Project Deliverables

Context

Secure Hashing Algorithm 1(SHA1) was an algorithm developed for the US Government in 1995 to have a standard algorithm for security based applications. It transforms plaintext of any length into a 160 bit string randomly generated and it began to become insecure with cracks showing around 2005 with it generally being assessed as insecure around 2010.

Within this project I create my own SHA1 Implementation from scratch and then attempt to brute force it to enhance my understanding of hashing algorithms,

Problem Statement

The main problem I wanted to solve was that hashing algorithms are poorly understood in that generally Engineers(even security engineers) tend to see hashing algorithms as a blackbox that transforms plaintext into a fixed string. I aimed to fix this by writing easy and simple to read python code which implements the SHA1 hash specification in the simplest possible way. Along with this I wanted to demonstrate a brute force approach of the algorithm to further enhance my understanding of hashing.

Implementation

The code mentioned in report is stored in this GitHub repository here (https://github.com/nullpointertong/SHA1Implmentation). I first wrote all functionalities of the code mentioned bellow without the Visual Studio Code Co-pilot extension and then used copilot with Claude Sonnet 4 to do code reviews/simplify the code.

Implementation wise I started by researching the SHA1 Algorithm and creating a high level design/steps that SHA1 uses to hash an input. I did some research on SHA1 using wikipedia (Wikipedia, 2024) and based of the pseudo code provided I try to create some high level steps I could use/follow in order to code my own implementation of SHA1.

Step 1: Pad and Split the data into 512 bit chunks, Add Padding if not cleanly breakable into 512 bits

Step 2: Preprocess the data into 512-bit chunks
- *Step 2.a:* Convert chunks into 16 32-bit words with big endian order
- *Step 2.b:* Extend the 16 words to 80 words

Step 3: Initialize working variables
Step 4: 80 rounds of mixing:
- *Step 4a:* choose bits from b, c, d
- *Step 4b:* XOR words together
- *Step 4d:* again XOR words together

Step 5: Add the mixed words to the working variables to receive the hash

From this I added the steps as comments and started filling out my code. In Appendix A we can see my implementation for step 1-3. You can see for step 2.b I extended from 16 to 80 words by XORing the words with offsets of 3,8,14 and 16. In Appendix B we can see me cover the rest of the steps highlighted above from 4-5. In Step 4a you can the bitwise operations done and how all the variables are put into an AND operation with the hex value 0xFFFFFFFF in order to keep the values to a 32 bit value.

Along with the implementation of SHA1 I also wrote code to brute force the algorithm itself which can be seen in Appendix C/D. The approach I took for the brute forcing of SHA1 was that I wanted to use a multi-threaded approach where each thread searched a search space of 10000 possible characters and to search the search space I would check if a certain sequence of characters would match the target hash. To achieve this in python due to the limitations of the Global Interpreter Lock python processes are used to achieve concurrency. In Appendix C you can see the verification function and the shared variables I use to retrieve the value if found and the event raised to stop the other processes if a match is found. In Appendix D we see the code that generates the entire search space and then assigns a process to check 10000 combinations each with the code checking if the shared event is raised and then exiting and returning a result the event is raised.

Conclusion

After doing all the work to implement SHA1 from scratch I came away with a couple of learning/conclusions.

The first thing I learnt is that true threats to hashing algorithm are under threat from both exploits which reduce the search space in this case chosen prefix attacks reduce the search space down to $2^{63.4}$ and increasing computational power. With the prevalence of cloud computing as well, now attackers can access pseudo-super computer levels of compute(an AWS EC2 instance with 448 cores and 6tb of RAM has an on demand cost of ~$55 AUD an hour) allowing almost anyone to have access to incredible computational power for relatively reasonable prices. It'll be interesting to see how these threats evolve with incredible leaps in computational power i.e Shor's algorithm/quantum computers making prime numbers factorization polynomial.

The other thing I learnt is that despite the fact that exploits within cryptographic algorithms are very difficult to implement from scratch (hence the exclusion of the chosen prefix attack) due to nature of software and abstraction basically anyone can use the chosen prefix attack which means no matter how complex an exploit is if an exploit is found that is computational feasible an algorithm can be generally considered insecure.

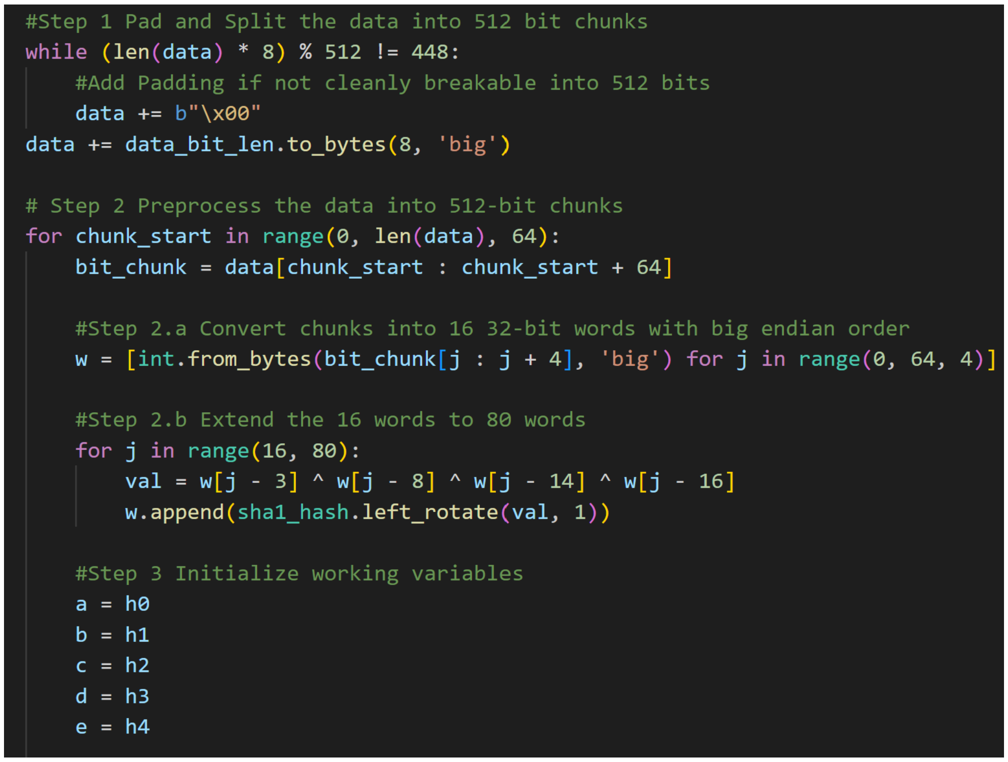
The final thing I learnt is that regardless of algorithm increasing entropy is an important thing to do for your passwords. This is because the biggest use case for cryptographic hashing algorithms is protecting passwords. This came after testing the brute force method on some examples on Appendix E and some other examples outside of the table and realising that increasing the entropy exponentially increases the time it takes to brute force the hash with the common recommendations for passwords i.e increasing length, mixing upper case and lower case along with adding symbols makes a big difference in protecting the hash against brute force attacks.

References

Wikipedia. (2024). *SHA-1 – SHA-1 pseudocode*. [online] Available at: https://en.wikipedia.org/wiki/SHA-1#SHA-1_pseudocode

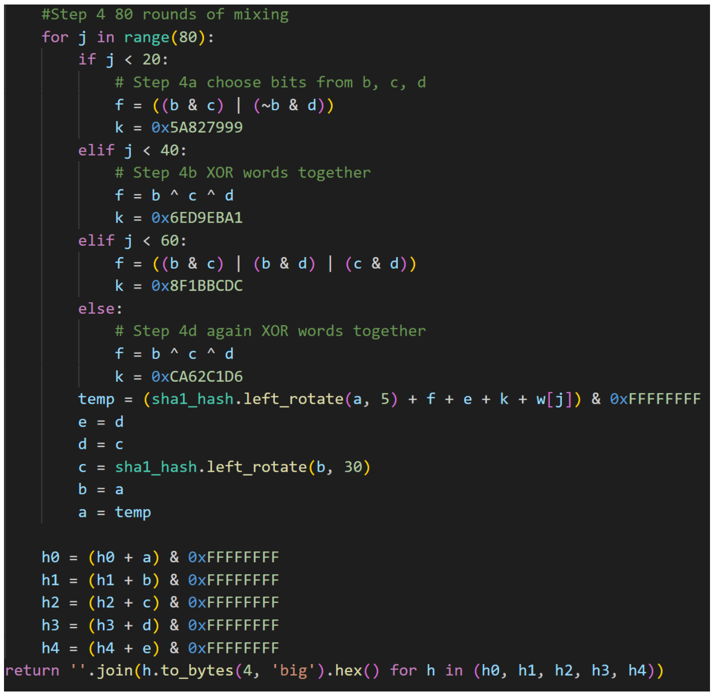
Appendix

Appendix A



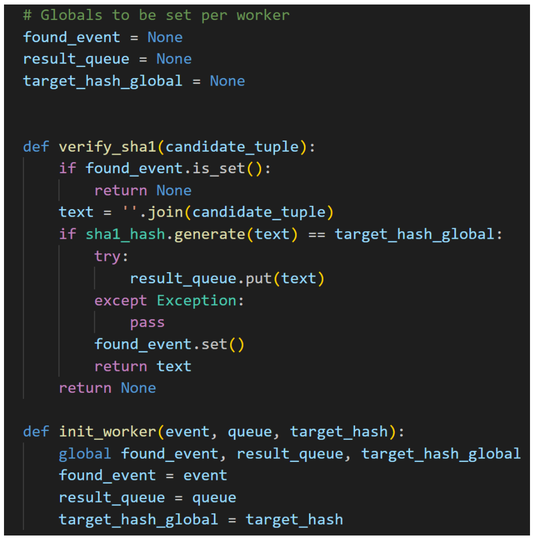
```python
#Step 1 Pad and Split the data into 512 bit chunks
while (len(data) * 8) % 512 != 448:
    #Add Padding if not cleanly breakable into 512 bits
    data += b"\x00"
data += data_bit_len.to_bytes(8, 'big')

# Step 2 Preprocess the data into 512-bit chunks
for chunk_start in range(0, len(data), 64):
    bit_chunk = data[chunk_start : chunk_start + 64]

    #Step 2.a Convert chunks into 16 32-bit words with big endian order
    w = [int.from_bytes(bit_chunk[j : j + 4], 'big') for j in range(0, 64, 4)]

    #Step 2.b Extend the 16 words to 80 words
    for j in range(16, 80):
        val = w[j - 3] ^ w[j - 8] ^ w[j - 14] ^ w[j - 16]
        w.append(sha1_hash.left_rotate(val, 1))

    #Step 3 Initialize working variables
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4
```

Appendix B



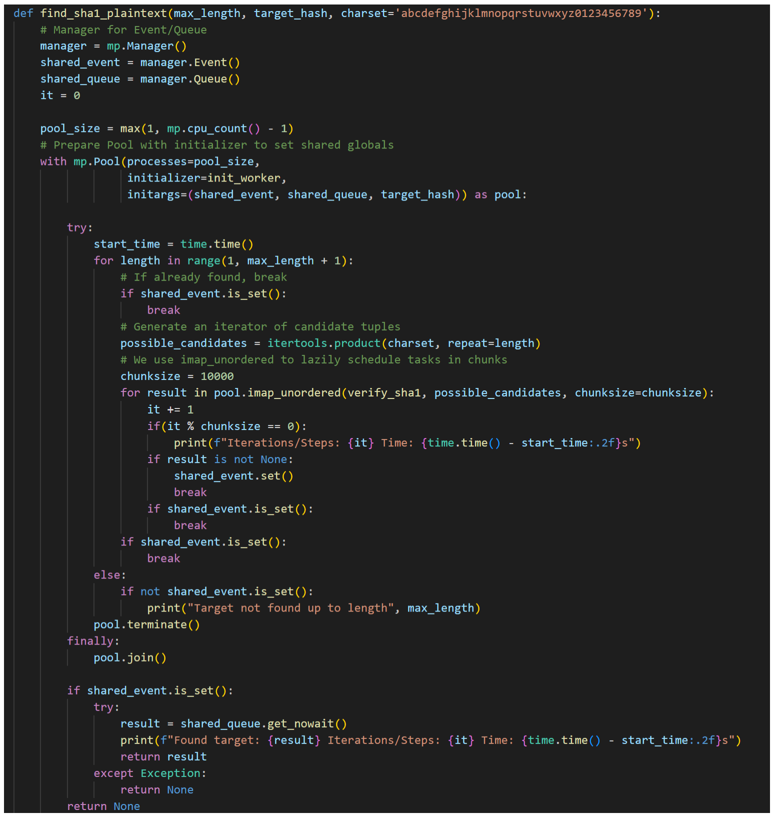
```python
    #Step 4 80 rounds of mixing
    for j in range(80):
        if j < 20:
            # Step 4a choose bits from b, c, d
            f = ((b & c) | (~b & d))
            k = 0x5A827999
        elif j < 40:
            # Step 4b XOR words together
            f = b ^ c ^ d
            k = 0x6ED9EBA1
        elif j < 60:
            f = ((b & c) | (b & d) | (c & d))
            k = 0x8F1BBCDC
        else:
            # Step 4d again XOR words together
            f = b ^ c ^ d
            k = 0xCA62C1D6
        temp = (sha1_hash.left_rotate(a, 5) + f + e + k + w[j]) & 0xFFFFFFFF
        e = d
        d = c
        c = sha1_hash.left_rotate(b, 30)
        b = a
        a = temp

    h0 = (h0 + a) & 0xFFFFFFFF
    h1 = (h1 + b) & 0xFFFFFFFF
    h2 = (h2 + c) & 0xFFFFFFFF
    h3 = (h3 + d) & 0xFFFFFFFF
    h4 = (h4 + e) & 0xFFFFFFFF
return ''.join(h.to_bytes(4, 'big').hex() for h in (h0, h1, h2, h3, h4))
```

Appendix C



```python
# Globals to be set per worker
found_event = None
result_queue = None
target_hash_global = None


def verify_sha1(candidate_tuple):
    if found_event.is_set():
        return None
    text = ''.join(candidate_tuple)
    if sha1_hash.generate(text) == target_hash_global:
        try:
            result_queue.put(text)
        except Exception:
            pass
        found_event.set()
        return text
    return None

def init_worker(event, queue, target_hash):
    global found_event, result_queue, target_hash_global
    found_event = event
    result_queue = queue
    target_hash_global = target_hash
```

Appendix D



```python
def find_sha1_plaintext(max_length, target_hash, charset='abcdefghijklmnopqrstuvwxyz0123456789'):
    # Manager for Event/Queue
    manager = mp.Manager()
    shared_event = manager.Event()
    shared_queue = manager.Queue()
    it = 0

    pool_size = max(1, mp.cpu_count() - 1)
    # Prepare Pool with initializer to set shared globals
    with mp.Pool(processes=pool_size,
                 initializer=init_worker,
                 initargs=(shared_event, shared_queue, target_hash)) as pool:

        try:
            start_time = time.time()
            for length in range(1, max_length + 1):
                # If already found, break
                if shared_event.is_set():
                    break
                # Generate an iterator of candidate tuples
                possible_candidates = itertools.product(charset, repeat=length)
                # We use imap_unordered to lazily schedule tasks in chunks
                chunksize = 10000
                for result in pool.imap_unordered(verify_sha1, possible_candidates, chunksize=chunksize):
                    it += 1
                    if(it % chunksize == 0):
                        print(f"Iterations/Steps: {it} Time: {time.time() - start_time:.2f}s")
                    if result is not None:
                        shared_event.set()
                        break
                    if shared_event.is_set():
                        break
                if shared_event.is_set():
                    break
            else:
                if not shared_event.is_set():
                    print("Target not found up to length", max_length)
            pool.terminate()
        finally:
            pool.join()

        if shared_event.is_set():
            try:
                result = shared_queue.get_nowait()
                print(f"Found target: {result} Iterations/Steps: {it} Time: {time.time() - start_time:.2f}s")
                return result
            except Exception:
                return None
        return None
```

Appendix E

Plaintext	Cracking Time (seconds)
123	1.00
1234	45.30
food	8.8
hello	474.67
hell0	633.55