7. Computing on Untrusted Servers (Encrypted Databases - CryptDB)

Naranker Dulay

n.dulay@imperial.ac.uk
https://www.doc.ic.ac.uk/~nd/peng

Outsourcing Databases to the Cloud

- Amazon AWS
- Microsoft SQL Azure
- Google App Engine
- Apple iCloud

- Business database needs are growing by over 50% per year.
- Costs of database management are increasing as well.
- Database outsourcing is now a viable choice for many businesses
- But should organisations trust the database service provider to protect the confidentiality of personal data (financial, medical, mail/messages etc) or company data (financial, operational, research, security, business plans etc)?

Threats

- **A cloud database provider might** breach their data processing agreement, for example:

  Pass data to another party (***Theft***) or to a government agency (***Lawful Theft***)

  Deny access to the database in a dispute e.g. over payment/quality of service.

  Use other providers for services (e.g. storage) without permission from users.

  Not have adequate security measures and gets hacked.

  Keep changing terms and conditions ...

- **A privileged user (or hacker) who was malicious might be able to:**

  Read (copy) data from Disk/RAM. Modify/delete data. Swap/Delay/Replay data. Manipulate access controls. Observe who accesses which data when and from where. Etc....

▶ A good solution should protect against such attacks ◀

Why not encrypt the database?

We could encrypt/decrypt all attributes at the ***client*** side. But what if we need a few tuples (rows) from relations (tables) with millions of tuples (rows)?

```
select * from Purchases where Bought between today-6 and today;
```

How might this be executed?

Why not encrypt the database?

We could encrypt/decrypt all attributes at the ***client*** side. But what if we need a few tuples (rows) from relations (tables) with millions of tuples (rows)?

```
select * from Purchases where Bought between today-6 and today;
```

We might:

(1) **Fetch ALL** encrypted `Bought` dates and primary keys from the Server.

(2) Decrypt `Bought` dates

(3) Find tuples matching the `where` clause

(4) Fetch the remaining attributes for the matched tuples from the Server using their Primary Keys

(5) Decrypt the fetched attributes

For this particular example, we could perform a `select` on 7 encrypted dates.

Encrypted Queries

Why not encrypt the data and encrypt the query?

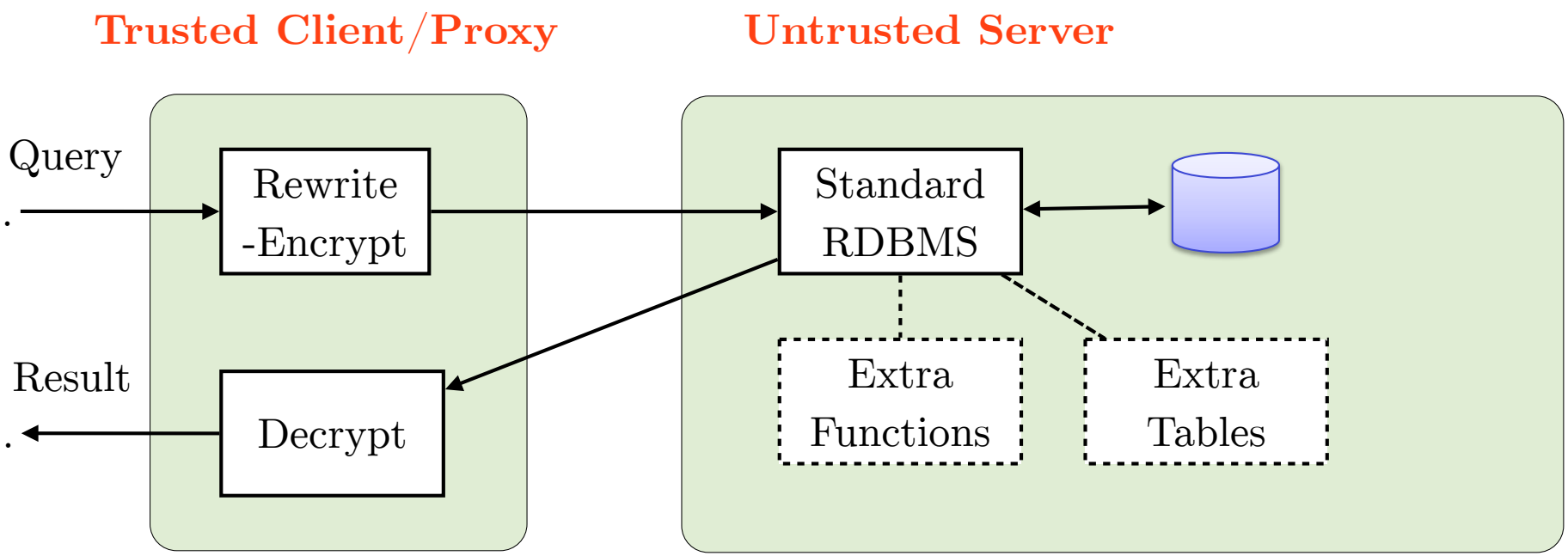
What we'd like to have, is the DBMS perform an encrypted query on the encrypted data without decrypting the data or query (c.f. FHE)

This is not easy in general since DBMSs are so feature rich.

For example a DBMS supporting encrypted processing would need to

- Handle numerous datatypes and operators — ints, strings, booleans, floats, decimals, arithmetic, comparisons, aggregate functions, sorting, ...
- Support transactions, access control, schema updates, etc.
- Keep memory, storage and time overheads low.
- Prevent leakage and traffic analysis, while ensuring integrity and freshness, etc.

CryptDB - Overview



In CryptDB a query like

```
select * from staff where salary >= 50000;
```

is rewritten to a query like

```
select * from table7 where col4 >= x738D3AB63;
```

with the returned results decrypted by the client

Notes

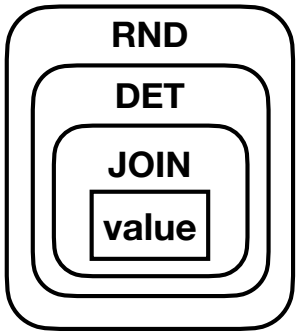
- CryptDB was the first encrypted database system that supported encrypted SQL operations on encrypted data.
- CryptDB provided efficient encryption schemes for a number of databases operations and ensured that different encryption schemes could co-exist without compromising security. Assumed an honest-but-curious adversarial model.
- Database operations implemented include: equality, comparison/sorting (ordering), sums and equi-joins.
- Designed as library of client-side and server-side code that could be easily added to SQL database management systems - versions for MySQL and PostgreSQL were implemented.
- MIT ran CryptDB live for a number of WordPress websites.
- CryptDB-MySQL throughput for the TPC-C database benchmark was ~25% worse than plain MySQL. Disk space usage was 5 times higher.

- In CryptDB, client SQL queries are sent from a trusted client application or via a trusted proxy.
- The trusted client application/proxy rewrites the query into a new SQL query that changes the names of tables and columns and encrypts constant values like integers.
- On receipt, CryptDB's server-side functions and additional tables (relations) perform the query and return an encrypted result back to the trusted client/proxy
- The trusted client/proxy decrypts the result to get the plaintext result.
- Of course it's easy for a human adversary to guess what data an obfuscated table name might represent. Also common constants used in queries might be guessable.
- Note: CryptDB did not protect the number of rows, the number of columns, approx size of data in a database.

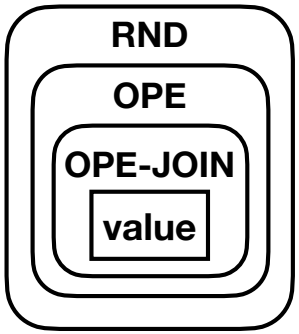
Encryption Onions

In order to provide basic SQL functionality, CryptDB employs different property-preserving encryption schemes (PPEs) and a clever approach to 'wrapping' and 'unwrapping' encryption layers.

Equals Onion

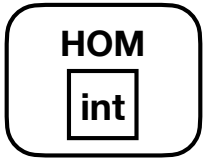


Order Onion



Search Onion

Addition Onion



Notes

- CryptDB uses **onions of encryption** - a stack of different encryption schemes (->multiple encryption)
- RND, DET, JOIN, OPE, OPE-JOIN, SEARCH, HOM are thedifferent encryption schemes.
- In the **EQUALS onion**, the plaintext value is first encrypted using JOIN, then with DET, and then with RND. Outer layers have stronger encryption properties. DET and JOIN can be combined into a single encryption layer.
- In the **ORDER onion**, the plaintext value is first encrypted using OPE-JOIN, then with OPE, and then with RND.
- The **SEARCH onion** and the **ADDITION onion** use a single encryption scheme, SEARCH and HOM respectively.
- Administrators can mark attributes "sensitive" (to use strong security) or "best-effort" (for less strong security)
- CryptDB does not address integrity, freshness or completeness of results.

- **Random** (**RND**). Ensures that the same value used in different places in the database encrypts to different ciphertexts. Implemented using AES & Blowfish.
- **Deterministic** (**DET**). Used for EQUALS(=), Equi-Joins, GROUP BY, COUNT, DISTINCT. Different keys are used for DET to prevent cross-column correlation. Based on AES.
- **Order preserving encryption** (**OPE**). Used for ordering. Ensures that if **x<y then OPE(x) < OPE(y)**. Also supports MIN, MAX, ORDER BY.
- **Homomorphic Encryption** (**HOM**). Used for Addition, Sum, Average. Ensures that $\mathbf{HOM_k(x) * HOM_k(y) = HOM_k(x+y)}$. Implemented using Paillier cryptosystem.
- **Join** (**JOIN** and **OPE-JOIN**). Needed because different keys are used for DET to prevent cross-column correlation.
- **Word search** (**SEARCH**). Supports Keyword search - LIKE.

Encrypted Tables & Queries

Student

ID	NAME
32	Alice

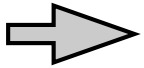


table7

c1_equals	c1_order	c1_add	c2_equals	c2_order	c2_search
84ab9a...	e98a..	54f3..	2445c...	abde8..	7acd..

```
update table7 set c2_equals = decrypt_RND(key, c2_equals)
```

Notes

- Each column(attribute) is mapped to one or more **onions** depending on application requirements.
- In the example, ID is mapped to
  (1) an equals onion (c1_equals),
  (2) an order onion (c1_order), and
  (3) an add onion (c1_add).
- NAME is mapped to
  (1) an equals onion (c2_equals),
  (2) an order onion (c2_order), and
  (3) a search onion (c2_search).
- Attribute values are shown as ciphertexts (e.g. 84ab9a...).
- Table (relation) names and column (attribute) names are replaced by CryptDB by less obvious names. For example might become Student becomes table7, ID might become c1.

- Each onion is initially encrypted upto the outer encryption scheme.
- To perform an operation on an attribute CryptDB will decrypt an onion down to the appropriate encryption scheme.
- In the example, the SQL **update** query, decrypts (unwraps) NAME's equals onion (c2_equals) from the **RND** ciphertext down to the **DET** ciphertext as preparation to one or more DET-related SQL operations.

  decrypt_RND is a CryptDB function installed on the DB server. **key** is derived from a master key for the table, column, onion, layer using AES.

Example

If a client requested Alice's ID, they might use:

```
select ID from Student where NAME='Alice'
```

this would be rewritten by the trusted client/proxy to

```
update table7 set c2_equals=decrypt_RND(K_t7,c2,Equals,RND, c2_equals)

select c1_equals from table7 where c2_equals=d3ac...
```

Notes

- In the example the SQL server will first decrypt(unwrap) the c2_equals RND ciphertext to the DET ciphertext.
- The do a search for the row(tuple) where c2_equals equals **d3ac...**. Here we assume that **d3ac...** is the double-encryption of 'Alice' first with key
  Kt7,c2,Equals,JOIN then with key
  Kt7,c2,Equals,DET
- The SQL server will return the c1_equals ciphertext of the selected row to the client/proxy
- The client/proxy will decrypt the return value using key:
  Kt7,c1,Equals,RND then with
  Kt7,c1,Equals,DET finally with
  Kt7,1,Equals,JOIN

- On completion of the encrypted search query the value of c2_equals is now only doubly-encrypted.
- Instead of immediately re-encrypting or restoring back to RND. CryptDB attempts to intelligently keep attributes at frequently accessed encryption levels, and re-encrypting based on some policy. This helps with performance.
- Operators like SUM, and + are rewritten as server side function calls, e.g. using the HOM functions.
- As we can see with this example, the management of keys is a crucial aspect of the CryptDB's implementation.

Example continued

If the next query is

```
select COUNT(*) from Student where NAME='Bob'
```

a client can use knowledge of the previous decryption to rewrite the query to

```
select COUNT(*) from table7 where c2_equals=34bd...
```

Here 'Bob' (ciphertext 34bd...) is encrypted with the key

$K_{t7,c2,Equals,JOIN}$ and then with the key:

$K_{t7,c2,Equals,DET}$

More Encrypted Databases

- **Property-preserving encryption**:
  CryptDB, Cipherbase, Monomi, ...
  Google Encrypted BigQuery
  Microsoft Always Encrypted v1 (included in SQL Server 2016)
  ...
- **Trusted Execution Environments (Intel SGX, FPGAs)**
  EnclaveDB (Imperial), StealthDB, ObliDB, EncDBDB,
  Microsoft Always Encrypted v2 (included in Azure SQL Database & SQL Server 2019),
  ...
- **Garbled circuits**:
  Arx DB (Tested on ShareLaTex service using MongoDB)

8. Computing on Untrusted Servers (Multi-user Keyword Search)

Naranker Dulay

n.dulay@imperial.ac.uk
https://www.doc.ic.ac.uk/~nd/peng

Multi-User Databases

CryptDB is a nice approach for many applications/scenarios but

- It was based on **secret keys shared by clients**.
- What if we need to **revoke** (change) the key, e.g. remove a user?
- We'd have to re-encrypt all the data at the database. This would be very inefficient.

Multi-User Databases

For multi-user databases we'd like to

- Allow multiple users to insert/update/search/read data
- Without sharing keys
- Easily add or revoke users without re-encrypting data.
- Use keys to authenticate users.

Shared and Searchable Data Spaces (Imperial)

- New users can be added easily. Keys can be revoked without affecting other users
- Keys are not shared. Keys can also be used to authenticate users.

RSA-based Proxy re-encryption scheme where

- Data encrypted by Alice is re-encrypted by the server (proxy).
- Re-encrypted data can be decrypted by Bob.
- Server and Bob don't know Alice's keys.
- Server is assumed to be **honest-but-curious** i.e. executes instructions faithfully but may be curious about what's stored or being searched for.
- Clients are assumed to be trusted.
- A trusted key management service generates key material for users.

RSA Proxy Key Construction

A trusted key management service constructs a master RSA Key pair (e, n) and (d, n) and then:

For each user A, the key management service constructs two new key pairs from the master RSA key pair, one pair for the user, one for the DB server:

User Key Pair 1. Ae1=$(e1, n)$ and Ad1=$(d1, n)$ is sent to the user

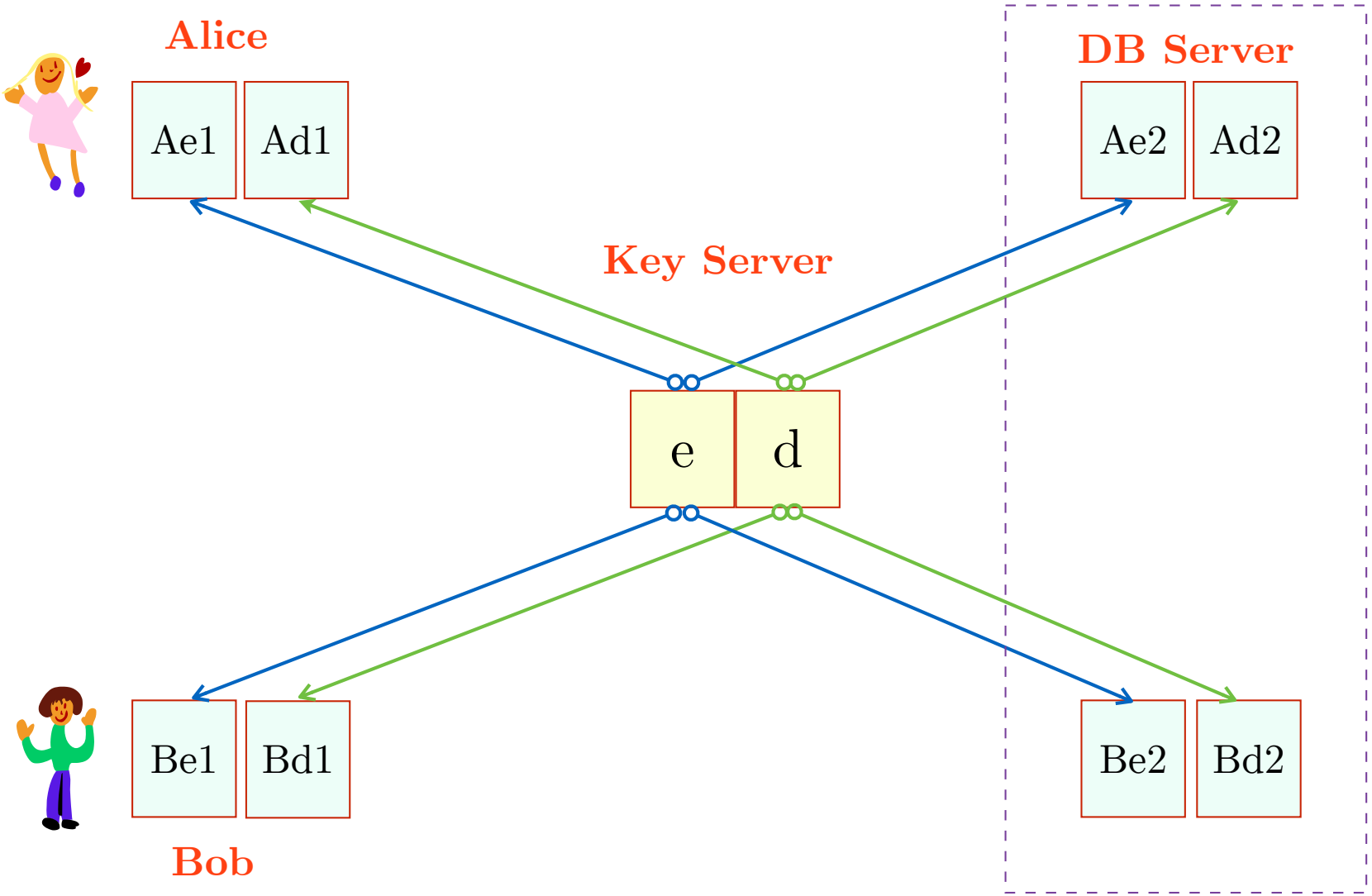
DB Key Pair 2. Ae2=$(e2, n)$ and Ad2=$(d2, n)$ is sent to the DB

such that

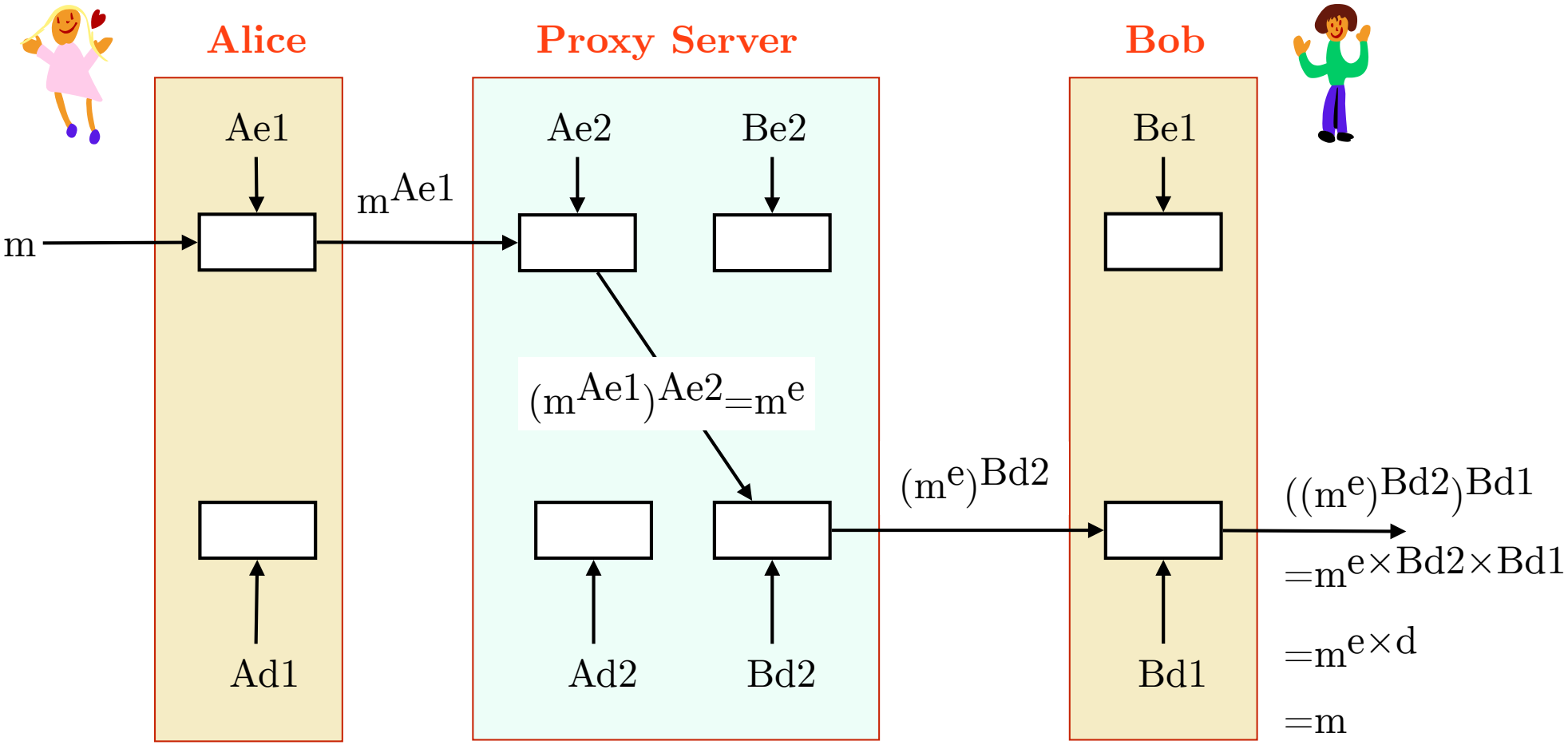
$$e1 \times e2 = e \mod (p-1)(q-1)$$

$$d1 \times d2 = d \mod (p-1)(q-1)$$

Key Generation



Encryption - Decryption



Example - Encrypted Keyword Search

- Alice encrypts the text to be stored with a symmetric key **k**.
- Alice then encrypts the symmetric key **k** with her public key **Ae1**
- Alice also hashes any keywords in the text and encrypts each hash value with **Ae1**
- Alice sends the (i) encrypted text, (ii) encrypted key and (iii) encrypted hash keywords to the DB server.
- DB server re-encrypts the encrypted key and encrypted hash keywords with **Ae2** (these are effectively encrypted with a master RSA encryption key e) and stores them, along the encrypted text (note - the encrypted text doesn't need to be re-encrypted by the DB server)

- Bob hashes his search keyword encrypts it with his public key **Be1** and sends to the DB Server as a query.
- The DB server re-encrypts Bob's encrypted hashed keyword again with **Be2**. The query is now effectively $\mathbf{hash(keyword)}^{e}$.
- The DB server does a search on the query to retrieve the matching encrypted text plus its encrypted symmetric key.
- The DB server then decrypts the doubly encrypted key **k** with **Bd2** and sends it to Bob along with the encrypted text.
- Bob decrypts the key again with **Bd1** to get **k** and then uses **k** to decrypt the encrypted text.

9. Computing on Untrusted Servers (Location Sharing - Longitude)

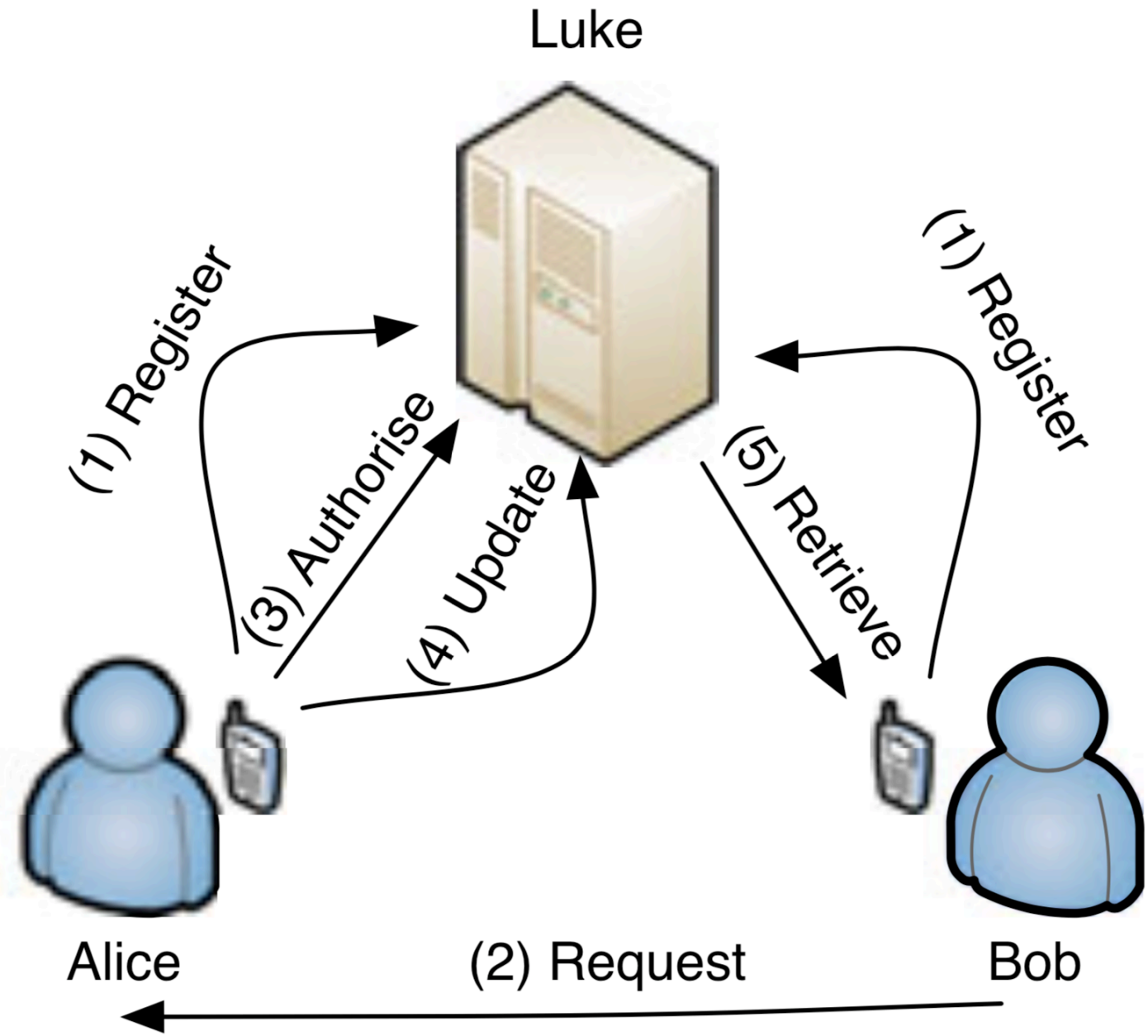
Naranker Dulay

n.dulay@imperial.ac.uk
https://www.doc.ic.ac.uk/~nd/peng

Longitude: Privacy-preserving location sharing

- Privacy-preserving location ***sharing*** service for mobile applications. c.f. Apple's Find My Friends and Google's long-abandoned Latitude service.
- Alice shares her location with Bob via a location sharing provider Luke (who is honest-but-curious).
- Alice defines the degree of precision for Bob, plus at what times Bob can see her location.
  - *What sharing policies should be supported?*
  - *What if Alice shares her location with Bob at full precision and David, her boyfriend, only at 1KM accuracy? What if David finds out?*
- Most computational cryptographic tasks are performed by Luke (saves communication and battery). Alice's cryptographic material can be pre-computed when her phone is charging.
- Alice only encrypts new location data once, regardless of the number of friends she shares her location with (this minimises computation and communication)

Longitude Protocol Simplified 1



Notes

- The design of Longitude is based on proxy re-encryption. In a proxy re-encryption scheme, a ciphertext encrypted by one key can be transformed by a proxy function into the corresponding ciphertext for another key without revealing any information about the keys and the plaintext.
- 1.Setup and Registration. To share her location with Bob, Alice and Bob must first register with the location service provider (Luke). During registration, Alice and Bob also obtain public cryptographic parameters and generate a public/private key pair locally on their mobile devices.
- 2. Sharing request. After registration, Bob can send a request to Alice asking her to share her location with him.The request can be done out of band without involving Luke. In the request, Bob provides a copy of his public key.
- 3. Sharing Authorisation. If Alice agrees, she computes a re-encryption key using Bob's public key and her own private key. Alice also decides how accurate the location should be for Bob and generates a corresponding precision mask. The re-encryption key and the precision mask are sent to Luke, and act as an authorisation policy that allows Bob to retrieve Alice's location.
- 4. Location Updates. Alice can now send encrypted location data to Luke. Bob's public key can also be discarded by Alice. Luke only stores a user's most recent location. The previous location is overwritten by a newly received location.
- 5. Location retrieve. When Bob wants to know where Alice is, he sends a request to Luke, who retrieves Alice's last encrypted location, applies the re-encryption key and policies defined by Alice then sends it to Bob. Bob can then decrypt the location received from Luke and process it as needed, e.g. to display Alice's location on a map.

Longitude Protocol Simplified 2

0. **Setup**
   - Luke: Luke generates public parameters common to all users
   - Alice, Bob, ...: Alice and Bob generate a public/private keypair: (**Pa**, **Va**), (**Pb**, **Vb**) respectively.

1. **Registration** Alice and Bob register with Luke
   - Alice → Luke: RegisterMe("*Alice*", ...)
   - Bob → Luke: RegisterMe("*Bob*", ...)

2. **Request:** Bob requests Alice to share her location with him
   - Bob → Alice: "*Please share your location with me?*", **Pb**
   - Alice → Bob: "*Okay*"

3. **Authorise:** Alice sends a **re-encryption key** to "authorise Bob at **Precision 5"** to Luke. Alice can also set other policies like the times that sharing is permitted or not.
   - Alice: **RKab** = ***genReEncKey***(**Va**, **Pb**), **PRECab**=5
   - Alice → Luke: Authorise("*Bob*", **RKab**, **PRECab**)

4. **Update:** Alice periodically sends Luke her location encrypted with *her* Public Key **Pa**. Luke keeps last update only.
   - Alice → Luke: ELOCa =***genEncLoc***(LOCa, Pa)

5. **Retrieve:** Bob requests Alice's last location. Luke returns the location encrypted for Bob using Alice's Re-encryption key for Bob at the precision Alice authorised.
   - Bob → Luke: "*Where is Alice?*"
   - Luke → Bob: ELOCab =***reGenEncLoc***(ELOCa, **RKab**, **PRECab**)
   - Bob: LOCa = ***Decrypt*** (ELOCab, **Vb**)

Precision of Encrypted Location

- Latitude and Longitude coordinates are represented as a pair of decimal numbers e.g. **(51.49875, -0.17917)** for the Department of Computing.

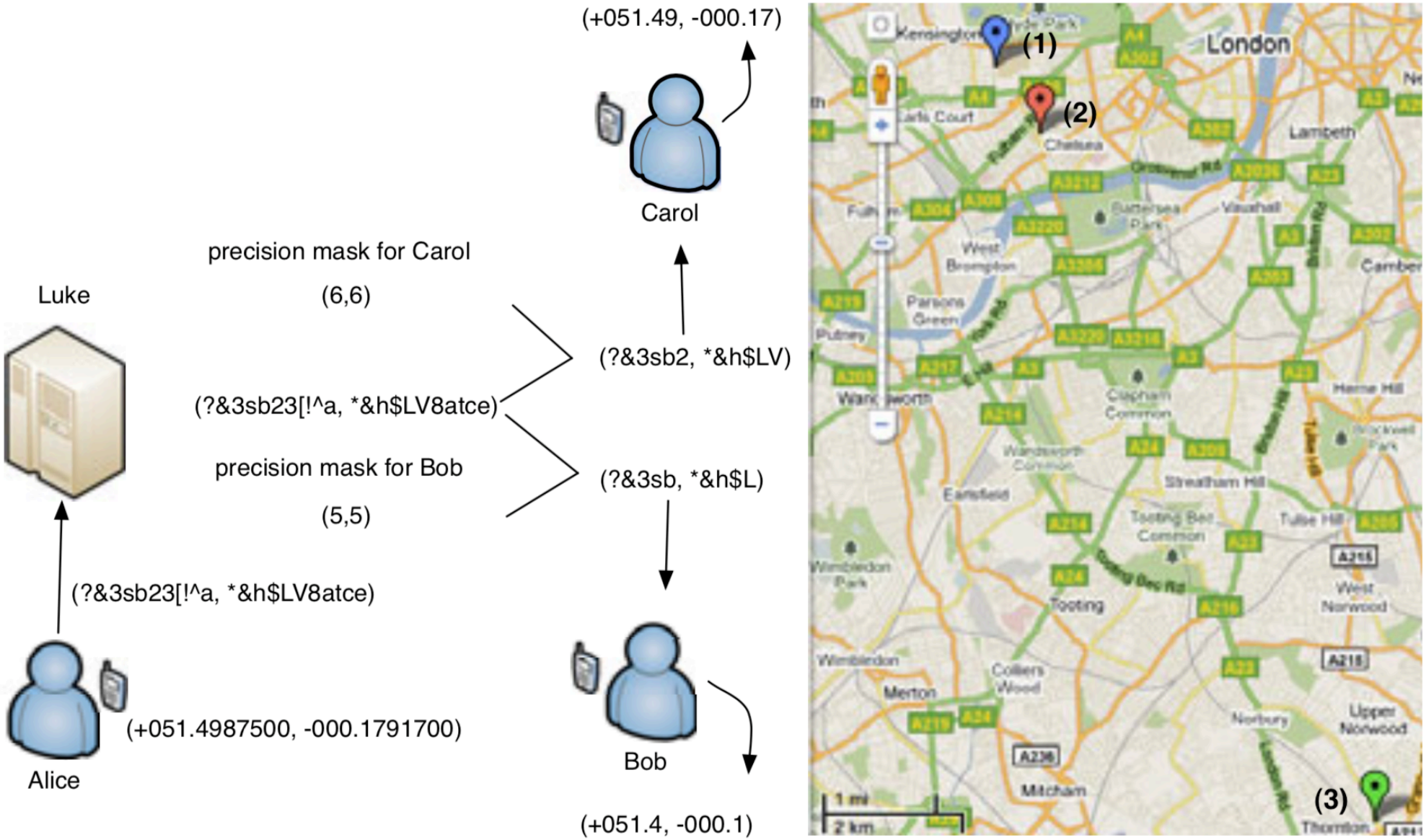
- 5 decimal places is precise to ~1m,
  4 decimal places is precise to ~11m,
  3 decimal places is precise to ~111m,
  2 decimal places is precise to ~1.1km,
  1 decimal places is precise to 11km.

- Alice encodes each coordinate as a 11 character string **SIIIFFFFFFF** (Sign, integer, fraction) e.g. 51.49875 would be encoded as "+**0514987500**"

- This is converted into bits and xor'ed with a random bit stream.

- The encrypted coordinate is truncated by Luke when sending location to Bob respecting Alice's policy - the precision level set by Alice in step 4.
  Luke does not need to decrypt the data to do this, he does it "blindly".

Applying a precision mask to encrypted location



Notes

- In the example:
  Point 1 (+051.4987500, -000.1791700) is Alice's actual location, while point 2 (+051.49, -000.17) and point 3 (+051.4, -000.1) are the displayed locations for two different precision masks (6,6) and (5,5), for two different friends, Carol and Bob.
- The precision mask (X, Y) denotes the first X characters of Longitude's 11 character string encoding for latitude. Similarly Y denotes the first Y characters of Longitude's 11 character encoding for longitude.
- Alice can also specify time-based policies to further control her privacy. An example of such a policy could be "My *co-workers* should not see my location during weekends". The policies are specified by Alice as constraints and uploaded to Luke. The policies do not need to be encrypted because they contain no location data (although they might contain other sensitive information). Luke is responsible for checking and enforcing these policies when Alice's location is requested by her co-workers.

- In the example (bottom left). Alice sends her location (`+051.4987500, -000.1791700`) to Luke as a pair of encrypted 11 character strings (`?&3sb23[!^a, *&h$LV8atce`) i.e. `SIIIFFFFFFF` versions xor'ed with a random bits of the same length (a truncate-able stream cipher). The key is also sent encrypted using the proxy re-encryption.
- If Carol requests Alice location she will be sent the first 6 characters of the encrypted pair i.e. (`?&3sb2, *&h$LV`) plus the encrypted key which she can decrypt and then use to get:
  (`+051.49, -000.17`) approx. 1.1 Km precision.
- Similarly if Bob requests Alice location he will be sent the first 5 characters of the encrypted pair (`?&3sb2, *&h$LV`) plus the encrypted key to get:
  (`+051.4, -000.1`) approx. 11 Km
-

Friend Revocation

- ***How can Alice stop Bob from accessing her location (revocation)?***
  Alice could ask Luke not to send any further updates to Bob, i.e. Luke removes the AliceBob Re-Encryption Key (RKab). Note: RKab is not the same as RKba.

- ***But what if Luke colludes with Bob?***
  To prevent this Alice can generate a new public-private keypair using parts of the old keypair and new re-encryption keys using parts of the old re-encryption keys for her remaining location-sharing friends (see tutorial).

- ***Also will Alice's remaining friends need to be notified and will they need to generate new re-encryption keys for Alice since the re-encryption key for Alice is generated from the friend's private key and Alice's public key?***
  Not in Longitude. In Longitude the friends re-encryption key for Alice is generated using an unchanged part of Alice's public key.

Proxy Re-encryption in Longitude

- Longitude's proxy re-encryption scheme is based on symmetric **bilinear pairings** that pair two elements from one *group* to an element of a second *group*.
- G_1 and G_2, are two cyclic groups of prime order q. g is a generator for G_1

(See Smart p4-p5 for a simple description of cyclic groups and generators).

The bilinear pairing $e: G_1 \times G_1 \rightarrow G_2$ has the following properties:

Bilinearity:

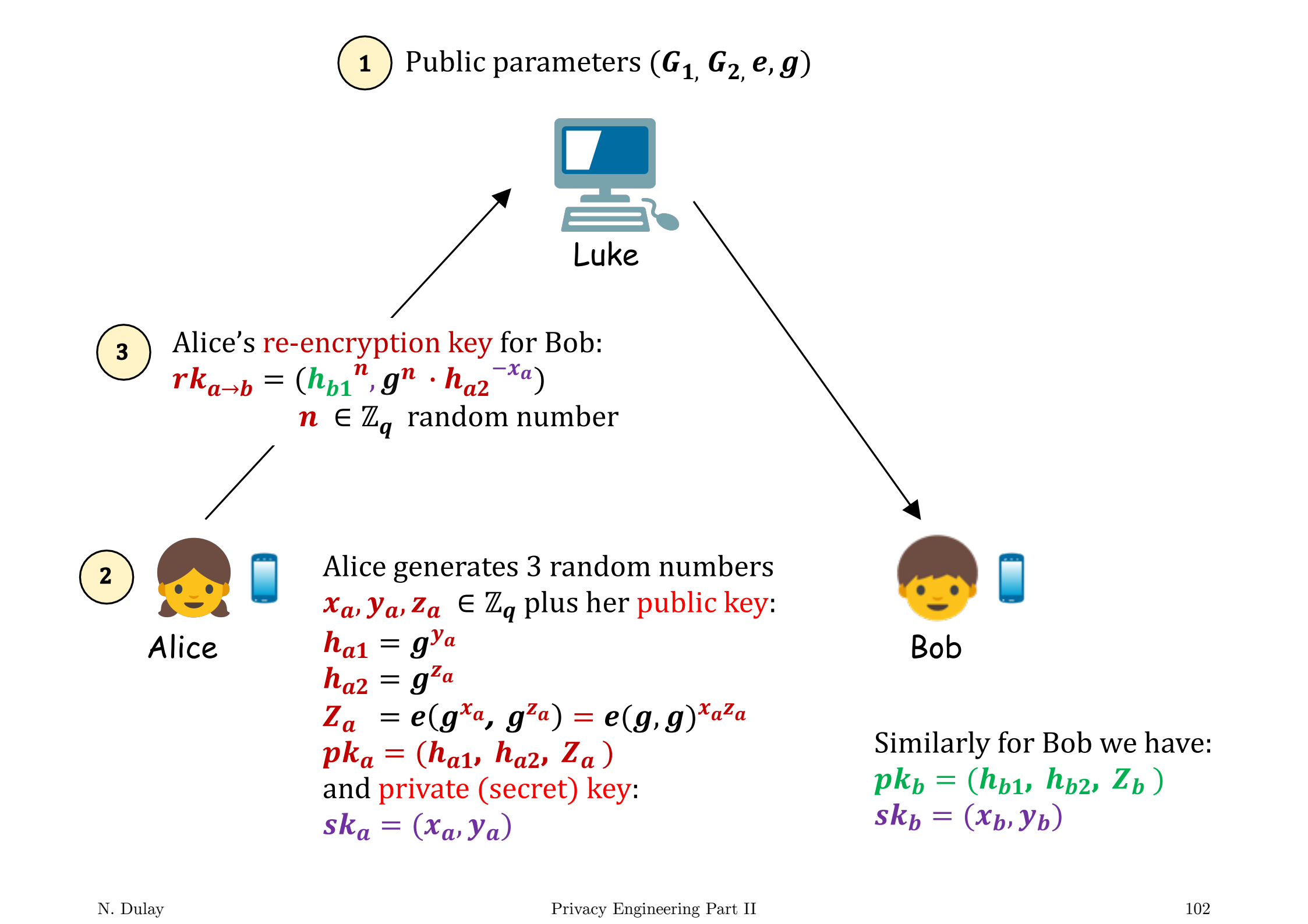
forall $u, v \in G_1$, $a, b \in \mathbb{Z}q$ we have $e(u^a, v^b) = e(u, v)^{ab}$

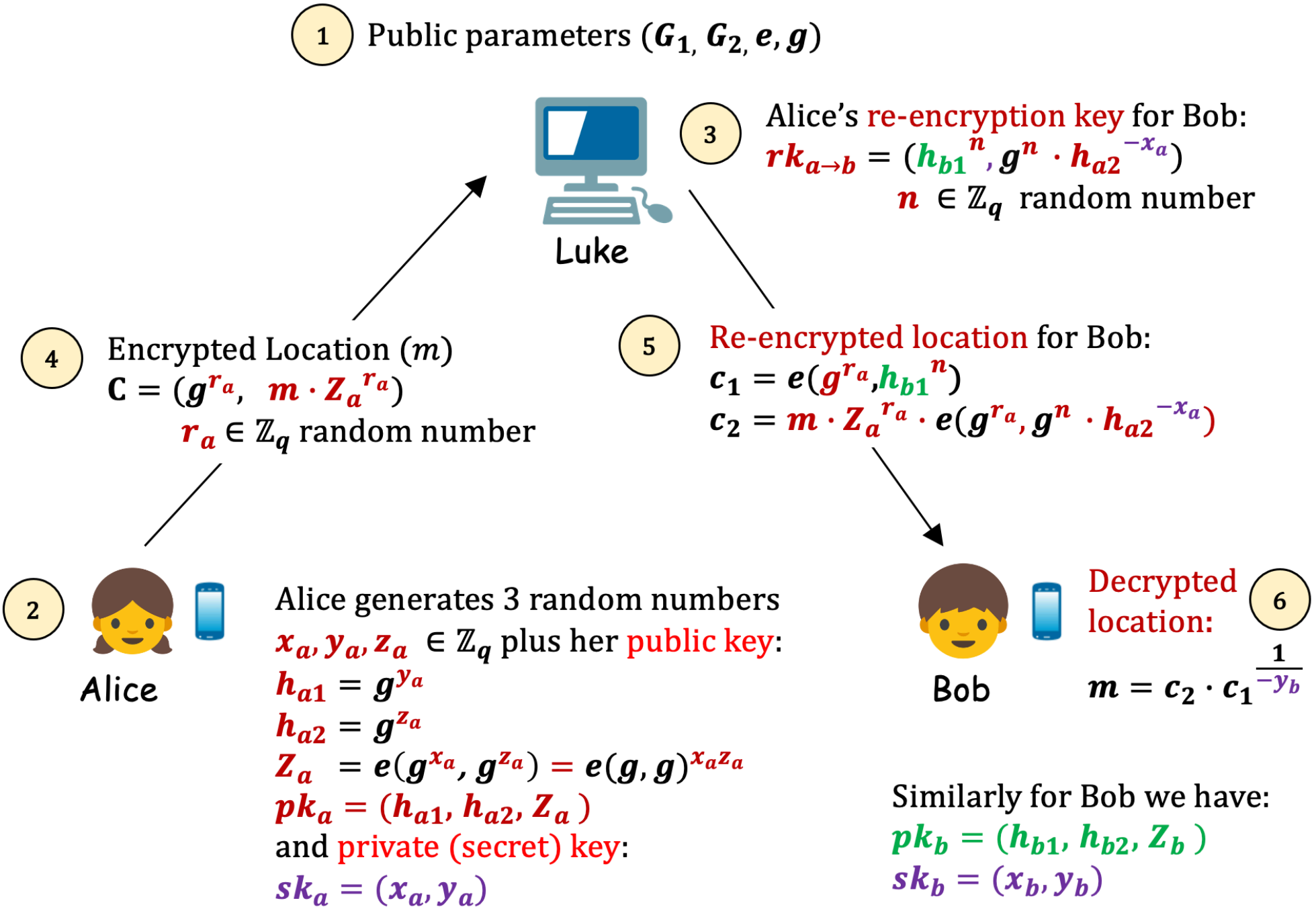
Non-degeneracy:

$e(g, g) \neq 1$

Computable:

There exists an efficient algorithm to compute $e(u, v)$ *forall* $u, v \in G_1$


1
Public parameters (G_1, G_2, e, g)
Luke
3
Alice's re-encryption key for Bob:
$rk_{a\to b} = (h_{b1}{}^{n}, g^{n} \cdot h_{a2}{}^{-x_a})$
$n \in \mathbb{Z}_q$ random number
2
Alice
Alice generates 3 random numbers
$x_a, y_a, z_a \in \mathbb{Z}_q$ plus her public key:
$h_{a1} = g^{y_a}$
$h_{a2} = g^{z_a}$
$Z_a = e(g^{x_a}, g^{z_a}) = e(g,g)^{x_a z_a}$
$pk_a = (h_{a1}, h_{a2}, Z_a)$
and private (secret) key:
$sk_a = (x_a, y_a)$
Bob
Similarly for Bob we have:
$pk_b = (h_{b1}, h_{b2}, Z_b)$
$sk_b = (x_b, y_b)$


1
Public parameters (G_1, G_2, e, g)
Luke
3
Alice's re-encryption key for Bob:
$rk_{a\to b} = (h_{b1}{}^{n}, g^{n} \cdot h_{a2}{}^{-x_a})$
$n \in \mathbb{Z}_q$ random number
4
Encrypted Location (m)
$C = (g^{r_a}, \; m \cdot Z_a{}^{r_a})$
$r_a \in \mathbb{Z}_q$ random number
5
Re-encrypted location for Bob:
$c_1 = e(g^{r_a}, h_{b1}{}^{n})$
$c_2 = m \cdot Z_a{}^{r_a} \cdot e(g^{r_a}, g^{n} \cdot h_{a2}{}^{-x_a})$
2
Alice
Alice generates 3 random numbers
$x_a, y_a, z_a \in \mathbb{Z}_q$ plus her public key:
$h_{a1} = g^{y_a}$
$h_{a2} = g^{z_a}$
$Z_a = e(g^{x_a}, g^{z_a}) = e(g,g)^{x_a z_a}$
$pk_a = (h_{a1}, h_{a2}, Z_a)$
and private (secret) key:
$sk_a = (x_a, y_a)$
Bob
Decrypted location:
6
$m = c_2 \cdot c_1{}^{\frac{1}{-y_b}}$
Similarly for Bob we have:
$pk_b = (h_{b1}, h_{b2}, Z_b)$
$sk_b = (x_b, y_b)$

Notes

- **1. Setup**
  Luke generates public parameters $(e, g, G1, G2)$ where e *is* bilinear pairing from $G1 \times G2 \rightarrow G2$, g is the generator for $G1$.

- **2. Alice's Key generation**
  Alice generates her "public key": $pk_a = (h_{a1}, h_{a2}, Z_a)$ and her "private key" $sk_a = (x_a, y_a)$ using e, g and 3 random numbers x_a, y_a, z_a. (see slide for details). Note: Alice's keys are not an RSA public-private key-pair, rather tuples of cleverly constructed values.

- **3. Re-encryption for friend Bob**
  Alice generates a re-encryption key $rk_{a \rightarrow b}$ for her friend Bob. This is generated using values from Alice $(h_{a2}{}^{-x_a})$ and Bob (h_{b1}).

  Luke will be able to use Alice's encrypted location (step 4) plus the two elements of the Alice's re-encryption key for Bob to re-encrypt the location such that Bob will be able to recover the location.

- **4. Alice updates (encrypted) location**
  Alice encrypts her location m using Z_a, g and a random number r_a and sends the resulting pair of values to Luke.

- **5. Luke Re-encrypts Alice encrypted location for friend Bob**
  When Bob asks for Alice's location, Luke will re-encrypt Alice's encrypted location using (i) both elements of Alice's encrypted location, (ii) both elements of Alice's re-encryption key The resulting ciphertexts c1 and c2 are sent to Bob.

- **6. Decrypt**
  On receipt, Bob decrypts c1 and c2 using an element (y_b) of his private key. Note y_b is related to h_{b1} has follows $h_{b1} = g^{y_b}$

- The tutorial sheet has questions to (i) elaborate step 5 and (ii) show that Bob correctly recovers m *in* step 6.

Private Clouds