Aryan Nath December 27, 2024

CS-1306 / MAT-1201 Final

Collaborators: *none*

Problem 1

Requirements:
1. Receipt generation 2. Threshold, the LE should not be able to read a message until they get k reports of it. 3. Prevent the same (anonymous) person from repeatedly reporting the same message.

Assumptions:
1. The service provider and the law enforcement are semi-honest, however the service provider is vulnerable to faults in its public key database. 2. Users are malicious and can collude.

Detailed protocol description:

I will start with the whatsapp (signal) protocol that we have discussed in class and then modify it to address our requirements.

1. Let's address the first requirement of the protocol, receipt generation. We need to do this while maintaining the anonymity of the user so we can't do something as simple as passing the phone number of the user, unencrypted, along with the report. We also need to make sure that no user can use someone else's receipt.

   In the unmodified whatsapp protocol, the message sender, say user u_1, generates $M = < m, sk_i, pk_i, \sigma_i >$, the same M is forwarded by all message receivers (with the assumption that the message sender is generating 1000 pairs of sk_i, pk_i). We want to modify this protocol such that when another user, say user u_{73} reports M (without modifying M itself), then the receipt generated can only be used by u_{73} and the user receives an acknowledgement that the law enforcement has received the message. The simplest way to trace the receipt back to u_{73} would be to use u_{73}'s phone number (without actually revealing u_{73}'s identity). Since, the users are malicious, we want to make sure that no other user can use this receipt in place of u_{73}. So we introduce digital signatures. Before reporting to the law enforcement u_{73} signs M with their secret key

$$sk_{73} = [\text{random number}]_{K-10} \| [\text{u}_{73}\text{'s phone number}]_{10}$$

and generate a public key pk_{73} using the RSA digital signature scheme. On receiving the pair $< M, sign(M) >$, the law enforcement signs sign(M) with its own signature s_{SKLE} and returns the receipt $\gamma = RSA_{encrypt}(M_{u_{73}} \mid t)$, where t is the timestamp, to the user u_{73}. Now suppose u_{73} gives their secret key to u_{50} and u_{50} generates the receipt in u_{73}'s name. When the receipt is being processed, the law enforcement can simply ask u_{50} to show them the secret key, which we know will contain u_{73}'s phone number and not u_{50}'s. Since, both sk_{73} and s_{SKLE} are used and the LE just signs the second in the pair $< M, sign(M) >$, a malicious user should not be able to gain anything by swapping M and $sign(M)$ in $< M, sign(M) >$. With this protocol, u_{73}'s anonymity is maintained until the receipt is processed by the LE, and this setting is secure against malicious users (Note: this part only addresses a single user reporting a message, I will soon combine it with the protocol for using a threshold on the reports). This protocol assumes that the LE (or SP if the LE does not maintain a database) discards the receipt from its storage once the report has been processed so the same receipt cannot be used twice.

2. Now let's address the second part of the protocol, that is, the LE should only be able to read a message after they get k reports for it. Based on what we have learnt in the course, the concept of a threshold being linked to unlocking a secret is mostly related to Shamir's secret key sharing. As done in the whatsapp protocol, the message sender generates $M =< m, sk_i, pk_i, \sigma_i >$. But now, the service provider also generates a key K, associated uniquely with σ_i that is not given to u_1 to prevent any form of collusion (I don't see any scenarios in which u_1 will collude but not giving them K will avoid any scenarios I could not think of). The service provider generates k shares for the key K. Now, the service provider has stored in its database a function for generating the shares (a new function will be required for each key), and pairs (σ_i, x_i), for $i = 1, \ldots, k$ (Note: I am assuming that it is very unlikely that the same person will get reported for two different messages within the time the LE gets access to m). Now, each time a reporter sends the pair $< M, sign(M) >$ through some gateway like TOR, the message is first received by the service provider and encrypted by it using the K associated with σ_i (can be easily deterministically generated using σ_i which is already there in M. So K can either be generated each time if feasible or stored). Then the SP sends $< M_K, sign(M), (f(x_i, x_i) >$ to the LE where $M_K =< AES_K(m), sk_i, pk_i, \sigma_i >$. I'm assuming that the LE has a smaller database storing the shares it receives from the SP for every message that is reported (the message LE receives is M_K and the LE knows that it cannot read it yet). So, only once the LE has k shares associated with σ_i (which is uniquely associated with the message sender u_1), the LE can get K and decrypt $AES_K(m)$ to finally read the message m. Note that the size of the database required for this is equal to the number of users the SP has queried and already has them stored in its database. In this protocol I am assuming that the LE and SP are semi-honest and the LE will not be able to get K from the SP (as then this will be the same as the LE processing each report despite its resource restriction). Hence, we

have ensured that the LE is only able to read, and hence process the message m once it receives k reports of it, without disturbing the protocol for generating a receipt. Also, we are assuming the LE is semi-honest so it will not collude with any user and get access to the message m. Users already have access to the SP for getting their σ_i's so this will not require any additional server or change in the whatsapp protocol except the reporting step. To improve this a bit, the shares can be stored in a stack and a counter can be used against the number of reports made per message at the SP (stored separately from the key shares) to confirm that no one has had malicious access to the SP's database of key shares.

3. Now, since shares are related to σ_i and not the reporting user, an issue that naturally comes up is that no user should be able to report the same message twice. In our protocol we already know that when reporting, the user say u_{73} is sending $< M, sign(M) >$ to the SP. Since, each message can be uniquely identified using $(m\|\sigma_i\|pk_i)$ and each user (based on part (1)) can be uniquely identified using $sign(M)$, and we have already proved $< M, sign(M) >$ to be fool proof against malicious reporting (assuming that the users cannot successfully modify M without access to the SP's database, as if the reporters just modify m, then the original message sender can deny it using their corresponding $< sk_i, pk_i, \sigma_i >$), we can combine this to maintain a check against duplicate reports. So, after every report, the SP computes $T_{u,m} = H(m\|\sigma_i\|pk_i\|sign(M))$, if a duplicate is found in the database then the SP is alerted and rejects the report.

   To make our protocol secure against faults in the public key database of the SP, we can assign a counter to each σ_i, sk_i, pki_i triplet and add $\sigma_i' = H(\sigma_i\|counter)$ to this triplet.

Problem 2

I'll begin my answer by first addressing what kind of malicious activity can take place on the server and user's side post payment.

User side malicious activities:

1. Attempting to overwrite or modify other user's files.
2. Attempting the de-anonymize other users.
3. Trying to exploit the file system to gain unauthorized access.
4. Uploading harmful or illegal content.

Server side malicious activities:

1. Tampering file contents.
2. Selectively censoring or deleting files.
3. Tracking or de-anonymizing users.
4. Breaking file confidentiality.
5. Violating agreed upon download periods.
6. Attempting to extract payment information, leading to de-anonymizing the file uploader.

Now I'll describe the minimal assumptions, in addition to a secure setup before the payment is done, that are required to enable the file hosting service with a malicious server post payment.

1. Encrypted file storage:

   The files should be encrypted before they are stored in the server, to guarantee that the file contents are only accessible to users with the public key (availed through a download link that cannot be used by the server itself).

2. File integrity:

   The system uses digital signatures to make sure that the server cannot modify the file contents and place it under the name of the original uploader. A secure form of digital signatures would place a signature of the file uploader on the hash of the file contents before the message is uploaded. Anyone with the user's public key (availed through the download link) can verify the signed hash $sign(H(F))$ against the hash of the file contents. The system should use a well known hash function such as SHA-256 that cannot be modified by the server without being detected by the service users.

3. Anonymous authentication:

   Once the users have made a payment, they need to be authenticated before they can upload their files, while maintaining their anonymity. This is done in two steps:

   (a) Payment and token issuance:
   The user pays the service via the payment processor (a trusted third party), and generates a commitment C

   $$C \equiv g^m h^r \pmod{p}$$

   where g, h are public generators of a cyclic group of order p, m is the payment message (eg. payment ID), and r is a random value to ensure the commitment is hiding (based on the **Pederson commitment** scheme), blinds it and sends it to the payment processor. The payment processor then verifies the payment and signs the blinded commitment using a blind signature scheme such as RSA blind signatures, let's call this signature $\sigma = S(C)$, and sends it back to the user. This enables the user to have an unlinkable proof of payment (C, σ).

   (b) Proving eligibility: Now when the user wants to upload a file, they prove to the server that they posses a valid token (C, σ) without revealing m or r. The user and the server engage in an interactive Zero-Knowledge Proof protocol that verifies that (1) the commitment C is both hiding and binding. Hiding means C does not reveal any information about the committed value. Binding means once C is generated the user cannot later change the committed value. (2) The signature σ on C is valid. (3) C was issued by a trusted third party and not forged (happy to discuss the zero knowledge proof during the viva, not including here because of space constraints).

4. Time bound access:

   The system needs to ensure that files are only accessible during the period determined based on the payment. The malicious server may collude with a user and increase the upload time, or for another user decrease the upload time. To prevent this, we can use a smart contract for making sure that the file is not deleted pre-maturely and a time-locked mechanism through which the decryption key K is available only within the allotted time, after which it doesn't make sense for the server to keep it hosted.

(a) Preventing pre-mature deletion using smart contracts:

Smart contracts are self-executing programs running on a blockchain, and can be used to enforce rules around file retention. They are immutable and tamper-proof. Before a payment is made, a file hash and the agreed retention time T is sent by the uploader to the smart contract. The server then makes a security deposit into the smart contract. If the user is not verified post payment, then the contract is withdrawn and the server should get back its security deposit, so the smart contract should begin only once the Zero-Knowledge Proof from the previous step has completed (the smart contract should be linked to the file hash and not the file uploader). The server is semi-honest here so it will send the correct details. The server then has to periodically submit a proof of storage (eg. the file hash) and a proof of retrievability (proof of retrievability can be done by splitting F into blocks and tagging them using K, which the server does not have direct access to. The smart contract then randomly selects block indices with a random coefficient for each and expects the server to be able to produce the required linear combination of the blocks. Assuming once the file is deleted, the tags are also deleted). In case the server deletes the file prematurely, an amount from the security deposit is forfeited.

(b) Preventing file retention after time period T:

We can use a semi-honest third party that provides the decryption key K only to users who download the file (and not the server). Setting up the download link can be done before the payment has been fully processed to ensure that the server is semi-honest. After time T, the third party deletes the key and no one can access the file contents anymore. So it would not make sense for the server to retain the file on the hosting service. I don't think using a third party can be avoided as using a cryptographic puzzle with sequential requests from the server does not guarantee that the server will make all requests within time T. This link can be signed by the third party to make sure that it has not been replaced by the server.

5. No collusion between a user and the server:

It is already guaranteed that the user cannot make the server change another user's file contents from the hash based digital signature that we have set up. If the malicious user prompts the server to delete another user's upload then that user's smart contract will enforce a penalty on the server. The user cannot de-anonymize a user since the encrypted files are not linked to uploader and cannot get unauthorized access to the files since they are only retained by the sever during the smart contract period (Note: the server stores encrypted files, so getting access to the server's storage will not help).

Problem 3

I'll begin this problem with a brief description of the assumptions Im making for UPI and UPI lite based transactions. UPI uses a central server managed by the NPCI, a two-factor authentication system and verification of transactions by the payer's and payee's bank servers. UPI lite on the other hand, allows offline small-value transactions. Payments in UPI lite are processed using a local wallet stored on the user's device without involving the central server for each transaction. The server's role in UPI lite is limited to initial wallet top-ups and settlement reconciliation.

Now let's discuss what problems can arise while adding a receipt mechanism:

1. Replay attacks:

   A malicious user could replay receipt messages to claim the transaction was re-initiated or completed multiple times.

2. Non-repudiation:

   The payer or merchant could deny having sent or received a receipt.

3. Integrity:

   The receipt could be tampered with.

4. Offline constraints:

   Since we also want to address UPI lite, for which there is no server to mediate disputes or provide a trusted timestamp.

5. Infinite acknowledgement loops:

   Without a clear termination mechanism, the acknowledgement process could loop indefinitely.

I am assuming that the payer and receiver can know each other's identity. If the verification is taking place without a server then the bank accounts will not get to know the users' identity until the wallet top-up takes place.

Now, I will address each of these problems before finally describing my receipt mechanism:

1. Replay attacks:

   Solution: A unique transaction ID (TID) will be assigned to each transaction and included in the receipts. This will ensure that each receipt is tied to a single receipt, as duplicate receipts will have a duplicate TID.

2. Non-repudiation:

   Solution: We will use digital signatures with the receipt so that neither the payer not merchant can deny the transaction. The public keys can be derived from something known like their phone numbers (and using RSA modulo n to get the secret key), to remove server involvement. The signature will be on the transaction details $T =$ {TID, Amount, Merchant Info, Timestamp}.

3. Integrity:

   Solution: We can use the message integrity mechanism with hash functions that we discussed in class, using a hash chain. The payer will sign $H(T)$ and send it along with T to the merchant. Then the merchant will sign $H(sign_{payer}H(T))$ with their own secret key and send it along with $sign_{payer}H(T)$ in the acknowledgement to the payer. This signing process will be repeated once more to complete the hash cycle. If during this cycle, any party changes the contents of the acknowledgement, then based on the hash based digital signature scheme, this change will be detected. This will also ensure that the TID is not altered, so that replay attacks can be prevented without the use of a nonce to uniquely link the transaction.

4. Infinite Acknowledgement loops:

   The hash chain described for preventing replay attacks will be used to define a clear termination of the acknowledgement mechanism. Checks can be made against the data held by the two parties individually to determine when $H(H(H(T)))$ has been generated.

All of these steps can be done offline based on the transaction details, without requiring any use of the NPCI server.

In summary, the entire receipt generation mechanism is as follows:

1. Payer initiates transaction:

   The payer creates the transaction details

   $$T = \{TID, Amount, Merchant\ Info, Timestamp\}$$

   then hashes it

   $$H(T)$$

   and signs the hash using their secret key generated using RSA modulo on their public key pk_{payer} (which is linked to their phone number and can be derived by the merchant without accessing the NPCI server). The payer then sends:

   $$sign_{payer} = sign_{sk_{payer}}(H(T))$$

   along with T and $H(T)$ to the merchant.

2. Merchant acknowledges the transaction:

   The merchant first verifies the payer's signature

   $$sign_{pk_{payer}}(sign_{sk_{payer}}(H(T))) = H(T)$$

   then generates the hash

   $$H(H(T))$$

   signs it and sends $H(T), H(H(T)), sign_{sk_{merchant}}(H(H(T))$ to the payer.

3. Payer acknowledges the receipt:

   The payer the verifies the merchant's acknowledgement using

   $$sign_{pk_{merchant}}(sign_{sk_{merchant}}(H(H(T))) = H(H(T))$$

   If the sign is valid, then the payer responds by sending a final acknowledgement with

   $$sign_{sk_{payer}}(H(H(H(T))))$$

   along with $H(H(T))$ and $H(H(H(T)))$. Note that the payer can detect if $H(T)$ and $H(H(T))$ have been switches as they already have T and H and can compute both the items.

4. Merchant finalizes the receipt:

   The merchant verifies the acknowledgement of the payer

   $$sign_{pk_{payer}}(sign_{sk_{merchant}}(H(H(H(T)))) = H(H(H(T)))$$

   which if valid, both parties have agreed on the receipt, and the transaction has been completed.