CS 302 Computer Security and Privacy

Debayan Gupta

Lecture 12
March 28/April 2, 2019

Hashed Data Structures

- Motivation: Peer-to-peer file sharing networks
- Hash lists
- Hash Trees

Lamport One-Time Signatures

Merkle Signatures

Authentication Using Passwords

- Authentication problem
- Passwords authentication schemes
- Secure password storage
- Dictionary attacks

Hashed Data Structures

Peer-to-peer networks

One real-world application of hash functions is to *peer-to-peer* file-sharing networks.

The goal of a P2P network is to improve throughput when sending large files to large numbers of clients.

It operates by splitting the file into blocks and sending each block separately through the network along possibly different paths to the client.

Rather than fetching each block from the master source, a block can be received from any node (peer) that happens to have the needed block.

The problem is to validate blocks received from untrusted peers.

Integrity checking

An obvious approach is for a trusted master node to send each client a hash of the entire file.

When all blocks have been received, the client reassembles the file, computes its hash, and checks that it matches the hash received from the master.

The problem with this approach is that if the hashes don't match, the client has no idea which block is bad.

What is needed is a way to send a "proof" with each block that the client can use to verify the integrity of the block when it is received.

Block hashes

One idea is to compute a hash h_k of each data block b_k using a cryptographic hash function H.

The client *validates* b_k by checking that $H(b_k) = h_k$.

This allows a large *untrusted* data block to be validated against a much shorter *trusted* block hashes.

Problem: How does the client obtain and validate the hashes?

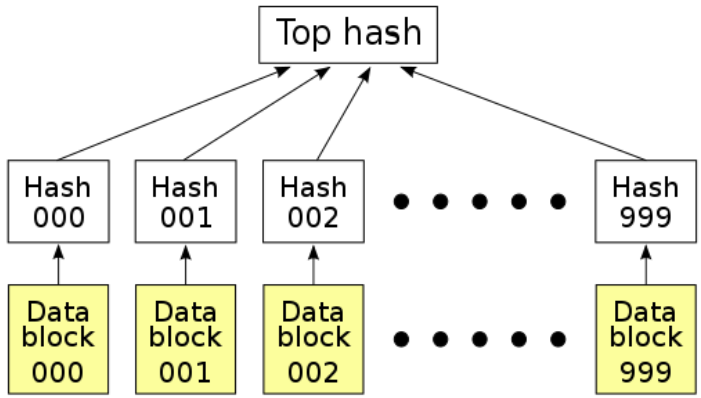
We desire a scheme in which a small amount of trusted data can be leveraged to validate the block hashes as well as the data blocks themselves.

Hash lists

A *hash list* is a two-level tree of hash values.

The leaves of the tree are the block hashes Hash_k.

The concatenation of the Hash_k are hashed together to produce a *top hash*.



From Wikipedia, "Hash List"

Using a hash list

The client receives top hash from the trusted source.

The client receives the list of Hash_k from any untrusted source.

The Hash_k are validated by hashing their concatenation and comparing the result with the stored top hash.

Each data block b_k is validated using the corresponding Hash_k.

Weakness of hash list approach

The main drawback of hash lists is that the entire hash list must be downloaded and verified before data blocks can be checked.

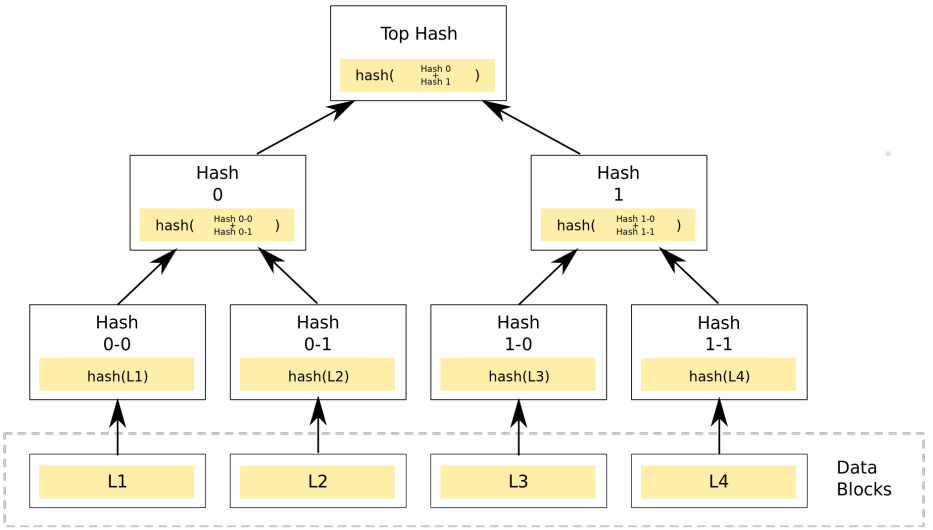
A bad Hash_k would cause a good b_k to be repeatedly rejected and refetched.

BitTorrent places all of the hashes in a single file which is initially downloaded from a trusted source.

Hash trees (Merkle trees)

Hash trees provide a way to attach a small untrusted *validation block* to each data block that allows the data block to be validated directly against a single trusted hash value *top hash*.

Neither the validation block nor the data block need be trusted; errors in either will be detected.



From Wikipedia, "Merkle tree"

Example: To validate L2, use Hash 0-0 and Hash 1 to compute Top Hash. Compare with trusted Top Hash. If they agree, can trust L2, Hash 0-0, and Hash 1.

Tree notation

A hash tree is a *complete binary tree* with $N = 2^n$ leaves.

Label the nodes by strings $\sigma \in \{0,1\}^*$, where σ describes the path from the root to the node.

The *root* is denoted by v_ε, where ε is the *null string*. Its two sons are v_0 and v_1.

The two children of any internal node v_j are denoted by v_{j0} and v_{j1}.

Let σ_k be the path from the root to the k^{th} leaf. Then v_{σ_k} denotes the leaf node corresponding to data block b_k.

Node values in a hash tree

Let v_τ be a node in a hash tree.

Define Hash_τ, the hash value at node v_τ:

- If v_τ is a leaf, then $\tau = \sigma_k$ for some k, and

$$\mathsf{Hash}_\tau = H(b_k).$$

- If v_τ is an internal node, then

$$\mathsf{Hash}_\tau = H(\mathsf{Hash}_{\tau 0} \cdot \mathsf{Hash}_{\tau 1}).$$

Companion nodes

Let v_τ be an internal node, and let $v_{\tau'}$ be its sibling in the tree. We say that τ' is the *companion* of τ. τ' is obtained from τ by flipping the last bit.

Example: The companion of 1011 is 1010 since v_{1011} and v_{1010} are the two children of v_{101}.

Validation block

The *validation block* B_k for data block k consists of the sequence of hash values $\mathsf{Hash}_{\tau'}$ for each companion τ' of each non-null prefix τ of σ_k.

Example: Let $\sigma_k = 1011$.

- The non-null prefixes of σ_k are 1011, 101, 10, 1.
- The corresponding companions are 1010, 100, 11, 0.
- The validation block is

$$B_k = (\mathsf{Hash}_{1010}, \mathsf{Hash}_{100}, \mathsf{Hash}_{11}, \mathsf{Hash}_{0}).$$

Block validation using hash trees

Validating data block b_k requires top hash and validation block B_k.

One proceeds by computing Hash_τ for each τ that is a prefix of σ_k, working in order from longer to shorter prefixes.

- If $\tau = \sigma_k$, then $\text{Hash}_\tau = H(b_k)$.
- Let τ be a proper prefix of σ_k. Assume w.l.o.g. that $\tau 0$ is a prefix of σ_k and $\tau 1$ is its companion. Then $\text{Hash}_{\tau 0}$ has just been computed, and $\text{Hash}_{\tau 1}$ is available in the validation block. We compute

$$\text{Hash}_\tau = H(\text{Hash}_{\tau 0} \cdot \text{Hash}_{\tau 1}).$$

Validation succeeds if $\text{Hash}_\varepsilon = \text{top hash}$.

Lamport One-Time Signatures

Overview of Lamport signatures

Leslie Lamport devised a digital signature scheme based on hash functions rather than on public key cryptography.

Its drawback is that each key pair can be used to sign only one message.

We describe how Alice uses it to sign a 256-bit message. As wtih other signature schemes, it suffices to sign the hash of the actual message.

How signing works

The private *signing key* consists of a sequence $r = (r^1, \ldots, r^{256})$ of pairs (r_0^k, r_1^k) of random numbers, $1 \leq k \leq 256$.

Let m be a 256-bit message. Denote by m_k the k^{th} bit of m.

The *signature* of m is the sequence of numbers $s = (s^1, \ldots, s^{256})$, where

$$s^k = r_{m_k}^k.$$

Thus, one element from the pair (r_0^k, r_1^k) is used to sign m_k, so $s^k = r_0^k$ if $m_k = 0$ and $s^k = r_1^k$ if $m_k = 1$.

How verification works

The public *verification key* consists of the sequence $v = (v^1, \ldots, v^{256})$ of pairs (v_0^k, v_1^k), where $v_j^k = H(r_j^k)$, and H is any one-way function (such as a cryptographically strong hash function). Thus, v comprises the hashes of all random numbers in the private signing key r.

To verify a signed message (m, s), Bob checks that $H(s^k) = v_{m_k}^k$ for each k.

Assuming all is well, we have

$$H(s^k) = H(r_{m_k}^k) = v_{m_k}^k$$

for all k, so the verification succeeds.

Forgery resistance

Lamport signatures are one-time because half of the private key is released when it is used.

The revealed r_j^k would allow anyone to construct a valid signature of the same message m, but that signature is already known and valid.

However, if other numbers of the private key are ever disclosed, then Eve could in general produce valid signatures of messages that have never been signed by Alice.

How to forge if the key is reused

Suppose Alice signs m and m' using the same private key.

Suppose further that m and m' differ in two bits j and ℓ. Then Eve can create a signature for the new message m'', where

$$m''_k = \begin{cases} m'_j & \text{if } k = j \\ m_k & \text{if } k \neq j \end{cases}$$

Example: $m = 01101$, $m' = 00100$. They differ in bit positions $j = 2$ and $\ell = 5$. Then Eve can produce a valid signature for $m'' = 00101$ even though Alice never signed m''.

Merkle Signatures

Merkle signature scheme

The Merkle signature scheme applies hash trees to Lamport signatures to allow a large number $N = 2^n$ of messages to be signed, yet each can be verified using the same short trusted verification key.

It works by creating N Lamport signing keys $R_1, \ldots, R_N$ and N corresponding verification keys $V_1, \ldots V_N$. The i^{th} message is signed using key pair (R_i, V_i).

Reducing the size of the trusted verification key

The problem here is the need for Bob to obtain and store the N verification keys.

Rather than distributing all V_i in advance, V_i is sent to Bob along with the i^{th} signed message since it is only needed to verify that one message.

But this creates the problem of validating V_i, since how does Bob know that V_i is genuine rather than a fake designed to make a forged signature look good?

Hash trees to the rescue

Validating received data is what hash trees are good at.

When Alice creates the N Lamport key pairs (R_i, V_i), she puts all V_i into a hash tree, computes top hash, and makes it public.

When she signs the i^{th} message m, she creates a Lamport signature s using the pair (R_i, V_i) as before.

She then sends Bob (m, s) along with V_i and B_i, where B_i is the validation block for V_i.

Verifying the Merkle signature

We assume Bob knows top hash. He validates the signed message $(m, (s, i, V_i, B_i))$ in two steps:

1. He uses the Lamport signature verification algorithm with verification key V_i to check that (m, s) is valid.
2. He checks that V_i is consistent with top hash.

Step 1 is done exactly the same as before, i.e., Bob checks that $H(s^k) = v^k_{m_k}$ for each k.

For step 2, Bob validates V_i against the hash tree by walking up the tree from leaf i to root, computing the hash of each node in turn. For this, he uses the hash values stored in B_i. He checks that the final hash value equals the stored top hash.

Authentication Using Passwords

The authentication problem

The *authentication problem* is to identify with whom one is communicating.

For example, if Alice and Bob are communicating over a network, then Bob would like to know that he is talking to Alice and not to someone else on the network.

Knowing the IP address or URL is not adequate since Mallory might be in control of intermediate routers and name servers.

As with signature schemes, we need some way to differentiate the real Alice from other users of the network.

Possible authentication factors

Alice can be authenticated in one of three ways:

1. By something she knows;
2. By something she possesses;
3. By something she is.

Examples:

1. A secret password;
2. A smart card;
3. Biometric data such as a fingerprint.

Passwords

Assume that Alice possess some secret that is not known to anyone else. She authenticates herself by proving that she knows the secret.

Password mechanisms are widely used for authentication.

In the usual form, Alice authenticates herself by sending her password to Bob.

Bob checks that it matches Alice's password and grants access.

This is the scheme that is used for local logins to a computer and is also used for remote authentication on many web sites.

Weaknesses of password schemes

Password schemes have two major security weaknesses.

1. Passwords may be exposed to Eve when being used.
2. After Alice authenticates herself to Bob, Bob can use Alice's password to impersonate Alice.

Password exposure

Passwords sent over the network in the clear are exposed to various kinds of eavesdropping, ranging from ethernet packet sniffers on the LAN to corrupt ISP's and routers along the way.

The threat of password capture in this way is so great that one should *never* send a password over the internet in the clear.

Some precautions

Users of the old insecure Unix tools should switch to secure replacements such as ssh, slogin, and scp, or kerberized versions of telnet and ftp.
Do any of you even know what the insecure tools were called?

Web sites requiring user logins generally use the TSL/SSL (Transport Layer Security/Secure Socket Layer) protocol to encrypt the connection, making it relatively safe to transmit passwords to the site, but some do not.

Depending on how your browser is configured, it will warn you whenever you attempt to send unencrypted data back to the server.

Password propagation

After Alice's password reaches the server, it is no longer the case that only she knows her password.

Now the server knows it, too!

This is no problem if Alice only uses her password to log into that *that particular* server.

However, if she uses the same password for other web sites, the first server can impersonate Alice to any other web site where Alice uses the same password.

Multiple web sites

Users these days typically have accounts with dozens or hundreds of different web sites. (I personally have over 300.)

The temptation is strong to use the same username-password pairs on all sites so that they can be easily remembered.

But then anyone with access to the password database on one site can log into Alice's account on any of the other sites.

Typically different sites protect data of very differing sensitivity.

An on-line shopping site may only be protecting a customer's shopping cart, whereas a banking site allows access to a customer's bank account.

Password policy advice

My advice is, use a different password for each account.

Of course, nobody can keep dozens of different passwords straight, so the downside of my suggestion is that the passwords must be written down and kept safe, or stored in a properly-protected password vault.

If the primary copy gets lost or compromised, then one should have a backup copy so that one can go to all of the sites ASAP and change the passwords (and learn if the site has been compromised).

The real problem with simple password schemes is that Alice is required to send her secrets to other parties in order to use them. We will later explore authentication schemes that avoid this.

Secure password storage

Another issue with traditional password authentication schemes is the need to store the passwords on the server for later verification.

- The file in which passwords are store is highly sensitive.
- Operating system protections can (and should) be used to protect it, but they are not sufficient.
- Legitimate sysadmins might use passwords found there to log into users' accounts at other sites.
- Hackers who manage to break into the computer and obtain root privileges can do the same thing.
- Backup copies may not be subject to the same system protections, so someone with access to a backup device might read everybody's password from it.

Storing encrypted passwords

Rather than store passwords in the clear, it is usual to store "encrypted" passwords.

That is, the hash value of the password under some cryptographic hash function is stored instead of the password itself.

Using encrypted passwords

The authentication function

- takes the cleartext password from the user,
- computes its hash value,
- compares the computed hash value with the stored value.

Since the password file does not contain the actual password, and it is computationally difficult to invert a cryptographic hash function, knowledge of the hash value does not allow an attacker to easily find the password.

Dictionary attacks on encrypted passwords

Access to an encrypted password file opens up the possibility of a *dictionary attack*.

Many users choose weak passwords—words that appear in an English dictionary or in other available sources of text.

If one has access to the password hashes of legitimate users on the computer (such as is contained in `/etc/passwd` on Unix), an attacker can hash every word in the dictionary and then look for matches with the password file entries.

Harm from dictionary attacks

A dictionary attack is quite likely to succeed in compromising at least a few accounts on a typical system.

Even one compromised account is enough to allow the hacker to log into the system as a legitimate user, from which other kinds of attacks are possible that cannot be carried out from the outside.

Salt

Salt is a way to make dictionary attacks more expensive.

- *Salt* is a random number that is stored along with the hashed password in the password file.
- The hash function takes two arguments, password and salt, and produces a hash value.
- Because the salt is stored (in the clear) in the password file, the user's password can be easily verified.
- The same password hashes differently depending on the salt.
- A successful dictionary attack now has to encrypt the entire dictionary with every salt value that appears in the password file being attacked.
- This increases the cost of the attack by orders of magnitude.