Assignment 5 Immutable Records

Farhaan Jiwa and Ashlyn Schultz
Modern Cryptography
Professor Zheng
12 April 2020

We declare that we have completed this assignment completely and entirely on our own, without any consultation with others. We have read the UAB Academic Honor Code and understand that any breach of the Honor Code may result in severe penalties.

We also declare that the following percentage distribution faithfully represents individual group members' contributions to the completion of the assignment.

Name	Overall Contribution (%)	Major work items completed by me	Signature	Date
Farhaan Jiwa	60	Worked on the development of the code and program. Also helped with edits	Farhaan Jiwa	4/12/2020
Ashlyn Schultz	40%	Worked mostly on the building of report and editing of the document.	Ashlyn Schultz	4/12/2020

Abstract:

In order to create a security system that checks the authenticity of website content, we used SHA256 in the hashlib of python to create a program which generates a Merkle hash tree from snapshots. These snapshots are fictitiously represented by text files Day0-5. These files are then hashed, hashed together and combined to ultimately create the final hash root Day012345. The branches within this hash tree can be used to verify and check hash file integrity. This implementation prevents someone from altering contents of public platforms after they have been snapshotted. Each snapshot that is taken daily is hashed and published so if its authenticity comes intos question, the other branches of the hash tree can be used to verify each other.

Design:

Our design uses a directory of files simulating days 0-5 for a total of 6 days. The days are representations of snapshots of what could be a website or online data collection. Each day, the "snapshot" text document is hashed separately from the next in order to securely preserve an editable copy of the snapshot. The code uses two imports: Sys and Haslib.

The `sys` module built into Python gives information regarding different types of constants, functions, and methods that can be used in a Python interpreter.

The `hashlib` import is used to create a `sha256()` constant for each individual day. This object is a hashed version of the text file used to represent a snapshot of a date. Then the `hexdigest()` function is used to return the hexadecimal concentration of strings within the `sha256()` object.

In order to create a preserved copy of each snapshot, a new object was created. This object was then digested using an update of the `hashlib` in order to prevent any manipulation of the past hashes that were generated and saved from previous days.

Example of Day 0:

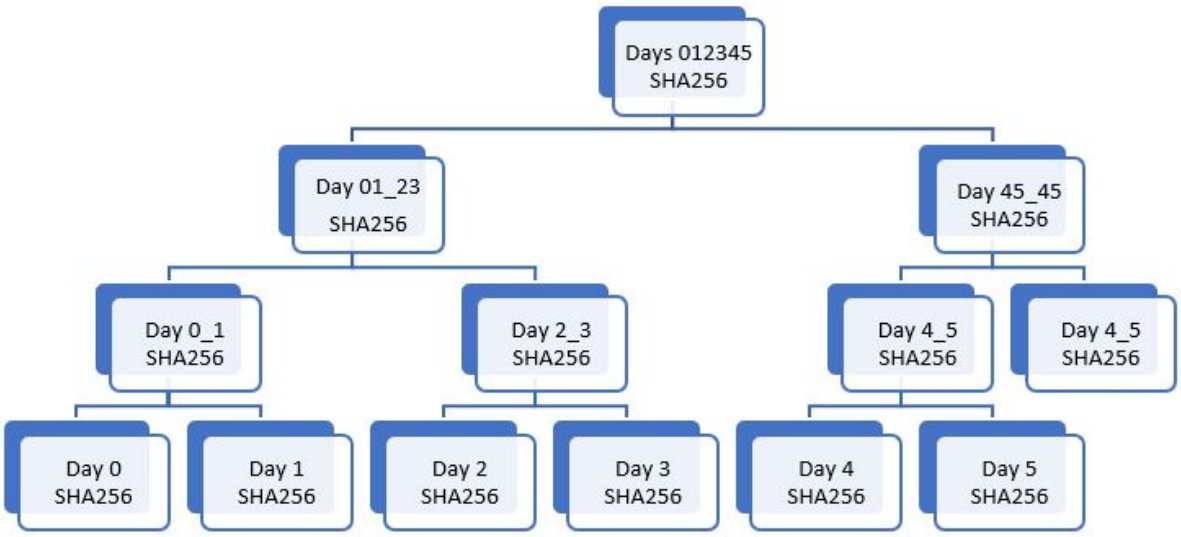
```
day0 = "Day0.txt"
BUF_SIZE = 65536

sha = hashlib.sha256()

with open(day0, 'rb') as f:
        sha.update(f.read(BUF_SIZE))

shahash = sha.hexdigest()

print ("DAY 0 SHA256: ", shahash)
```

Each cycle of the code updates the Day#.txt file to the appropriate day, then the sha object is updated with the new txt file. In order to confirm the hash has been saved and updated correctly, the hex digest is printed . This print can be preserved as an immutable copy, keeping record of the day to day changes between the snapshots.

Program Output:

```
= RESTART: C:¥Users¥Ashly¥Downloads¥one way hashes¥final_copy_of_assignment.py =
DAY 0 SHA256:  e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
Day 1 SHA256:  adb98777abaa3446c354a84a50e16c8219c5c4669a038de07d9e71c9659bd89d
Day 2 SHA256:  7176e0e2f12fdf438dc88fb148991b96a319891c66bb31fec2d813811ffae57b
Day 3 SHA256:  d1c5f2a0cec185387e5286b817215e12ea65cf86ab05251b06a766fa616ae6e7
Day 4 SHA256:  35841160f7bbfabe106bb472837b9317da218d15b6fd78f4229268af69cc5892
Day 5 SHA256:  cf5d4e505180844090212d58f7b73d2c46356a6f08ea95e779d706a4ef528958
Day 0_1 SHA256:  3f66254e701a8513b715c3f656c849f34c4a8328caf49e3a913e9ded7d0824e5
Day 2_3 SHA256:  2f45ab9cc1cc0d7445f0ad7aad5ec554f327cf375a6ca3e5b2988f3f6d77ec9e
Day 4_5 SHA256:  079c7c3b87b80826828c54ccb3a8eef639e5243765356 8f845d87cfc252657dd
Day 01_23 SHA256:  5956180ff170f1dc5300fb6dc3717a4c75843d1ce0d94eaf120f2e71408b91b8
Day 45_45 SHA256:  bd93fe473a9622e25a3f5545b1691488d38b8f0e3dfddef39c2da69972148d21
Root Hash: Days 012345 SHA256:  be5e7fc4d3fc7a27bcc7f12c281d4e336a90354f2e35df39b1daa1e29530fa36
>>>
```

Each day snapshot is individually hashed, and then those hashes are combined in the order `Day 0-1`, `Day 2-3`, `Day 4-5`. After every two hashes are paired, the hashes are combined for a third time with the `Day 0-1` and `Day 2-3` together and then `Day 4-5` is hashed twice together. The `Day 45_45` is created using a self hash so that it can be combined with the `Day 01_23` hash. The final hash contains this combination of those two parts to complete the hash sequence creating the `Day 012345` hash.



This chart displays the pattern in which the program combines each snapshot hash together to create the root hash at the top. Each leaf node of the tree represents one generated hash. The bottom of the tree begins with hashing each original snapshot directly, then as the program moves up the tree, hashes of each day are strategically combined together to generate new hashes.

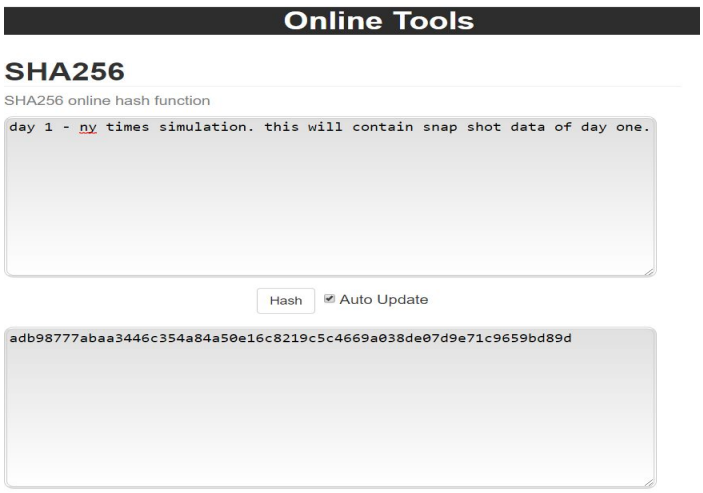
Merkle hash trees such as this one allow for individual branch inspection. This means the integrity of a file or in this case a snapshot can be checked in each particular branch. When files

are damaged or altered, checking certain areas for hash differences can be done in a more focused and precise manner. For example the integrity of `Day 1 SHA256` hash block can be checked using the hash blocks of `Day 0_1 SHA256` and `Day 0 SHA256`. The same can be done on any hash block.

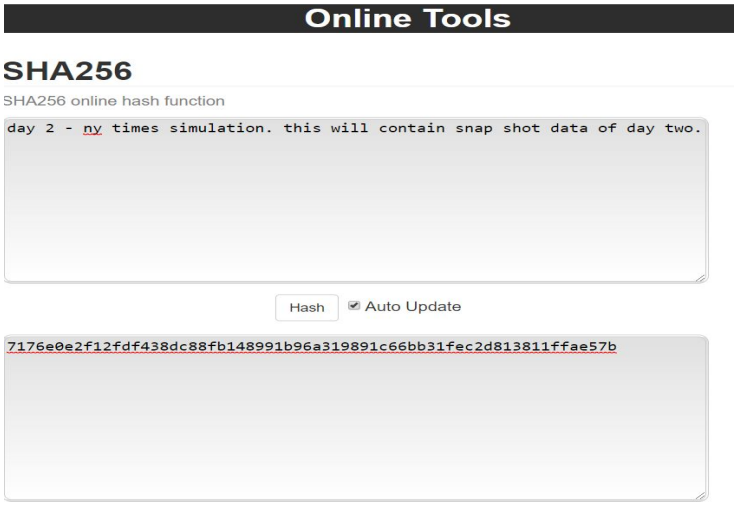
Hash Verification:

In order to verify that our program correctly hashed the snapshot files, the text contained within Days 1-5 were inserted into an online hashing tool at emn178.github.io/online-tools/sha256.html. The hashes for all the days in our program, and the combinations of hashes up to the root hash, match the hashes generated in this online tool to prove validity of our output hashes.

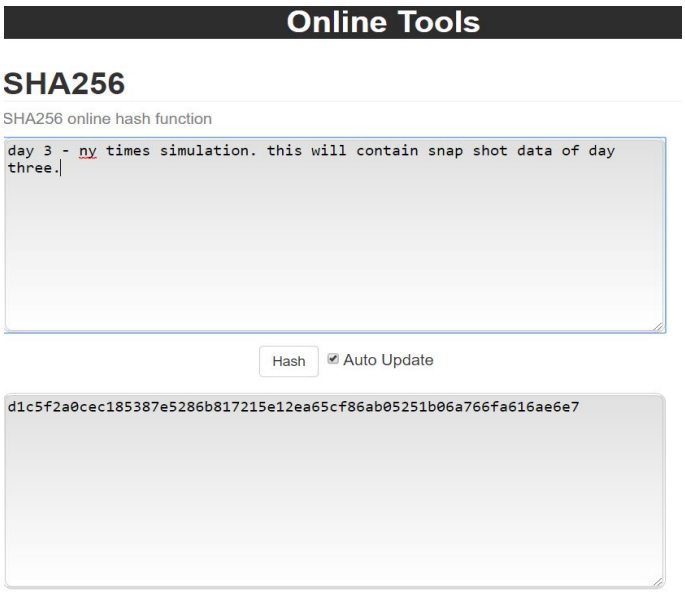
Day 1:



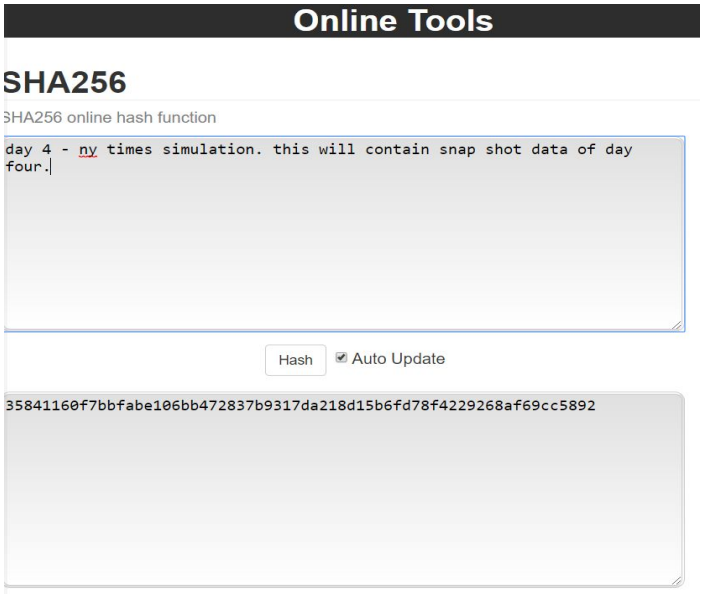
Day 2:



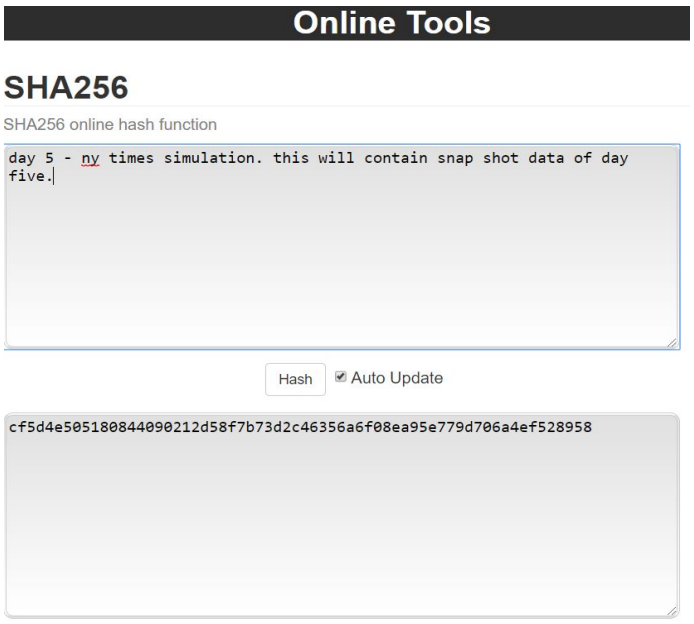
Day 3:



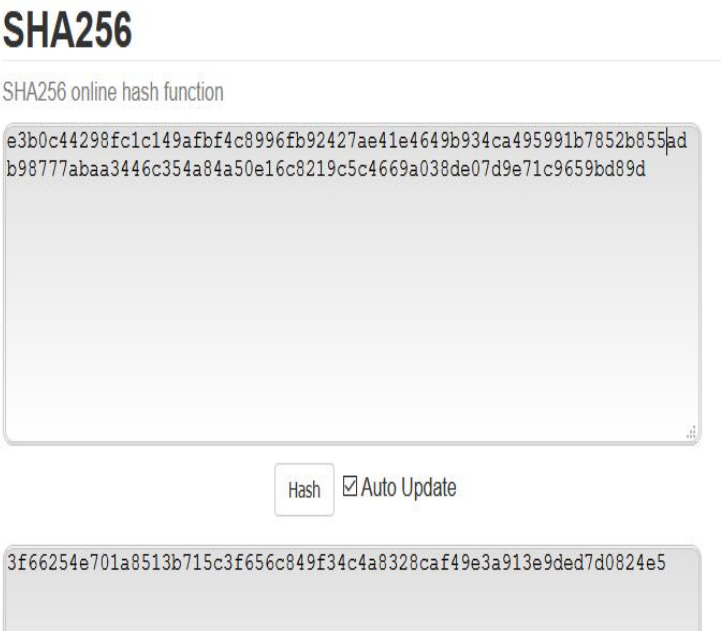
Day 4:



Day 5:



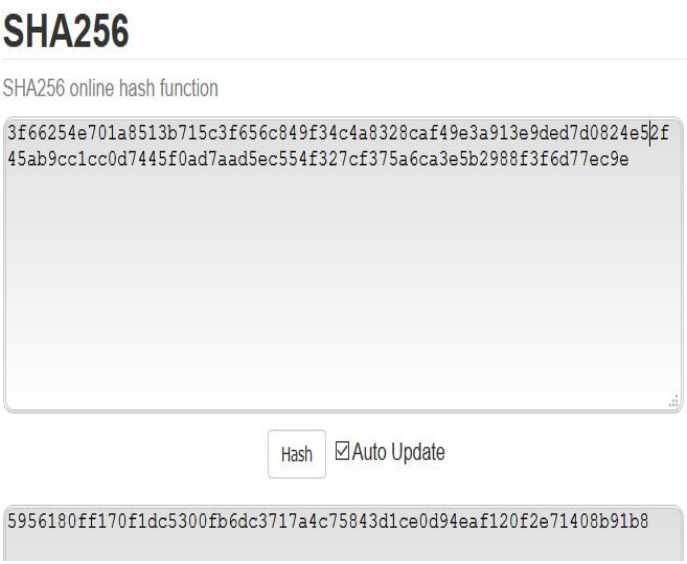
Day 1_2:



Day 2_3:

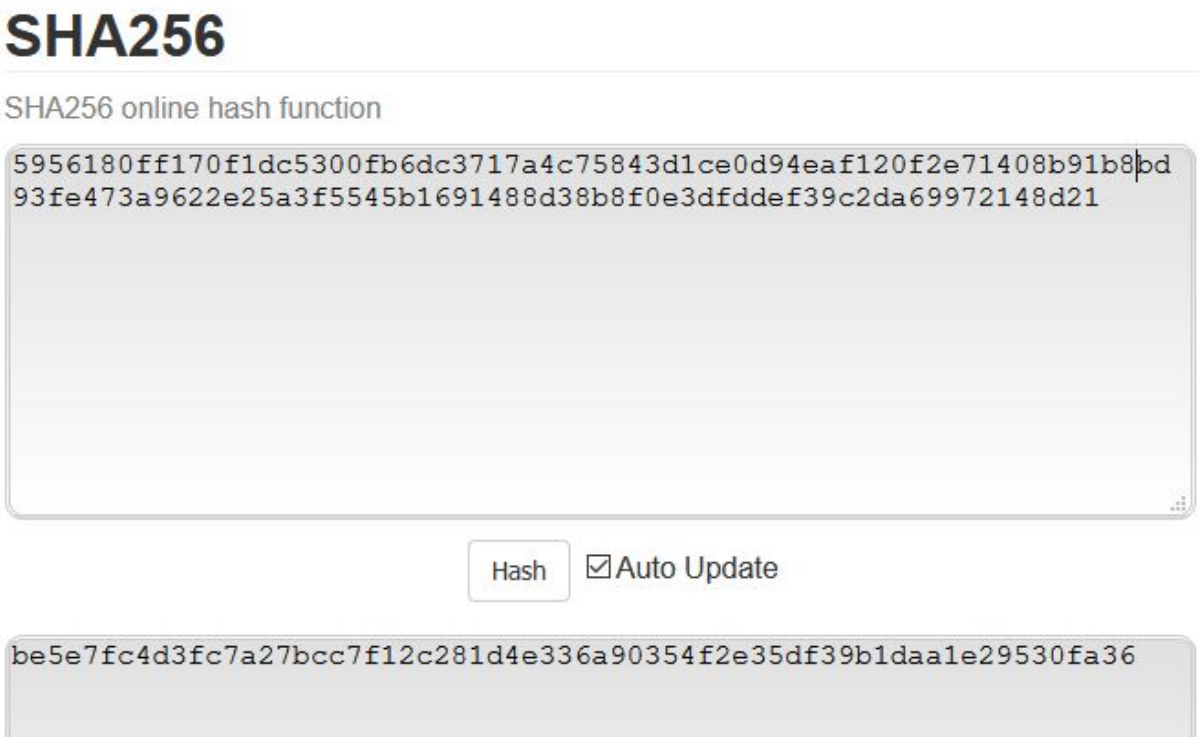
Day 4_5:

Day 01_23:



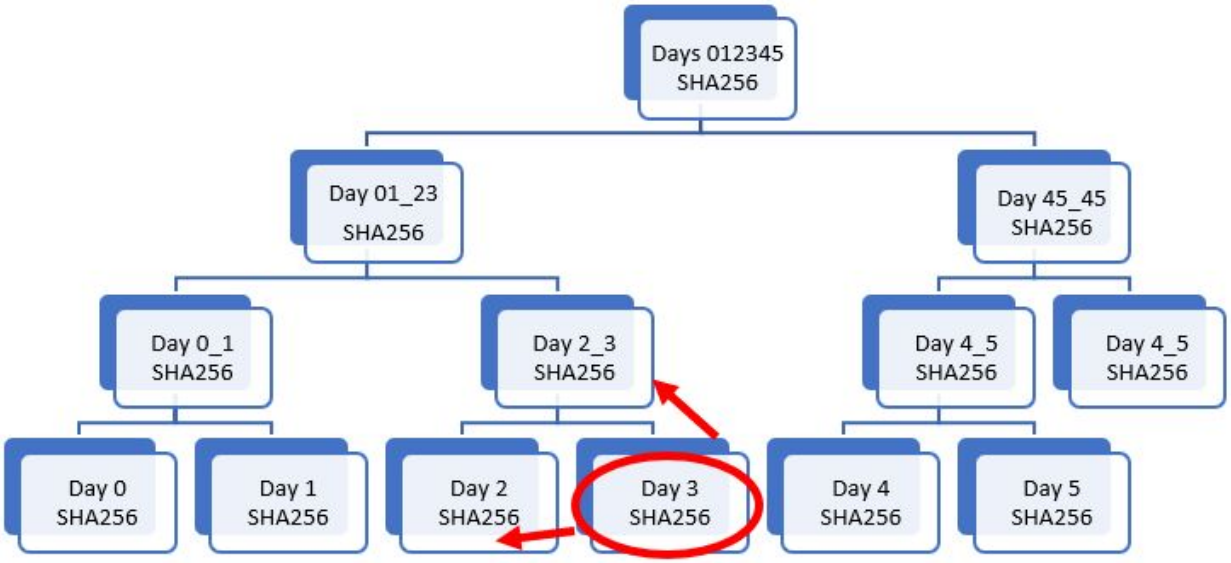
Day 45_45:

RootHash(Day 012345)



Immutability:

If a snapshot is altered, the hashes can be compared for any differences. As the snapshots are hashed and stored daily, if that day's hashed version of the website is altered, the saved original hash can be directly compared to the potentially altered version. If the hashes are different, then the modification can be confirmed.



For example, to check the integrity of Day 3's snapshot hash, the `Day 3 SHA256` hash can be compared to the `Day 2 SHA256` and `Day 2_3` hashes. If the difference in those two hashes does not match the current hash of `Day 3 SHA256`, then the hashed snapshot has been altered. If the hashes match, then the integrity of the hash has been confirmed.

Vulnerabilities:

Second preimaging attacks are probably the biggest vulnerability of Merkle tree hash security programs. This attack occurs when a second hash is created with the same hash root as the original file. This means that even when checking a specific branch of the tree, the hashes would return back as matching even though the files integrity has been compromised. If an attacker were to replace `Day 2_3` hash with a hash pattern that matches a false Day 3 hash and has the same hash root as the tree, when Day 3 is checked against `Day 2 SHA256` and the replaced `Day2_3 SHA256`, the changed hash would not be caught.

Shortened Time Intervals:

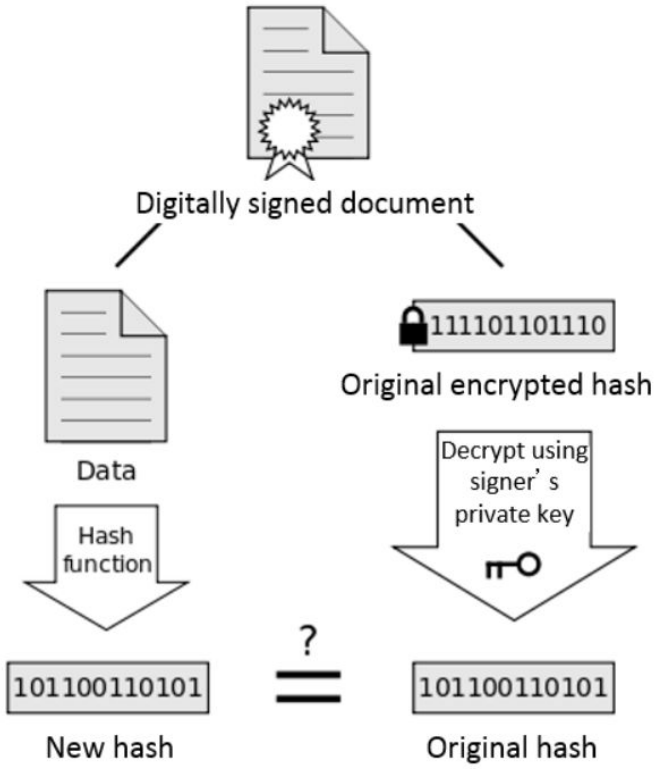
Pros: Shortening the time interval of snapshots that are hashed would allow for higher security. The saved hashes could be used to continually and more frequently verify changes made to a website to avoid and detect falsifications of website content.

Cons: By increasing the rate at which snapshots are taken and stored, the data volume would drastically increase. The hashed versions of the website would require a larger volume available to store.

Ultimately, the perfect balance between most frequent time and available storage for the hashes would have to be calculated. Also, deciding an expiration on website hash values would be important too. If storage hashes for extremely long periods, shorter time intervals could be harmful to the storage capacity.

Possible Improvements:

Using Digital Signature: By applying a digital signature to the hashed snapshots, the security of the snapshot greatly increases. Those without access to the key for the signature would be unable to edit or modify the hashed snapshot in any way. Signatures are also a good way to verify document integrity. If a snapshot is altered, when someone checks the file with their signature key, the hashes will return no match. This would tell an administrator that the hashed snapshot has been modified and the appropriate digital signature was not applied.



Lessons Learned:

We were able to learn how Merkle Tree hashes are created and built through a careful pattern of combined hashes. Creating this program allowed us to explore the structure and creation of Merkle Tree hashes and better understand their vulnerabilities and ways to secure hash files.

We also learned what second preimaging attacks were and how they exploited a vulnerability within Merkle Tree hashes, but that applying extra security like digital signatures could help better defend against this type of attack.

Resources:

Olenski, Julie. “How Do Digital Signatures Work.” *Global Sign*, Global Sign, 18 May 2015, www.globalsign.com/en/blog/how-do-digital-signatures-work.

“Online Tools.” *SHA256 Online*, emn178.github.io/online-tools/sha256.html.