CS 419: Computer Security

Recitation: week of 2020-11-02

Project 4 Discussion

TA: Shuo Zhang
Paul Krzyzanowski

Assignment 13 (Project 4)

- **This assignment is short and comprises 2 parts**
- **This is an <u>individual</u> assignment**
- **Goal: implement a *hashcash*-like Proof of Work system for files:**
  1. Create a header file to accompany a file
     - The header will contain a proof-of-work value for the file
  2. Write a program to validate the proof-of-work header against the file

Environment

- **You should be able implement this on any platform**
  - You may use Go, Python, Java, C, C++
- **But you are responsible to make sure it works on the Rutgers iLab machines with no extra software**
- **You must create executable program or scripts that will run your code**
  - Include a Makefile if your code needs to be compiled
    - We should be able to type *make* to generate the code
  - We should be able to run your programs by typing the commands:
    - ./pow-create
    - ./pow-check

Hashcash

- **Hashcash was system created to reduce spam by requiring sender to:**
  - Solve a difficult problem before sending the message
  - Provide proof of solving this problem
- **For hashcash, this proof was a "stamp" – a header in the mail message**
- **How was this supposed to reduce spam?**
  - Your email client might spend a few seconds solving a problem to create the stamp
  - A spammer who wants to send a million messages would have to spend years of compute time to do this
- **The solution should be verified efficiently by the receiver**
- **The idea behind hashcash was adopted by Bitcoin (and others) as Proof of Work for adding a new block to the blockchain**

The puzzle

- **What problem is easy to solve in one direction but difficult in the other?**
  - One-way functions ⇒ cryptographic hashes
- **A SHA-256 hash of "The grass is green" is f3ccca8f3852f5e2932d75db5675d59de30d9fa10530dd9855bd4a6cd0661d8e**
- **It takes a few milliseconds to compute this**
- **The inverse – find the text when given the hash – requires a brute-force search**
  - Try hashing many possible texts to get that value
- **That's too difficult!**

The easier puzzle

Create some text *W* that when concatenated with the message *M* produces a hash with a certain property

- **A SHA-256 hash of "The grass is green" is f3ccca8f3852f5e2932d75db5675d59de30d9fa10530dd9855bd4a6cd0661d8e**
- **The first high-order bits:** `1111 0011 1100 ...`
- **What can we prefix to the message so the first 6 bits of the hash will all be 0?**
  - We can't figure this out
  - We need to try different combinations … but not a a lot in this case
  - After 41 tries, we find that W="f" and M="The grass is green" produces

hash(W || M) = 0189108649ff4cd02c8af4e0…

= `0000 0000 0001 1000` …

Adaptive difficulty

- **We can set the average difficulty (D) of the problem by changing the number of leading 0 bits we need to find.**

- **Here's how the problem gets difficult with increasing D**
  - Hashing (W || M) where M = "The grass is green"

Difficulty, D	Iterations	Prefix, W	Time (s)
9	1,891	JQ	0.002491
17	20,271	d$3	0.02586
23	1,108,192	et*2	1.4
27	28,415,235	3O941	36.59
28	248,316,223	VaKH9	323.5
30	351,377,855	)FT5D	453.1
31	4,490,406,584	8(i6N2	5063.6
32	22,016,518,319	tJ2IRB	12,270

Your results may vary – these are based on my sequence of W values and my old 3.4 GHz i7 iMac

Adaptive difficulty

- **Large content takes longer to hash than short content**
- **We can keep the content size similar by adding prefixes (W) to the hash of the message M: hash(W || hash(M))**
- **The difficulty is adjusted by changing values of *D*:**
  - Searching for a hash result with *n* leading 0 bits:

$$\textbf{hash(W || hash(M))} < 2^{256-D}$$

- **Will depend on:**
  - Luck (but that averages out with many messages)
  - Your computer speed (and quality of code)
  - Value of *D*

Proof of Work

- **The prefix, *W*, that we found to so the message hash has the desired properties is called the Proof of Work**
- **For example**
  - It took trying 351,377,855 hashes to find a prefix that would cause 'The grass is green' to create a hash with the top 30 bits all 0
  - You only need to do one hash to verify the result
- **Original hash**

```
$ echo -n 'The grass is green' |openssl sha256
f3ccca8f3852f5e2932d75db5675d59de30d9fa10530dd9855bd4a6cd0661d8e
```

- **With Proof-of-work =)FT5D**

```
$ echo -n ')FT5DThe grass is green' |openssl sha256
0000002ccc523fe126c1db89d4ddd426b9f8087f2e29574d29628314fd877ed
```

Your assignment: part 1

- Write a program called pow-create
- It will compute a proof of work string for the specified difficulty
  - For us, difficulty will be the # of leading 0 bits in a SHA-256 hash
- For example, suppose we have a file walrus.txt:

The time has come, the Walrus said,
To talk of many things:
Of shoes — and ships — and sealing-wax —
Of cabbages — and kings —
And why the sea is boiling hot —
And whether pigs have wings.

- We can find the SHA-256 hash with the *openssl* command:

```
$ openssl sha256 < walrus.txt
66efa274991ef4ab1ed1b89c06c2c8270bb73ffdc28a9002a334ec3023039945
```

Your assignment: part 1

- To generate a proof of work with a difficulty of 20, we run

```
$ ./pow-create 20 walrus.txt 2>/dev/null
File: walrus.txt
Initial-hash: 66efa274991ef4ab1ed1b8...28a9002a334ec3023039945
Proof-of-work: hl04
Hash: 000002b2311ce58427ab7c1bfd0cb1...3d948c1c603a524dc11fb28
Leading-bits: 22
Iterations: 1496419
Compute-time: 1.75376
```

- This tells us it took 1,496,419 tests and 1.75 seconds to find a value that can be prefixed to the initial hash value to create a hash whose value has at least 20 leading 0 bits
- The proof of work value is the string `hl04`

Your assignment: part 1 – test your results!

```
$ ./pow-create 20 walrus.txt 2>/dev/null
Initial-hash: 66efa274991ef4ab1ed1b8...28a9002a334ec3023039945
Proof-of-work: hl04
Hash: 000002b2311ce58427ab7c1bfd0cb1...3d948c1c603a524dc11fb28
Leading-bits: 22
Compute-time: 1.75376
```

Recreate the original hash:

```
$ openssl sha256 <walrus.txt
66efa274991ef4ab1ed1b89c06c2c8270bb73ffdc28a9002a334ec3023039945
```

Add the proof-of-work prefix

```
$ echo -n 'hl0466efa27499...9002a334ec3023039945'|openssl sha256
000002b2311ce58427ab7c1bfd0cb1679906b24343d948c1c603a524dc11fb28
```

check the leading bits: [5 0s ⇒ 5*4 = 20 bits of 0] + [2=0010 ⇒ 2 bits of 0]

What you need to do

- **Find the SHA-256 hash of a file**
- **Convert it to a printable hex string (just like the openssl command shows)**
- **Try various prefixes to this printable format of the hash**
  - Compute the SHA-256 hash of the result
  - See if it has at least the desired # of zeros
  - If no, try again

What you need to do: output

Print your output in a standard header format (e.g., mail headers, HTTP headers) — one item per line — with the following fields:

```
File: filename
Initial-hash: sha-256 hash printed as a hex string
Proof-of-work: proof of work string
Hash: sha-256 hash of the proof of work with the hash string
Leading-bits: number of leading 0 bits in the hash
Iterations: how many prefixes you had to try
Compute-time: compute time in seconds
```

Hints

- **Don't write your own SHA-256 function**
  - You can use *hashlib* in python or find source for other languages
  - If using source
    - Do NOT submit entire crypto libraries – prune the source to ONLY the file you need
    - Provide a Makefile – we will not try to figure out how to build anything
    - Make sure it works on the iLab systems
- **Make your hash output look like the same output *openssl* produces**
  - You need this for valid hashing
  - However, do not invoke *openssl* from your program – that would be horribly inefficient
- **It's up to you to figure out prefixes**
  - BUT keep them printable – No whitespace characters and avoid quotes for simplicity

By the way

- **You might want to set thresholds on the # of iterations of prefixes you try to avoid running too long**
- **Test with small difficulty levels – especially on shared iLab systems**
  - Once you get to 30 or so leading 0 bits, it will take a VERY long time
  - Try difficulty values in the range 8 – 20
- **If you were really going to use this:**
  - You would compute the hash based on a binary prefix with a binary hash instead of the string
  - We use text here just for convenience in output and testing
  - The only important values are the proof of work and the # of bits
  - You would use longer difficulty values.

Part 2: Verify

- **The second part of the program is to write a verifier**

  pow-check *powheader file*

- **Checks the proof-of-work in the file *powheader* against the file *file***
- **The *powheader* file is the output of the *pow-create* command**
- **This program:**
  - Validates the hash in the `Initial-hash` header
  - Computes hash of the `Proof-of-work` string prepended to the original hash string
  - Compares this value with the `Hash` header
  - The `Leading-bits` data must match the # of leading 0 bits in the Hash header
- **The output will be "passed" or "failed"**
  - Specify which tests failed

What to submit

- **First, test your programs thoroughly**
  - Test on different input data – don't expect it to be text.
- **Source files only – no object files, Java class files, etc.**
- **If compilation is needed**
  - Include a `Makefile` that will generate the necessary executables from source
- **Provide or generate two programs**
  - `pow-create` *difficulty sourcefile*
  - `pow-check` *headerfile sourcefile*

The End