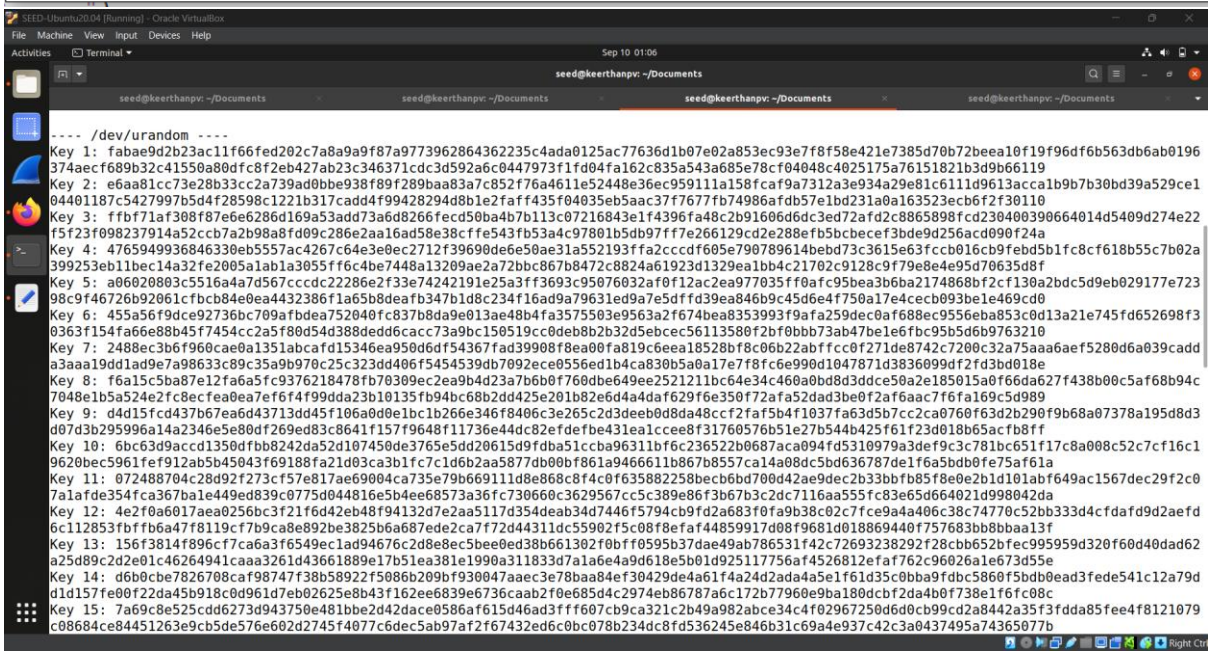


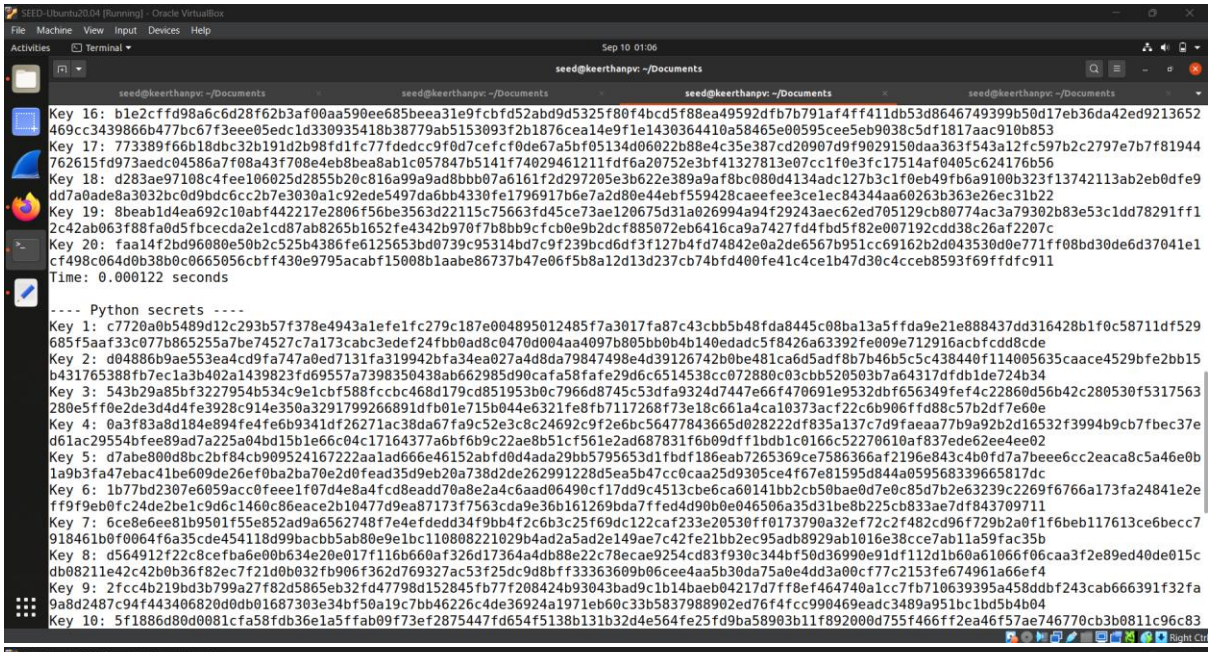
```python
import secrets
import time

def generate_from_dev_random(num_keys=20, size=128):
    keys = []
    start = time.time()
    with open("/dev/random", "rb", buffering=0) as f:
        for _ in range(num_keys):
            keys.append(f.read(size))
    end = time.time()
    return keys, end - start

def generate_from_dev_urandom(num_keys=20, size=128):
    keys = []
    start = time.time()
    with open("/dev/urandom", "rb", buffering=0) as f:
        for _ in range(num_keys):
            keys.append(f.read(size))
    end = time.time()
    return keys, end - start

def generate_from_secrets(num_keys=20, size=128):
    keys = []
    start = time.time()
    for _ in range(num_keys):
        keys.append(secrets.token_bytes(size))
    end = time.time()
    return keys, end - start

# Run each method
om, t_random = generate_from_dev_random()
```



```python
    end = time.time()
    return keys, end - start

def generate_from_secrets(num_keys=20, size=128):
    keys = []
    start = time.time()
    for _ in range(num_keys):
        keys.append(secrets.token_bytes(size))
    end = time.time()
    return keys, end - start

# Run each method
keys_random, t_random = generate_from_dev_random()
keys_urandom, t_urandom = generate_from_dev_urandom()
keys_secrets, t_secrets = generate_from_secrets()

# Print results with numbering and rounded time
print("---- /dev/random ----")
for i, k in enumerate(keys_random, 1):
    print(f"Key {i}: {k.hex()}")
print("Time:", round(t_random, 6), "seconds")

print("\n---- /dev/urandom ----")
for i, k in enumerate(keys_urandom, 1):
    print(f"Key {i}: {k.hex()}")
print("Time:", round(t_urandom, 6), "seconds")

print("\n---- Python secrets ----")
for i, k in enumerate(keys_secrets, 1):
    print(f"Key {i}: {k.hex()}")
print("Time:", round(t_secrets, 6), "seconds")
```

```
seed@keerthanpv: ~/Do...   seed@keerthanpv: ~/Do...   seed@keerthanpv: ~/Do...   seed@keerthanpv: ~/Do...
[09/10/25]seed@keerthanpv:~/Documents$ touch pes2ug23cs272_task_3.py
[09/10/25]seed@keerthanpv:~/Documents$ python3 pes2ug23cs272_task_3.py
---- /dev/random ----
Key 1: 696ae9d862de
Key 2: 3ee620ffa890
Key 3: c0f6a11a4758
Key 4: 66d76b0c8d04
Key 5: 876820057bc3
Key 6: 529b4a6d57e2
Key 7: 5529338a4d43
Key 8: 56517cec3dce
Key 9: 965f28e83583
Key 10: 4f28e610d631
Key 11: 162a44ba465c
Key 12: 899ff79b31db
Key 13: 39e46b6c687c
Key 14: 9b0ed86fe743
Key 15: a252f3c6b39f
Key 16: c30b44064903
Key 17: bcaa9f0739f1
Key 18: 798a79d2b988
Key 19: af55e3b8c267
Key 20: 7fae0338fc9a
Time: 171.902025 seconds
```





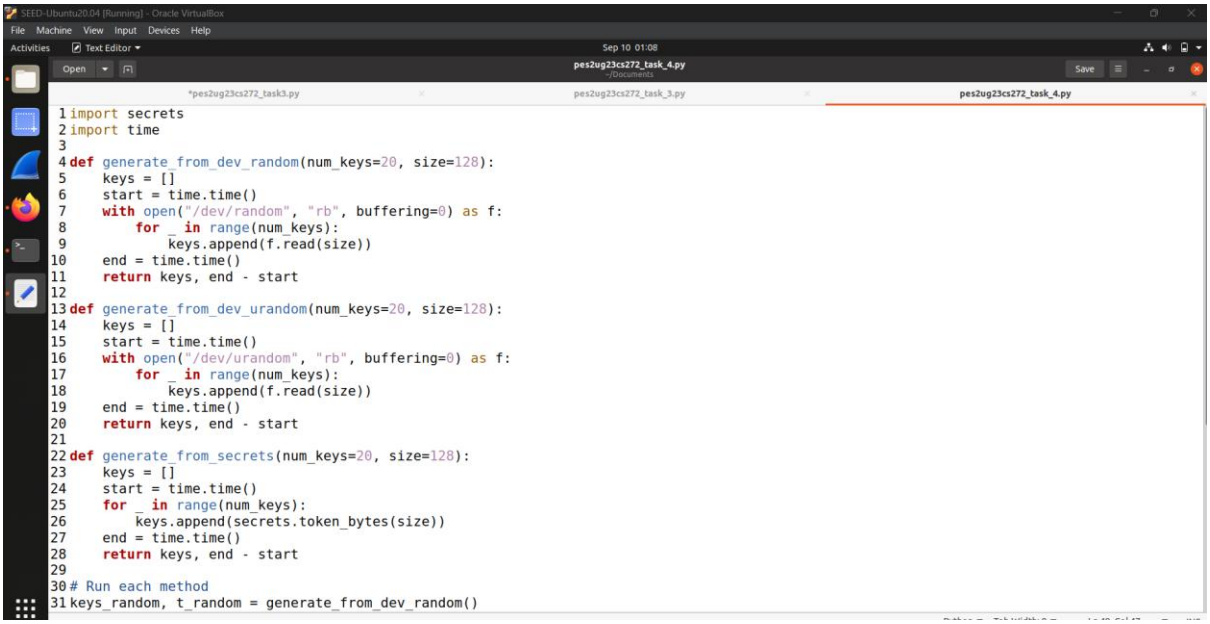
---

4. When would you use /dev/random instead of /dev/urandom?

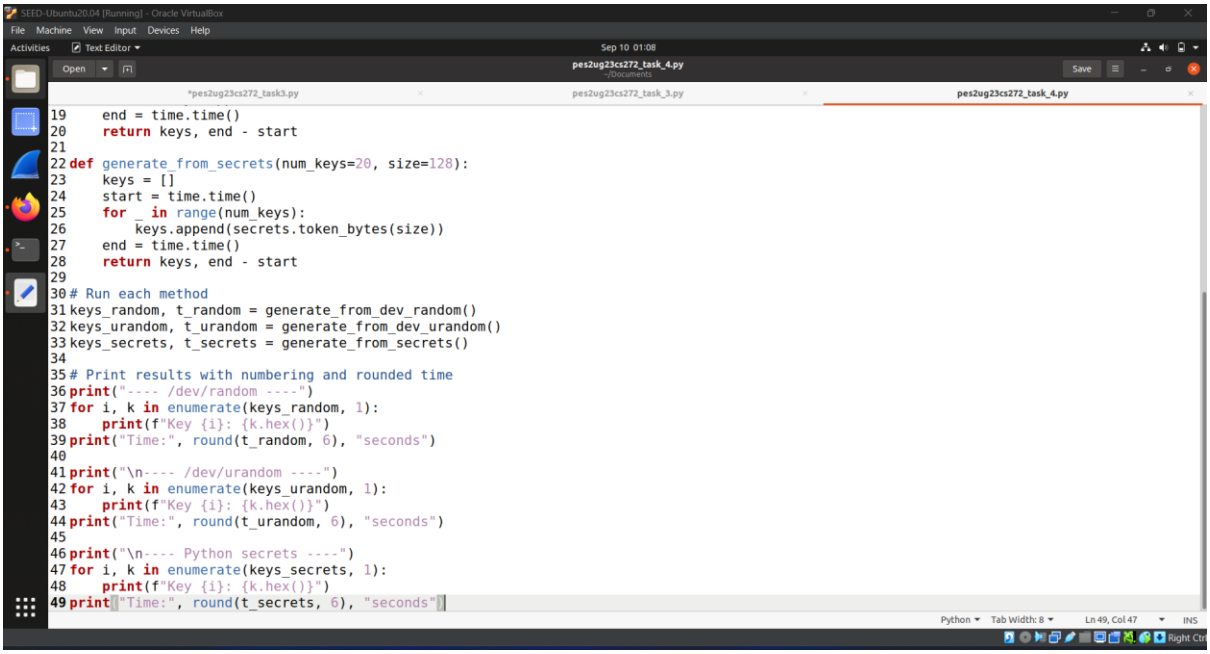
You'd only use /dev/random when you need the very best randomness possible, like creating a very long-term encryption key. For most normal use cases, /dev/urandom or Python's secrets is better, because they're secure enough and don't get stuck waiting.

Task4

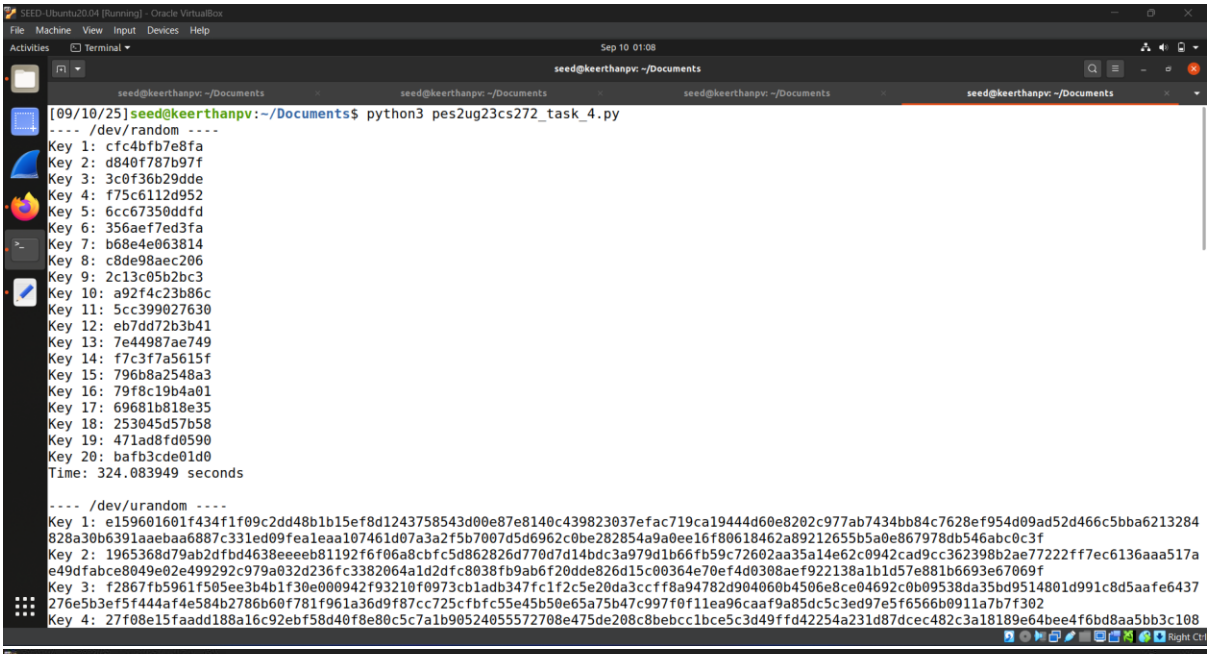
- Complete the given code and provide both the completed code and an output screenshot with the corresponding SRN clearly displayed.
- Ensure that all sub-questions are answered comprehensively.

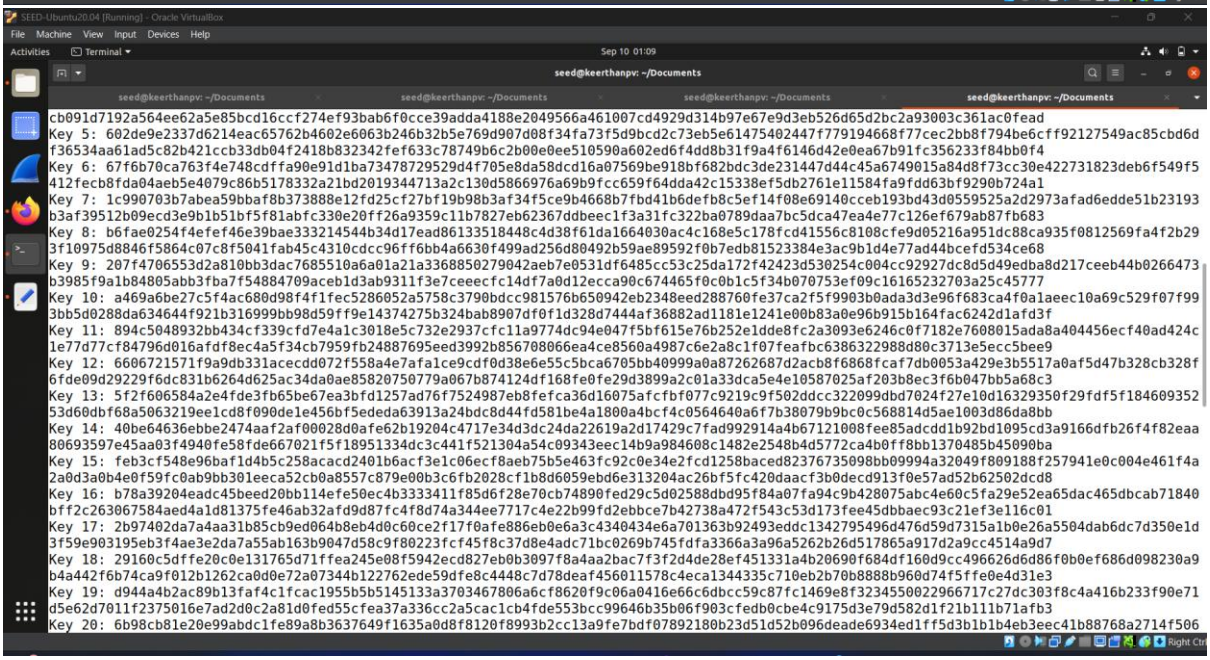


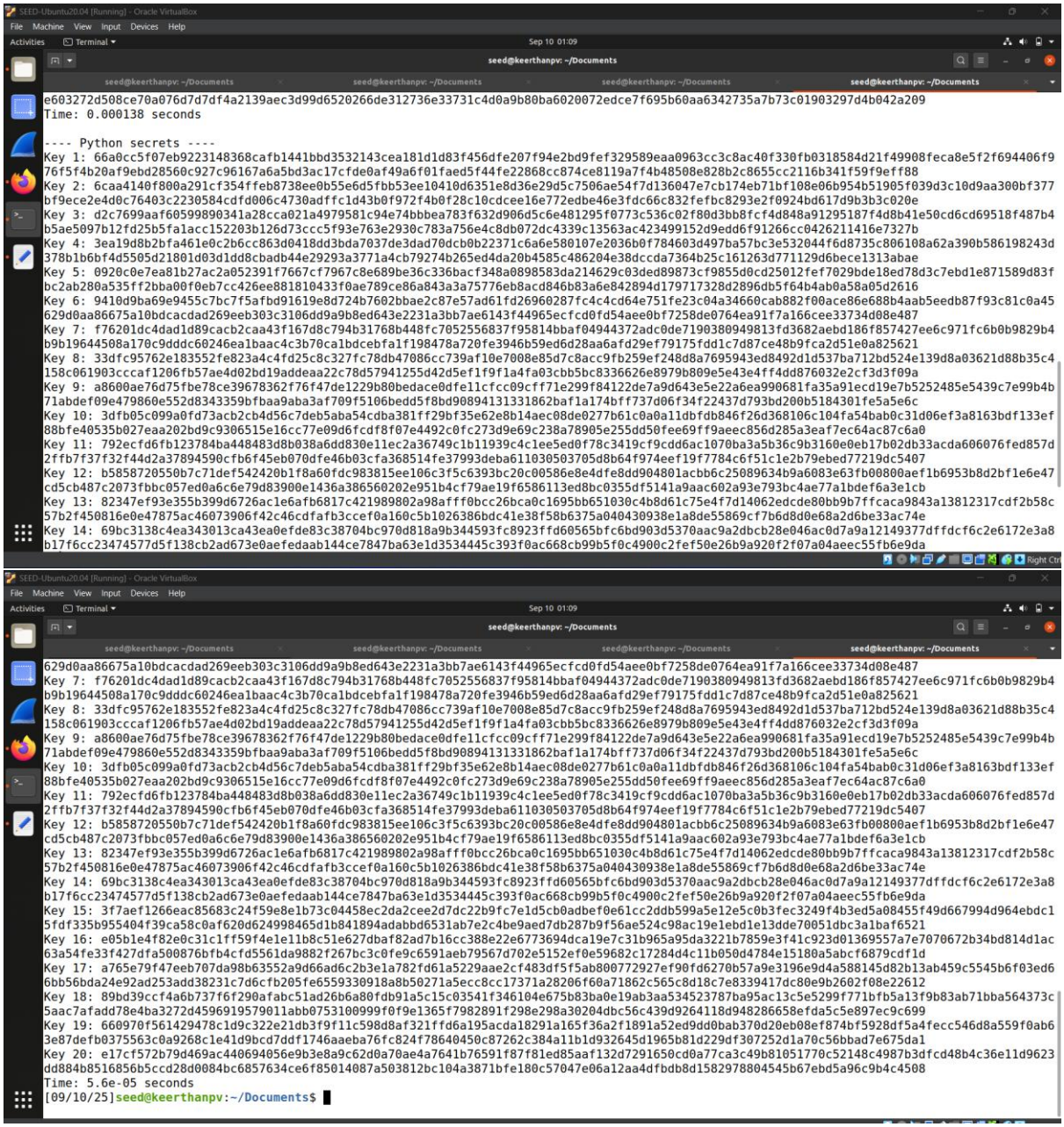
```python
import secrets
import time

def generate_from_dev_random(num_keys=20, size=128):
    keys = []
    start = time.time()
    with open("/dev/random", "rb", buffering=0) as f:
        for _ in range(num_keys):
            keys.append(f.read(size))
    end = time.time()
    return keys, end - start

def generate_from_dev_urandom(num_keys=20, size=128):
    keys = []
    start = time.time()
    with open("/dev/urandom", "rb", buffering=0) as f:
        for _ in range(num_keys):
            keys.append(f.read(size))
    end = time.time()
    return keys, end - start

def generate_from_secrets(num_keys=20, size=128):
    keys = []
    start = time.time()
    for _ in range(num_keys):
        keys.append(secrets.token_bytes(size))
    end = time.time()
    return keys, end - start

# Run each method
keys_random, t_random = generate_from_dev_random()
```



```python
    end = time.time()
    return keys, end - start

def generate_from_secrets(num_keys=20, size=128):
    keys = []
    start = time.time()
    for _ in range(num_keys):
        keys.append(secrets.token_bytes(size))
    end = time.time()
    return keys, end - start

# Run each method
keys_random, t_random = generate_from_dev_random()
keys_urandom, t_urandom = generate_from_dev_urandom()
keys_secrets, t_secrets = generate_from_secrets()

# Print results with numbering and rounded time
print("---- /dev/random ----")
for i, k in enumerate(keys_random, 1):
    print(f"Key {i}: {k.hex()}")
print("Time:", round(t_random, 6), "seconds")

print("\n---- /dev/urandom ----")
for i, k in enumerate(keys_urandom, 1):
    print(f"Key {i}: {k.hex()}")
print("Time:", round(t_urandom, 6), "seconds")

print("\n---- Python secrets ----")
for i, k in enumerate(keys_secrets, 1):
    print(f"Key {i}: {k.hex()}")
print("Time:", round(t_secrets, 6), "seconds")
```


SEED-Ubuntu20.04 [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Activities Terminal Sep 10 01:08
seed@keerthanpv: ~/Documents
seed@keerthanpv: ~/Documents
seed@keerthanpv: ~/Documents
seed@keerthanpv: ~/Documents
seed@keerthanpv: ~/Documents
[09/10/25]seed@keerthanpv:~/Documents$ python3 pes2ug23cs272_task_4.py
---- /dev/random ----
Key 1: cfc4bfb7e8fa
Key 2: d840f787b97f
Key 3: 3c0f36b29dde
Key 4: f75c6112d952
Key 5: 6cc67350ddfd
Key 6: 356aef7ed3fa
Key 7: b68e4e063814
Key 8: c8de98aec206
Key 9: 2c13c05b2bc3
Key 10: a92f4c23b86c
Key 11: 5cc399027630
Key 12: eb7dd72b3b41
Key 13: 7e44987ae749
Key 14: f7c3f7a5615f
Key 15: 796b8a2548a3
Key 16: 79f8c19b4a01
Key 17: 69681b818e35
Key 18: 253045d57b58
Key 19: 471ad8fd0590
Key 20: bafb3cde01d0
Time: 324.083949 seconds
---- /dev/urandom ----
Key 1: e159601601f434f1f09c2dd48b1b15ef8d1243758543d00e87e8140c439823037efac719ca19444d60e8202c977ab7434bb84c7628ef954d09ad52d466c5bba6213284
828a30b6391aaebaa6887c331ed09fea1eaa107461d07a3a2f5b7007d5d6962c0be282854a9a0ee16f80618462a89212655b5a0e867978db546abc0c3f
Key 2: 1965368d79ab2dfbd4638eeeeb81192f6f06a8cbfc5d862826d770d7d14bdc3a979d1b66fb59c72602aa35a14e62c0942cad9cc362398b2ae77222ff7ec6136aaa517a
e49dfabce8049e02e499292c979a032d236fc3382064a1d2dfc8038fb9ab6f20dde826d15c00364e70ef4d0308aef922138a1b1d57e881b6693e67069f
Key 3: f2867fb5961f505ee3b4b1f30e000942f93210f0973cb1adb347fc1f2c5e20da3ccff8a94782d904060b4506e8ce04692c0b09538da35bd9514801d991c8d5aafe6437
276e5b3ef5f444af4e584b2786b60f781f961a36d9f87cc725cfbfc55e45b50e65a75b47c997f0f11ea96caaf9a85dc5c3ed97e5f6566b0911a7b7f302
Key 4: 27f08e15faadd188a16c92ebf58d40f8e80c5c7a1b90524055572708e475de208c8bebcc1bce5c3d49ffd42254a231d87dcec482c3a18189e64bee4f6bd8aa5bb3c108
Right Ctrl


SEED-Ubuntu20.04 [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Activities Terminal Sep 10 01:09
seed@keerthanpv: ~/Documents
seed@keerthanpv: ~/Documents
seed@keerthanpv: ~/Documents
seed@keerthanpv: ~/Documents
seed@keerthanpv: ~/Documents
cb091d7192a564ee62a5e85bcd16ccf274ef93bab6f0cce39adda4188e2049566a461007cd4929d314b97e67e9d3eb526d65d2bc2a93003c361ac0fead
Key 5: 602de9e2337d6214eac65762b4602e6063b246b32b5e769d907d08f34fa73f5d9bcd2c73eb5e61475402447f779194668f77cec2bb8f794be6cff92127549ac85cbd6d
f36534aa61ad5c82b421ccb33db04f2418b832342fef633c78749b6c2b00e0ee510590a602ed6f4dd8b31f9a4f6146d42e0ea67b91fc356233f84bb0f4
Key 6: 67f6b70ca763f4e748cdffa90e91d1ba73478729529d4f705e8da58dcd16a07569be918bf682bdc3de231447d44c45a6749015a84d8f73cc30e422731823deb6f549f5
412fecb8fda04aeb5e4079c86b5178332a21bd2019344713a2c130d5866976a69b9fcc659f64dda42c15338ef5db2761e11584fa9fdd63bf9290b724a1
Key 7: 1c990703b7abea59bbaf8b373888e12fd25cf27bf19b98b3af34f5ce9b4668b7fbd41b6defb8c5ef14f08e69140cceb193bd43d0559525a2d2973afad6edde51b23193
b3af39512b09ecd3e9b1b51bf5f81abfc330e20ff26a9359c11b7827eb62367ddbeec1f3a31fc322ba0789daa7bc5dca47ea4e77c126ef679ab87fb683
Key 8: b6fae0254f4efef46e39bae33214544b34d17ead86133518448c4d38f61da1664030ac4c168e5c178fcd41556c8108cfe9d05216a951dc88ca935f0812569fa4f2b29
3f10975d8846f5864c07c8f5041fab45c4310cdcc96ff6bb4a6630f499ad256d80492b59ae89592f0b7edb81523384e3ac9b1d4e77ad44bcefd534ce68
Key 9: 207f4706553d2a810bb3dac7685510a6a01a21a3368850279042aeb7e0531df6485cc53c25da172f42423d530254c004cc92927dc8d5d49edba8d217ceeb44b0266473
b3985f9a1b84805abb3fba7f54884709aceb1d3ab9311f3e7ceeecfc14df7a0d12ecca90c674465f0c0b1c5f34b070753ef09c16165232703a25c45777
Key 10: a469a6be27c5f4ac680d98f4f1fec5286052a5758c3790bdcc981576b650942eb2348eed288760fe37ca2f5f9903b0ada3d3e96f683ca4f0a1aeec10a69c529f07f99
3bb5d0288da634644f921b316999bb98d59ff9e14374275b324bab8907df0f1d328d7444af36882ad1181e1241e00b83a0e96b915b164fac6242d1afd3f
Key 11: 894c5048932bb434cf339cfd7e4a1c3018e5c732e2937cfc11a9774dc94e047f5bf615e76b252e1dde8fc2a3093e6246c0f7182e7608015ada8a404456ecf40ad424c
1e77d77cf84796d016afdf8ec4a5f34cb7959fb24887695eed3992b856708066ea4ce8560a4987c6e2a8c1f07feafbc6386322988d80c3713e5ecc5bee9
Key 12: 6606721571f9a9db331acecdd072f558a4e7afa1ce9cdf0d38e6e55c5bca6705bb40999a0a87262687d2acb8f6868fcaf7db0053a429e3b5517a0af5d47b328cb328f
6fde09d29229f6dc831b6264d625ac34da0ae85820750779a067b874124df168fe0fe29d3899a2c01a33dca5e4e10587025af203b8ec3f6b047bb5a68c3
Key 13: 5f2f606584a2e4fde3fb65be67ea3bfd1257ad76f7524987eb8fefca36d16075afcfbf077c9219c9f502ddcc322099dbd7024f27e10d16329350f29fdf5f184609352
53d60dbf68a5063219ee1cd8f090de1e456bf5ededa63913a24bdc8d44fd581be4a1800a4bcf4c0564640a6f7b38079b9bc0c568814d5ae1003d86da8bb
Key 14: 40be64636ebbe2474aaf2af00028d0afe62b19204c4717e34d3dc24da22619a2d17429c7fad992914a4b67121008fee85adcdd1b92bd1095cd3a9166dfb26f4f82eaa
80693597e45aa03f4940fe58fde667021f5f18951334dc3c441f521304a54c09343eec14b9a984608c1482e2548b4d5772ca4b0ff8bb1370485b45090ba
Key 15: feb3cf548e96baf1d4b5c258acacd2401b6acf3e1c06ecf8aeb75b5e463fc92c0e34e2fcd1258baced82376735098bb09994a32049f809188f257941e0c004e461f4a
2a0d3a0b4e0f59fc0ab9bb301eeca52cb0a8557c879e00b3c6fb2028cf1b8d6059ebd6e313204ac26bf5fc420daacf3b0decd913f0e57ad52b62502dcd8
Key 16: b78a39204eadc45beed20bb114efe50ec4b3333411f85d6f28e70cb74890fed29c5d02588dbd95f84a07fa94c9b428075abc4e60c5fa29e52ea65dac465dbcab71840
bff2c263067584aed4a1d81375fe46ab32afd9d87fc4f8d74a344ee7717c4e22b99fd2ebbce7b42738a472f543c53d173fee45dbbaec93c21ef3e116c01
Key 17: 2b97402da7a4aa31b85cb9ed064b8eb4d0c60ce2f17f0afe886eb0e6a3c4340434e6a701363b92493eddc1342795496d476d59d7315a1b0e26a5504dab6dc7d350e1d
3f59e903195eb3f4ae3e2da7a55ab163b9047d58c9f80223fcf45f8c37d8e4adc71bc0269b745fdfa3366a3a96a5262b26d517865a917d2a9cc4514a9d7
Key 18: 29160c5dffe20c0e131765d71ffea245e08f5942ecd827eb0b3097f8a4aa2bac7f3f2d4de28ef451331a4b20690f684df160d9cc496626d6d86f0b0ef686d098230a9
b4a442f6b74ca9f012b1262ca0d0e72a07344b122762ede59dfe8c4448c7d78deaf456011578c4eca1344335c710eb2b70b8888b960d74f5ffe0e4d31e3
Key 19: d944a4b2ac89b13faf4c1fcac1955b5b5145133a3703467806a6cf8620f9c06a0416e66c6dbcc59c87fc1469e8f32345500229667l7c27dc303f8c4a416b233f90e71
d5e62d7011f2375016e7ad2d0c2a81d0fed55cfea37a336cc2a5cac1cb4fde553bcc99646b35b06f903cfedb0cbe4c9175d3e79d582d1f21b111b71afb3
Key 20: 6b98cb81e20e99abdc1fe89a8b3637649f1635a0d8f8120f8993b2cc13a9fe7bdf07892180b23d51d52b096deade6934ed1ff5d3b1b1b4eb3eec41b88768a2714f506
Right Ctrl



1. Which sources have the highest entropy?

/dev/random, /dev/urandom, and Python's secrets all give very high entropy, almost the maximum (~8 bits/byte). The test patterns and all-zeros are much lower.

2. Why is /dev/urandom faster but still as good?

/dev/random waits for real hardware noise, so it's slow. /dev/urandom uses a secure algorithm to stretch that noise into more random data, so it's fast but still gives nearly the same randomness.