Massachusetts Institute of Technology
6.857: Network and Computer Security
Professors Ronald L. Rivest and Yael Tauman Kalai

Handout 3
February 27, 2017
Due: March 13, 2017

Problem Set 2

This problem set is due on *Monday, March 13, 2016* at **11:59 PM**. Please note our late submission penalty policy in the course information handout. Please submit your problem set, in PDF format, on Gradescope. There will be a place in Gradescope to add your team members to the assignment. Have **one and only one group member** submit the finished problem writeups. Please title the PDF file with the Kerberos of your group members as well as the problem set number (i.e. *kerberos1_kerberos2_kerberos3_pset2.pdf*).

You are to work on this problem set with your assigned group of three or four people (which will be sent out by email). If you have not been assigned a group, please email `6.857-tas@mit.edu`. Be sure that all group members can explain the solutions. See Handout 1 (*Course Information*) for our policy on collaboration.

Homework must be submitted electronically! Each problem answer must be provided as a separate pdf. Mark the top of each page with your group member names, the course number (6.857), the problem set number and question, and the date. We have provided templates for LaTeX and Microsoft Word on the course website (see the *Resources* page).

Grading: All problems are worth 10 points.

With the authors' permission, we may distribute our favorite solution to each problem as the "official" solution—this is your chance to become famous! If you do not wish for your homework to be used as an official solution, or if you wish that it only be used anonymously, please note this in your profile on your homework submission.

Our department is collecting statistics on how much time students are spending on psets, etc. For each problem, please give your estimate of the number of person-hours your team spent on that problem.

Problem 1-1. What if Bitcoin had used SHA-1?

Perfect timing for this class! It has just been announced that collisions have been found for SHA-1:

`https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html`

`http://shattered.io/`

It seems we can no longer trust SHA-1 to be a collision-resistant hash function.

Please carefully describe the consequences to Bitcoin if Bitcoin had used SHA-1 instead of SHA-256.

(Here is a **big** hint: `https://eprint.iacr.org/2016/167.pdf`.)

Problem 1-2. 857coin

Rumor has it that a new cryptocurrency has sprung up at MIT!

In 857coin, blocks operate just as in bitcoin. Blocks has a single nonce n. To compute a proof of work for a difficulty d, we compute the hash h of the block with respect to the nonce n and then check that the first d bits of h are all 0.

Note: this problem is somewhat of an experiment for us, and we reserve the right to tweak it with reasonable warning as events unfold.

(a) To get started, visit `http://6857coin.csail.mit.edu:8080/` And read the API for 857coin. Then, look at the provided miner.py template and make the required modifications to begin mining. (miner.py use argon2_cffi package. To install it, run "pip install argon2_cffi") You will receive full credit for part (a) after successfully mining a block that appends to any tree rooted at the genesis block. To receive credit for your team, include your team members' usernames separated by commas in the block contents.

(b) Now see where you can optimize your miner even further. The slower your miner is in comparison to other miners, the longer it will take to add to the main (longest) chain. You will receive full credit for part (b) if you ever append to the main[1] chain and a description of your strategy for mining a block on the main chain (hint: parallism!). Remember to include your team in the block contents! Also note that the earlier you start, the slower your competition will be! Feel free to get creative by using different languages or hardware.

(c) (Optional food for thought / open research question) Argon2 is the password-hashing function that won the Password Hashing Competition a couple of years ago. Evaluating Argon2 is supposed to require a certain amount of memory usage (specifically, any evaluation algorithm is supposed to have to store a certain number of hash outputs during the evaluation), and therefore, it is hoped that an attacker using special-purpose hardware (like ASICs) to try hashing lots of potential passwords in parallel would still need time-per-evaluation similar to an honest evaluator, since the memory requirement should be a bottleneck. However, it has not been proven that Argon2 cannot be evaluated using less memory[2] than the standard implementation.

It would be very interesting to find a way to evaluate Argon2 while storing fewer hashes than the standard implementation! *Beware, however, that this is an* ***open research question****:* it may be fun to think about, but invest time in it at your own peril. If you've finished the rest of the homework and want to put some thought into this, here are some resources. *Looking at these resources and putting thought into this are both* ***entirely optional!***

- Argon2 specification: `https://github.com/P-H-C/phc-winner-argon2/blob/master/argon2-specs.pdf`. Note that for our problem, we are using Argon2d with 3 iterations, 4M memory, 1 parallelism, and 32-byte output.
- A 2015 research paper describing some ways to decrease the memory usage of Argon2i... but their techniques as described are not efficient enough to deploy in practice: `http://eprint.iacr.org/2016/759.pdf`

Problem 1-3. Stateless hash-based signatures

This problem proposes a stateless hash-based digital signature scheme based on the use of a Merkle tree and asks you to do some simple analyses of it. One motivation for using hash-based signature schemes is that they may be more resistant to attacks by quantum computers.

(The scheme is not entirely new; you can find schemes like it in the literature. We won't give you citations; if you look and find something, you must cite what you found. But it is not necessary to consult the literature to solve this problem.)

We develop this idea by presenting five versions: for signing one bit once, for signing a number of one-bit messages, for signing a number of variable-length messages, for doing so with a small public key, and for doing so in a stateless manner.

The final scheme is "stateless": producing a signature doesn't require knowing how many or what messages have been signed previously, or knowing what signatures have been produced for those messages. Each signing operation begins in the same state (except for knowledge of the message to be signed.) A stateless signature scheme is useful if you have several servers signing on behalf of a given web site.

The scheme is nonetheless "bounded" – you should sign at most s messages with a given secret signing key, where s is a parameter to the key-generation routine (imagine $s = 2^{30}$). After s signatures have been made, the signer should run key-generation again to produce a new public-key/secret-key pair. Security begins to degrade if one signs more than s messages.

Version I

This first version is for signing a single bit, just once.

[1] Longest at the time of the deadline.

[2] When we say "memory" here, we're referring specifically to the number of hash outputs stored.

Let h be a one-way collision-resistant hash function, such as SHA-256.

Alice randomly chooses two 256-bit values, p and q, and publishes as her public key the value $R = h(h(p)||h(q))$.

To later sign a bit b, she reveals p and $h(q)$ to sign $b = 0$, and reveals $h(p)$ and q to sign $b = 1$. Verification is done in the obvious way, recomputing R from the values revealed. The public key is only good for one signature.

(a) Explain carefully why signing $b = 0$ doesn't allow an adversary who learns p and $h(q)$ to produce a valid signature for $b = 1$.

Version II

The second version of the signature scheme is an extension of Version I for signing many bits; basically it just replicates Version I many times.

The public key R is now a sequence x_1, x_2, ..., x_n for some large $n = 2^k$, where each x_i is produced as in version I from randomly chosen p_i and q_i. The public key can be very big if n is large.

In version II, each message to be signed is also just a bit. Suppose the signer has already signed $t - 1$ bits, and now wants to sign the t-th bit, b_t. The signer uses x_t as in version I, and reveals either $(t, p_t, h(q_t))$ or $(t, h(p_t), q_t)$ according to the value of b_t.

As before, the public key values can each be used for at most one signature, so at most n messages (each one bit long) can be signed.

Note that this scheme is *stateful*—the signer needs with each signature to update and remember the number of signatures produced so far.

(b) Explain why an adversary can't use a signature for a given value t to produce a signature for a different value t'.

Version III

Version III is an extension of Version II, allowing arbitrary-length messages to be signed. The public-key setup is the same. To sign a message m (of arbtrary length), first compute the 256-bit hash $h(m)$ of m. Then sign each bit of $h(m)$ in turn, based on consecutive elements of the public key. Thus, the signature of an arbitrary-length message with the Version III scheme is 256 times as long as the signature using Version II of a single bit. The number of messages that can be signed with Version III is at most $n/256$, where n is the number of values in the public key. (The value t need only be given for the signature of the first bit; we can represent such a t in eight bytes.)

(c) How long is a Version III signature?

Version IV

Version IV is an extension of Version III, dramatically reducing the size of the public key. Instead of having n 256-bit values x_1, ..., x_n in the public key, where $n = 2^k$, the signer instead creates a Merkle tree of depth k with n leaves $x_1, \ldots, x_n$. The root R of the Merkle trees becomes the signer's public key; he remembers the whole tree, as well as the associated p's and q's, as his secret key. The signature is as in Version III, except that the signer also provides proof that the 256 leaves used in a signature are in fact leaves of the Merkle tree with root R.

(d) Suppose $k = 42$. How large is a Version IV signature? (Note that the Merkle trees proofs may allow some compression due to sharing of values.)

Version V

This is a stateless variant of Version IV. Instead of using a block of 256 consecutive leaves of the Merkle tree, Version V uses r *(pseudo-)random* positions, where r is a parameter of the system (less than 64).

Let $g(i, v)$ be a function that maps pairs (i, v) to random-looking values in the range 1 to $n = 2^k$. Here i is an integer in the range 1 to r and $v = h(m)$ is the 256-bit hash of the message being signed. The function g is pseudo-random; for example its output might be the last k bits of $h(i||v)$, interpreted as an integer in the range 1 to $n = 2^k$. The signature of message m in Version V uses the r Merkle tree leaves with indices $g(1, v)$, $g(2, v)$, ..., $g(r, v)$, instead of a block of 256 consecutive leaves; the i-th leaf used is for a one-bit signature for the i-th bit of $v = h(m)$. Note that values $g(i, v)$, which correspond to the value of t in the previous two versions, do not need to be included in the signature.

(e) Describe the length of the signature in terms of the parameters k and r. (Give your answer in terms of bits.) Compare to that of Version IV for $k = 42$ and $r = 16$.

(f) Show that for $r = 16$ the probability that all of the r leaves used for a new message signature were previously used in earlier signatures is at most 2^{-128}, if the number s of previously signatures is suitably restricted. What is the maximum s that works?

(g) Argue that the scheme provides a high degree of unforgeability, if the value of s is constrained as per your answer in the previous question.