Hand-Crafting a SHA1 Hash and then Breaking it, Richard Tong, F15A - Friday 3PM

Results

In this project I have created my own implementation of the SHA1 Hashing Algorithm and a multi-threaded brute force you can see this work in my 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 co-pilot with Claude Sonnet 4 to do code reviews/simplify the code. The code first hashes a plaintext into SHA1 and then measures the time taken and the number of steps needed to crack the SHA1 hash

I'm proud of the fact that I was able to get the SHA1 implementation working, given the fact that is a semi modern hashing method and despite the fact that I encountered hard to debug bit manipulation issues/other coding issues.

I am also proud of the multi-processing approach to the password cracker I took. Python uses a Global Interpreter Lock (GIL) so even code using the Python multithreading library is technically still single threaded, so to get around this and to induce concurrency into the cracker I use python processes which are run in parallel. In order to make sure these process ran efficiently in parallel I devised a strategy where I split the search space for the SHA1 algorithm up in chunks of 10000 characters at a time and then if a character is found a shared_event which is shared across the processes is fired to notify the other processes that a match was found.

What I did

In order to manage my time as I mentioned before in my project check-in I used the following milestone table to manage my time:

Milestone	Fallback	Time Box	Status
Write a SHA1 Implementation	Use default implementation	1 week	In progress
Write a basic iterative password cracker	Use built in library	3 days	
Use a multithreading approach to speed up the cracker (stretch milestone)		4 days	
Add a chosen-prefix collision attack approach as well (stretch milestone)		3 days	
Try one of the better know exploits for SHA1(stretch milestone)		3 days	

From the table above I got up to the multithreading approach milestone (marked in red).

Timewise writing the SHA1 Algorithm was a massive time drain which took roughly a week. I mainly used the pseudo code helpfully provided by Wikipedia(Wikipedia, 2024) (See Appendix A for the pseudo code). It was really challenging debugging this code especially with the bit shifting, and it required a lot of back and forth to get working. In order to overcome this I took a multi stepped approach:

1. I consulted the pseudo-code was what I was doing aligned to the pseudo code? (as seen in Appendix A)
2. If still stuck I wrote out on a piece of paper (an example of this is on Appendix C) what was supposed to happen step by step and see where my code and the expected results diverged.

After the work of implementing SHA1 I moved to the next task which was cracking the password programmatically. I first worked on a brute force method iteratively which was relatively easy at first as all I needed to do was to brute force through every combination of characters and took around 3 days. However with the multithreading aspect it became significantly harder as multithreading in python is a bit more difficult due to the presence of the Global Interpreter Lock which means that true multithreading can only be done via Processes not Threads. And as a result their were many hard to debug multithreaded problems with the processes which took skill and a lot of reading of the python process documentation to fix. This took roughly 4 days to figure out.

To recover from these problems I took the following approach:
1. I ran the code in synchronous mode first and stepped into the python debugger and tried to debug the code

2. For multi-threaded/processes errors I'd limit the number of processes to 2 and then from their I would examine how the processes would interact with each other i.e would it exit correct when a shared event was raised etc.

How I was challenged
I was challenged in a couple of ways. Firstly I was not familiar with Bit Manipulation coding so I had to learn things like ensuring an integer stays 32 bit by applying an AND with 0xFFFFFFFF (you can see an example of this in Appendix D) and the annoying little rules python constrains you to for bit manipulation. I.e if you left shift without thinking about thinking about overflow python will just continue expanding the bit size of the variable. To overcome these challenges I used the Visual Studio Code debugger and tried draw out on paper what I need to do with the bits I.e shift the bits this way etc. Through these challenges I learnt that I need to stop relying on computer tools to solve complex problems as drawing the solution out on paper help tremendously in untangling tricky bugs. From the start I wish I drew out on paper what SHA1 was doing and then modelled my code based on that.

I also learnt a lot about concurrency and especially within python i.e the existence of the Global Interpreter Lock and how due to it's existence the need to use processes instead. And the caveats with processes versus "native" threading and the absolute back breaking pain that comes with debugging multithreaded code and shared event triggers. To fix these problems I had to learn how to debug multi-threaded processes using the python debugger in Visual Studio Code. One thing I learnt about myself here is that I tend to get annoyed with problems when I can't solve them and I begin to obsess over them and work long hours while becoming increasingly less productive. I learnt that sometimes the best solution to a difficult problem is to take a step back and come back to it with a fresh pair of eyes. So if I had an opportunity to do a challenge like this again I wish I would take regular breaks when encountering difficult problems.

References

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

Appendix

Appendix A

```
Note 1: All variables are unsigned 32-bit quantities and wrap modulo 2^32 when calculating, except for
        ml, the message length, which is a 64-bit quantity, and
        hh, the message digest, which is a 160-bit quantity.
Note 2: All constants in this pseudo code are in big endian.
        Within each word, the most significant byte is stored in the leftmost byte position

Initialize variables:

h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

ml = message length in bits (always a multiple of the number of bits in a character).

Pre-processing:
append the bit '1' to the message e.g. by adding 0x80 if message length is a multiple of 8 bits.
append 0 ≤ k < 512 bits '0', such that the resulting message length in bits
   is congruent to −64 ≡ 448 (mod 512)
append ml, the original message length in bits, as a 64-bit big-endian integer.
   Thus, the total length is a multiple of 512 bits.

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

    Message schedule: extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
        Note 3: SHA-0 differs by not having this leftrotate.
        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1

    Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    Main loop:[3][56]
    for i from 0 to 79
        if 0 ≤ i ≤ 19 then
            f = (b and c) or ((not b) and d)
            k = 0x5A827999
        else if 20 ≤ i ≤ 39
            f = b xor c xor d
            k = 0x6ED9EBA1
        else if 40 ≤ i ≤ 59
            f = (b and c) xor (b and d) xor (c and d)
            k = 0x8F1BBCDC
        else if 60 ≤ i ≤ 79
            f = b xor c xor d
            k = 0xCA62C1D6

        temp = (a leftrotate 5) + f + e + k + w[i]
        e = d
        d = c
        c = b leftrotate 30
        b = a
        a = temp

    Add this chunk's hash to result so far:
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

Produce the final hash value (big-endian) as a 160-bit number:
hh = (h0 leftshift 128) or (h1 leftshift 96) or (h2 leftshift 64) or (h3 leftshift 32) or h4
```

Appendix B

```python
    # 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:
            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

                start_time = time.time()
                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
```

Appendix C

SHA 1

plaintext. abc → Binary 01100001 0110010 01100011

↓

padding add 1 bit

Binary: 01100001 0110010 01100011 10000000

↓

padding ~~448 mod 512~~ until 512bits / add 0s

↓

break into 16 32bit words

↓

Extend to 80 words → consult formula

↓

Mixing

↓

Add to hash

Appendix D

```python
def left_rotate(value, offset):
    return ((value << offset) & 0xFFFFFFFF) | (value >> (32 - offset))
```