Practical Reflections

Week 02 Practical Reflection – RSA Timing Test (with my real output)

Running the RSA timing script gave me a much clearer, hands-on understanding of how key size affects performance. Instead of just hearing "bigger keys are slower," I could actually *see* the slowdown in the printed results:

1024-bit → Encrypt 0.20 ms | Decrypt 1.03 ms

2048-bit → Encrypt 0.64 ms | Decrypt 3.54 ms

3072-bit → Encrypt 1.15 ms | Decrypt 10.56 ms

4096-bit → Encrypt 1.90 ms | Decrypt 20.36 ms

What I realized by looking at the real numbers

- Decryption consistently took much longer than encryption, and the gap widened as the key size increased. This was visible immediately in the output — by the time I reached RSA-4096, decryption was about 20 ms per operation, compared to only 1.9 ms for encryption.
- The timing increased in a non-linear way. For example:
    - Going from 1024 - 2048 more than tripled decryption time.
    - But 3072 - 4096 nearly doubled it again.

Having the actual printed values helped me understand that performance drops accelerate at higher key sizes.

- Even though the encryption times increased, they stayed relatively small. Most of the performance cost is clearly in decryption, which is something you only appreciate after running repeated timed operations like this.

What the coding process itself taught me

- Running 200 iterations was essential — if I tested just one encryption, the time fluctuations would hide the pattern. Averaging over 200 made the comparison meaningful.
- I also noticed that **key generation** was noticeably slow for the bigger keys. That wasn't part of the timing test, but seeing the delay in RSA-4096 key creation made me appreciate how expensive key setup can be.
- Printing results for each key size made the performance trend extremely obvious, and I could visually compare everything without guessing.

How this affects practical decisions

- Based on the actual timings from my machine, I can see why systems avoid RSA-4096 for high-volume operations. If I had to decrypt thousands of messages per second, RSA-4096 would be unusably slow.

- RSA-2048 seems like the best balance between speed and security for most real-world applications. The timing output reinforced that choice in a way reading a standard couldn't.

What I would improve

- Next time I would use time.perf_counter() for more precise timing.
- I might also test with larger messages or run the experiment on different hardware to compare performance changes.

Week 03 Practical Reflection – Experiment A (bcrypt cost vs. time)

Running the bcrypt timing test made it very clear how quickly the hashing time increases as the cost factor goes up. The actual numbers I got were:

Cost 8 → ~24 ms

Cost 10 → ~69 ms

Cost 12 → ~250 ms

Cost 14 → ~890 ms

What I learned by looking at my real output

- The jump from cost 12 to cost 14 was huge — almost 1 full second just to hash one password. Seeing it happen on my own machine made it obvious why websites can't just set bcrypt to extremely high cost values.
- The rise wasn't linear at all. For example:
    - Increasing from 8 - 10 only added ~45 ms,
    - But 12 - 14 added over 600 ms.

This showed me that bcrypt cost has exponential effects, not just "slightly slower."

- The experiment helped me understand the *practical trade-off* between security and user experience. With cost 14 taking nearly a second, I could imagine login screens lagging badly.

What I noticed in practice

- Each run generated a different hash even though the password was the same — so I could clearly see that salting was happening automatically.
- The hashing function ran synchronously, so the code froze until the hashing finished. This gave me a sense of what blocking behavior looks like in a real application.

How this affects real-world choices

- Cost 12 felt like the highest "reasonable" setting before performance becomes annoying.
- Cost 14 is probably too slow for busy systems, especially if multiple logins occur at once.

- Measuring this on my own hardware helped me understand that picking a bcrypt cost value should always be hardware-dependent, not one-size-fits-all.

Week 03 Practical Reflection – Experiment B (TOTP Drift Tolerance)

This experiment was interesting because I could visually see how time drift affects TOTP codes. My output was:

Secret: HDC4W5XEUOHAFDQMI7ITLX3233SM3IIF

Time drift -30 sec: 127536

Time drift -15 sec: 985165

Time drift +0 sec: 985165

Time drift +15 sec: 624369

Time drift +30 sec: 624369

What I learned from running the code

- The code at -15 seconds was identical to the current 0-second code, showing that the current 30-second TOTP window overlaps backward.
- The same happened for +15 seconds and +30 seconds, meaning the next time window overlapped forward.
- The –30 second code was completely different, confirming that past a certain drift, you fall outside the valid range.

Seeing this pattern directly in the output helped me understand how precise TOTP timing really is — you can be off by 15 seconds and still match, but being off by 30 seconds already gives completely different results.

Practical things I realized

- This test explained why authentication apps sometimes accept codes from a slightly different time window — they intentionally allow for minor clock drift.
- It also helped me see firsthand why TOTPs fail if your phone clock is too far off. If the device is shifted by a full minute, the generated code won't match at all.
- Generating a random secret each run also showed that the seed isn't predictable or reused.

How this matters in real implementations

- When building 2FA systems, you need to allow for small drift (e.g., ±1 time window), or people get locked out easily.
- You also need to make sure server clocks are synced; otherwise the codes won't authenticate correctly.

Overall takeaway

The experiment wasn't about learning theory — it was the first time I could *see* code values change as time shifted and could confirm with my own output how strict but flexible TOTP really is.

Week 04 Practical Reflection – File Integrity Checker

This experiment was very hands-on because I had to interact with the program by manually editing the file while it was running. The process and the printed output made the behavior of the script easy to understand in a real, practical way.

What I saw happening in real time

The script printed the baseline hash:

9d494be8ab41ae13c25965e0b71a3010b30d7563c1f10b14a3e58b71e378db6e

Then every two seconds it showed:

No change detected...

No change detected...

No change detected...

This continued until I opened *testfile.txt* and modified it. As soon as I saved the file, the program detected the change and printed:

File changed! New hash:
c32e5343541e87f56a14ec842021a81474608d26b28d2861c408a6fa30c0726b

What I learned from actually running the script

- The hash changed completely even though I only edited a small part of the file. Seeing the entire SHA-256 output change made it clear how sensitive hashing is — I didn't need theory; the output itself demonstrated it.
- The program behaved like a simple *real-time integrity monitor*. It kept polling the file every 2 seconds, which made the detection feel immediate once I made an edit.
- Watching the "No change detected..." messages repeating helped me visualize how monitoring systems work behind the scenes — constantly checking for modifications.
- The difference between the baseline hash and the new hash proved that the file was definitely modified, making the script feel reliable.

Practical insights from using the code

- The polling loop made me realize this method could become inefficient for large files or many files, because it re-hashes the entire file every time.
- Editing and saving the file manually let me test how quickly the system responds, and it reacted on the very next loop.
- I also saw that if I saved the file multiple times, the hash would keep changing — this shows how even whitespace or newline differences alter the hash completely.

How this applies in real scenarios

- This simple program behaves similarly to intrusion detection systems that monitor configuration files or critical logs.
- It made me understand the practical limitation of hash-based monitoring: if a file is huge, or if you check many files frequently, it becomes expensive.
- But for small files or critical config files, this kind of script is easy to implement and effective.

What I would improve next time

- Instead of constantly hashing, I would use a filesystem watcher library like watchdog to react instantly without polling.
- I might also log the timestamp of the change or store the old vs. new hash for auditing.

Experiment 1 – Hash Comparison + Avalanche Effect (Practical Reflection)

When I ran the script on my test.txt file, I got:

MD5: 098f6bcd4621d373cade4e832627b4f6

SHA1: a94a8fe5ccb19ba61c4c0873d391e987982fbbd3

SHA256: 9f86d081884c7d659a2feaa0c55ad015…

What I learned from actually running the code

- Even though all three hash functions read the *same file*, their outputs were completely different. Seeing all three hashes printed in front of me helped solidify the idea that algorithms produce unrelated outputs.
- The hashes remained **consistent** every time I ran the script, which confirmed to me that hashing is deterministic — the file didn't change, so the results didn't change.
- The hashes were different lengths, so I could physically see which algorithms produce shorter vs. longer digests.

What I noticed when testing the avalanche effect

- When I changed only a *tiny* piece of the text (adding a letter), the entire MD5, SHA1, and SHA256 outputs changed completely.
- This was the most practical demonstration of the avalanche effect — I didn't need theory; the before-and-after outputs visually showed how dramatic the change is.

Real-world insight

- I realised why even small file tampering is easy to detect. If someone modifies even a single character, the hash visibly changes into something totally unrelated.

Experiment 2 – Extracting Strings from a File (Practical Reflection)

The code scanned my test.txt file and extracted readable ASCII strings.
My output was:

test

What I learned while running the experiment

- Because the file I tested was just a small text file, the only string extracted was literally the word *"test"*. This emphasised that the script isn't doing anything magical — it simply finds readable sequences in binary data.
- Running this experiment made me understand that the amount of meaningful output depends heavily on the content of the file. If I used a PE file or executable, I would get many more strings like:
    - URLs
    - function names
    - suspicious keywords
- The function didn't crash on binary or text files, so the regex method is reliable for mixed or raw data.

Practical observation

- This script is a very lightweight "static analysis" tool. Even though it's simple, it effectively previews the human-readable content of a file — useful for malware triage or forensic tasks.
- It also made me aware that larger files may return **thousands** of strings, so in a real scenario I'd want filtering or keyword searching.

Experiment 4 – Simple YARA Rule (Practical Reflection)

I created a YARA rule to detect "http" inside my file.
The output was:

[]

What this taught me in practice

- The empty result ([]) showed that my file **did not contain** the string "http".
- This helped reinforce how YARA rules work at the simplest level — if the rule's string isn't present, nothing matches.
- It also showed me the rule syntax is working properly because the script ran without errors. If I insert "http" into the file and rerun, the rule should match immediately.

Practical insights

- I realised that YARA is extremely literal. If my file had "HTTP" or "Http", the rule still would not match because YARA string matching is case-sensitive by default.
- The experiment made it clear that building effective YARA rules requires:
    - multiple string patterns
    - wildcards
    - case-insensitive modifiers

- My current rule is very basic, but it confirmed that my YARA environment is set up and functioning.

Real-world application reminder

- Seeing no matches in my own file helped me understand that creating good rules involves testing with *known malicious samples* — otherwise, everything will always return empty.