How to Update Documents *Verifiably* in Searchable Symmetric Encryption

Kaoru Kurosawa and Yasuhiro Ohtaki

Ibaraki University, Japan

Outline

(1) Introduction

(2) Our (previous) verifiable SSE scheme

(3) Extend it to a dynamic SSE scheme
(but not yet fully verifiable)

(4) Verifiable dynamic SSE scheme

(5) Summary

A Searchable Symmetric Encryption (SSE) scheme

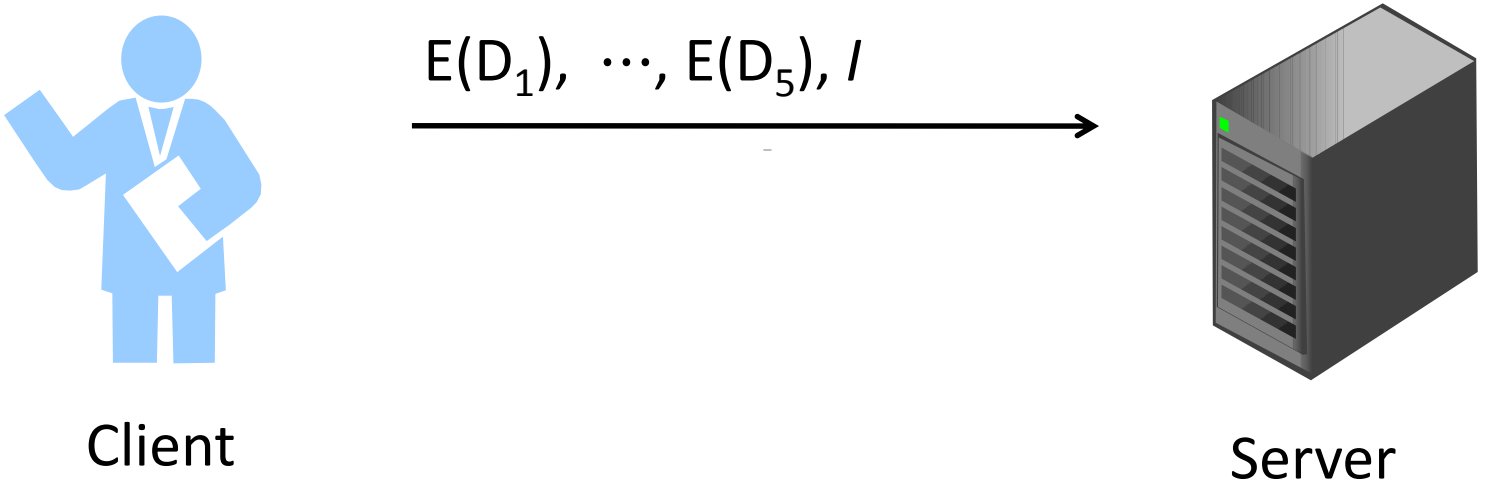
- Consists of a store phase and a search phase

- Suppose that we have
  5 documents and 3 keywords.

keyword	Documents
Austin	D_1, D_3, D_5
Boston	D_2, D_4
Washington	D_1, D_2, D_4

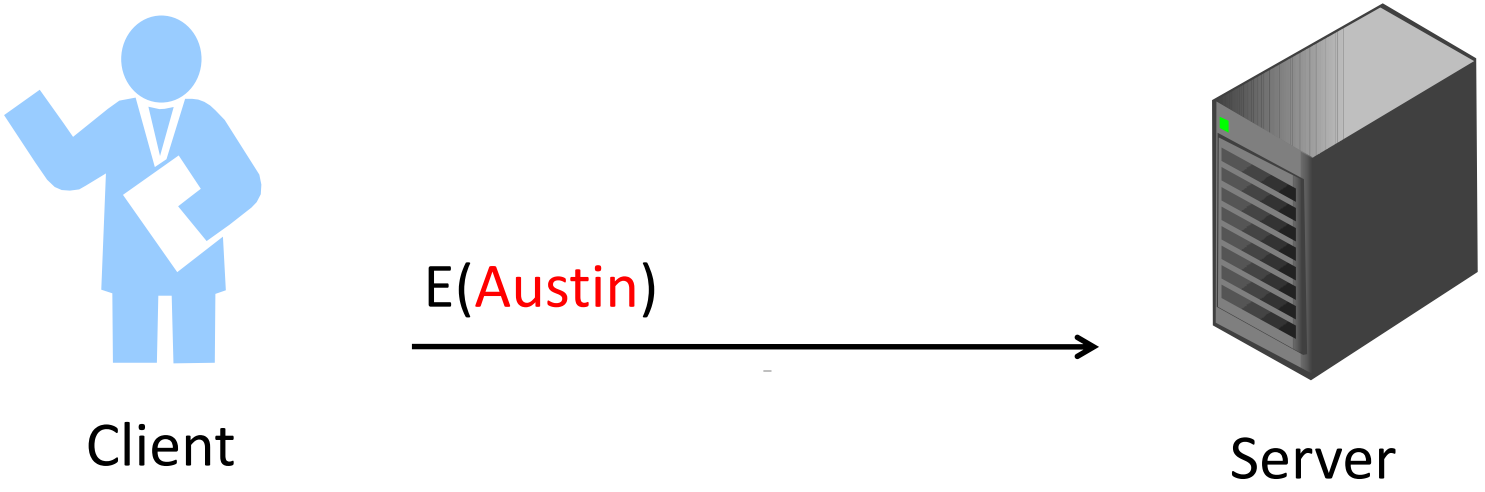
In the store phase,

- A client stores encrypted files (or documents) and an index *I* (in a encrypted form) on a server.



In the search phase,

- the client sends an encrypted keyword to the server.

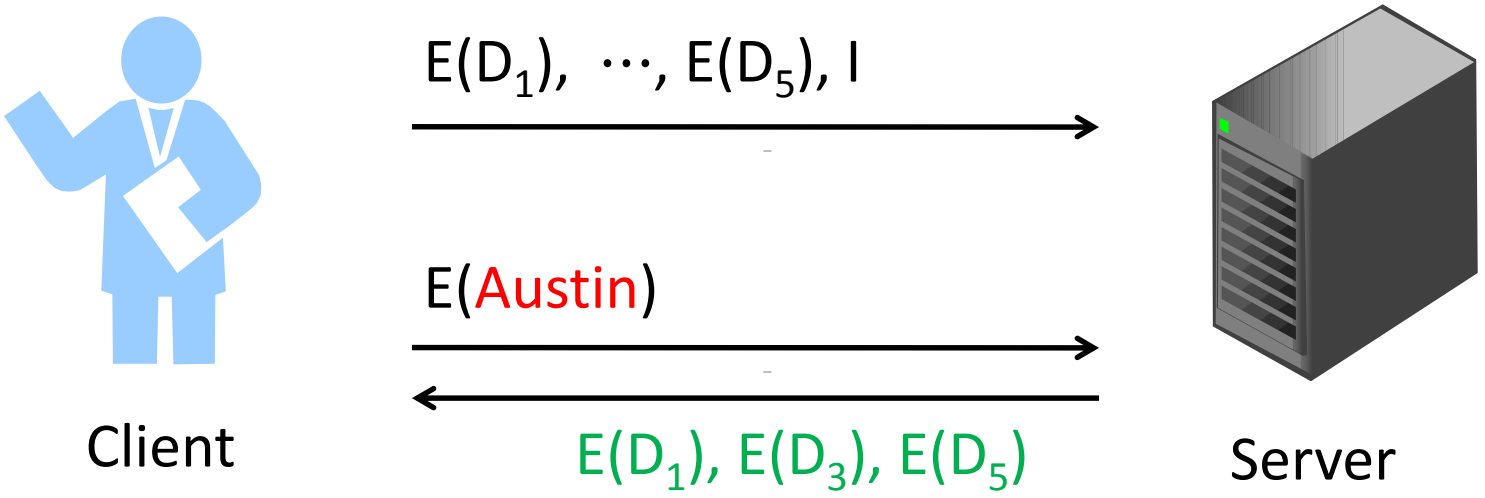


The server somehow returns

- the encrypted files $E(D_1), E(D_3), E(D_5)$ which contain the keyword.

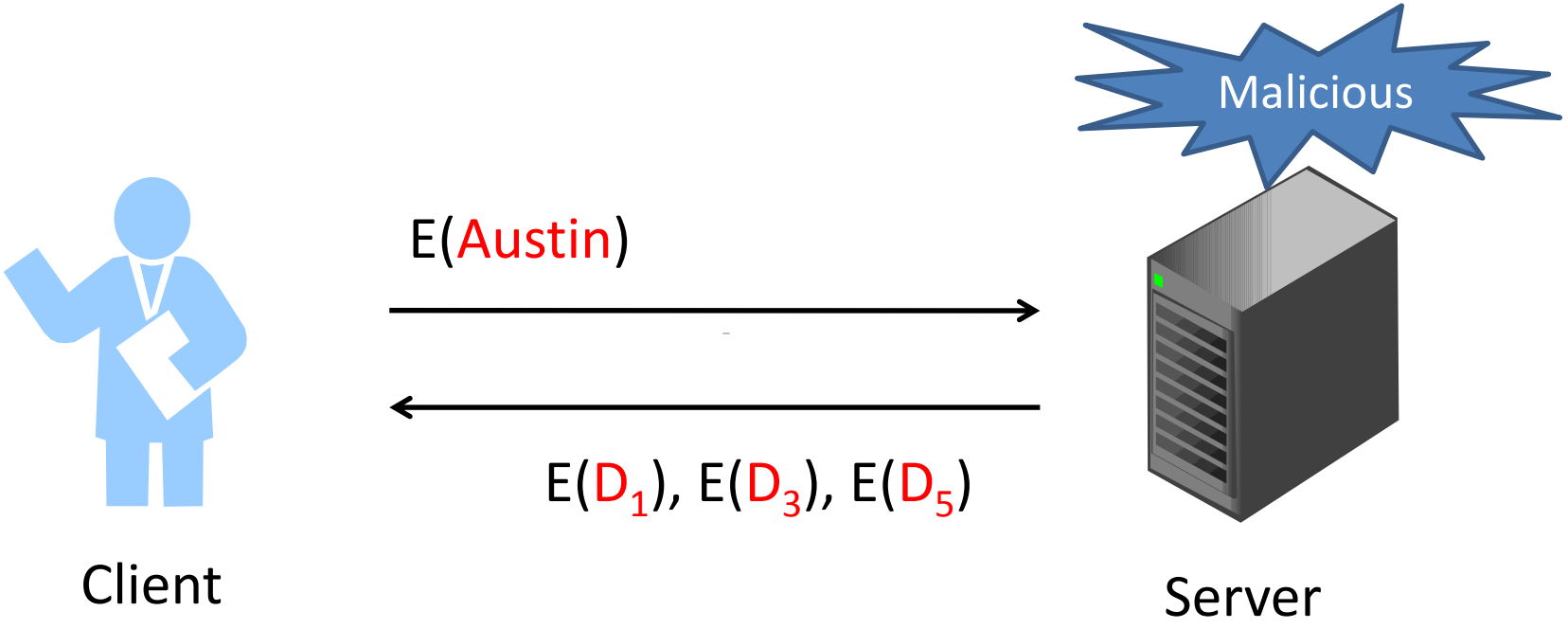
So the client can

- retrieve the encrypted files which contain a specific keyword, keeping the keyword and document secret to the server.



By Passive Attack

- A malicious server breaks the privacy.
- She tries to find
  the keyword and the documents.



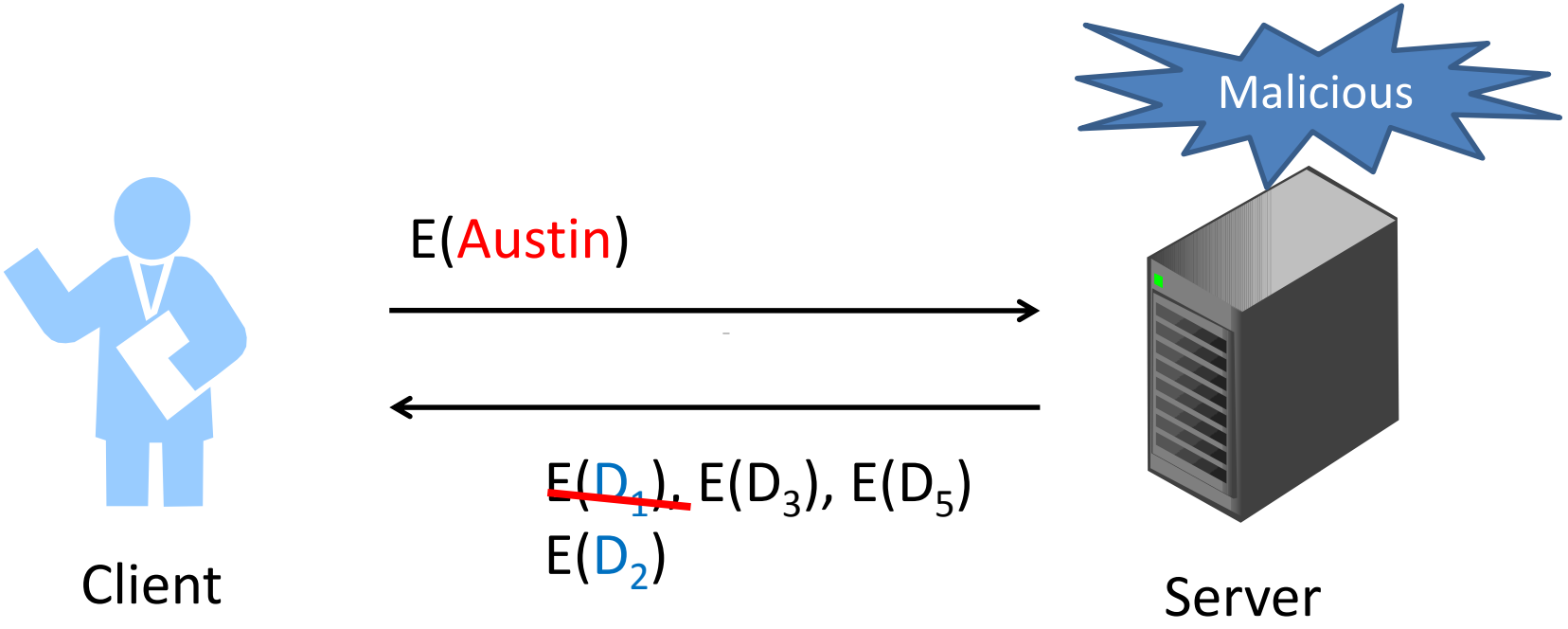
The security against passive attacks

- has been studied by several researchers.

- Curtmola, Garay, Kamara and Ostrovsky showed a rigorous definition of security against passive attacks.
- They also gave a scheme which satisfies their definition.

By Active Attack

- A malicious server breaks the reliability.
- She tries to forge/delete some files or replace E(D_1) with another (valid) E(D_2).



The security against active attacks

- has been studied by Kurosawa et al. (2011)

- They showed a definition of security against active attacks.
- They proposed a verifiable SSE scheme
  - the client can detect any cheating behavior of malicious server.
  - UC-secure

Dynamic SSE scheme

- Kamara et al. showed a dynamic SSE scheme.
  - the client can add and delete documents.
  - secure against adaptive chosen keyword attacks.

- Subsequently, they showed a parallel and dynamic SSE scheme.

- However, these dynamic schemes are not verifiable.

Our contribution

Comparison with Previous Works

	Verifiability	Dynamism
Curtmola et al.	X	X
Kurosawa et al.	O	X
Kamara et al.	X	O
This paper	O	O

Outline

(1) Introduction

(2) Our (previous) verifiable SSE

(3) Extend it to a dynamic SSE scheme
(but not yet fully verifiable)

(4) Verifiable dynamic SSE scheme

(5) Summary

Contents of the encrypted index

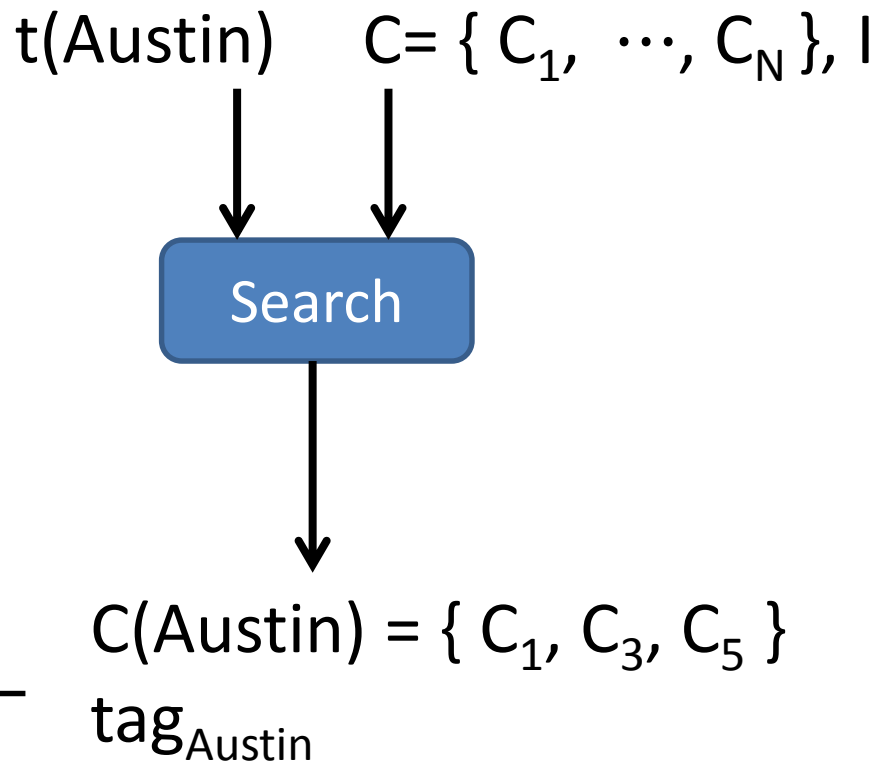
keyword	Documents
Austin	D_1, D_3, D_5
Boston	D_2, D_4
Washington	D_1, D_2, D_4

label	document ID	Authenticator
t(Austin)	*(1, 3, 5)*	tag_{Austin} =MAC(t(Austin), (C_1, C_3, C_5))
t(Boston)	*(2, 4)*	tag_{Boston} =MAC(t(Boston), (C_2, C_4))
t(Washington)	*(1, 2, 4)*	$tag_{Washington}$ =MAC(t(Washington), (C_1, C_2, C_4))

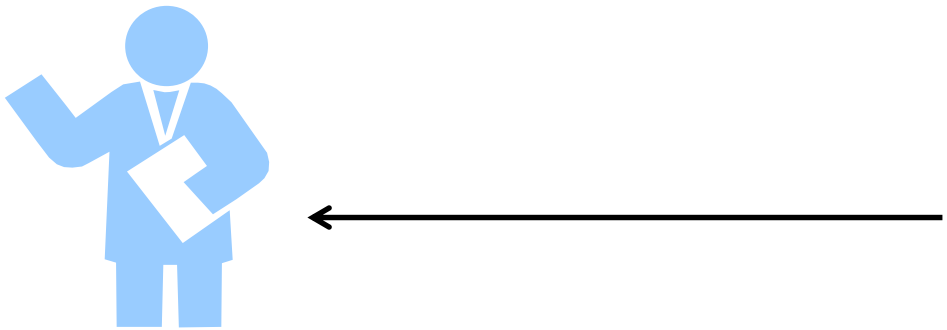
t() : some trapdoor function
MAC() : some Message Authentication Code

In the search phase,

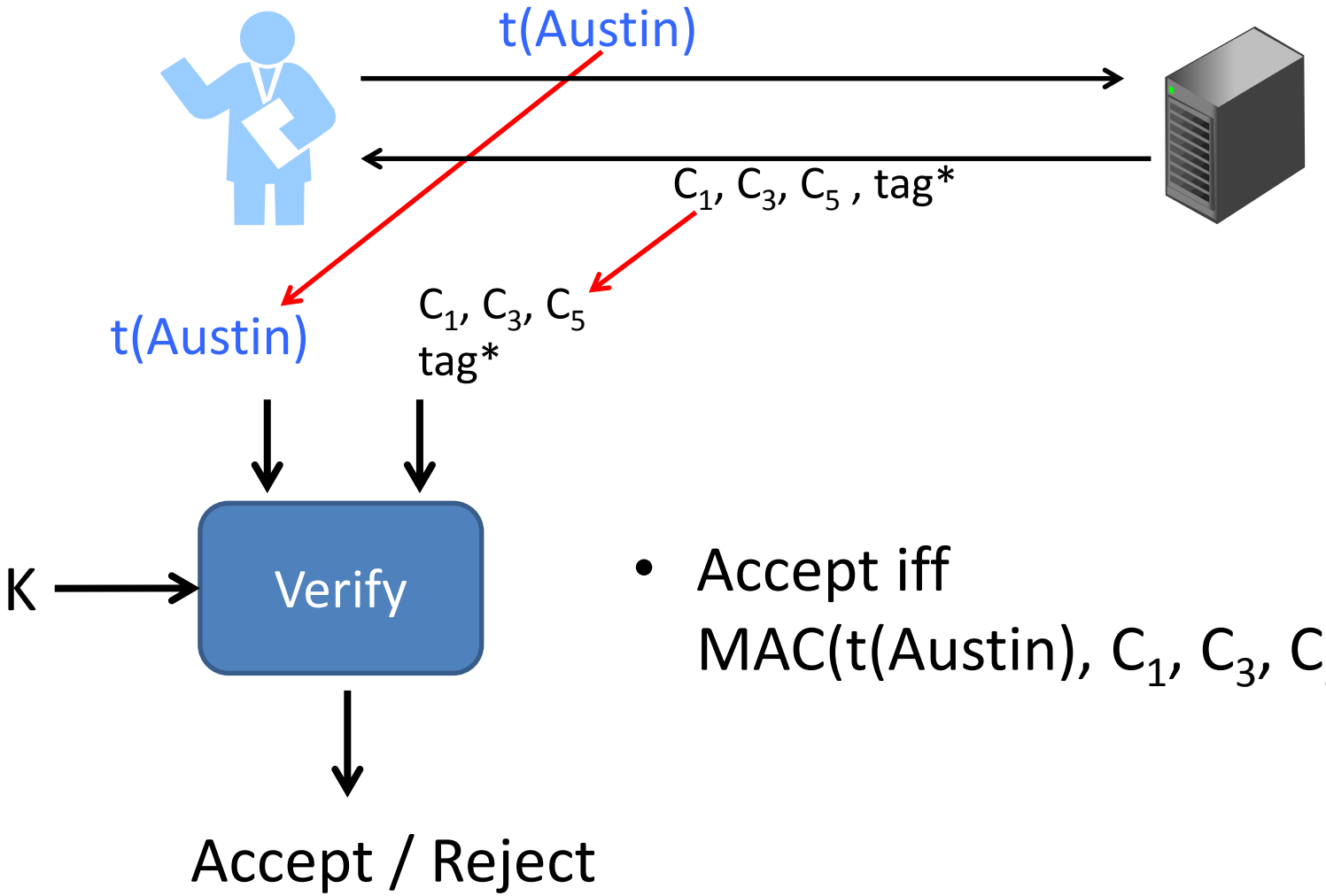
- The server receives
  t(keyword)
  and output
  C(keyword)
  $tag_{keyword}$



and send them to the client.



Verify algorithm



- Accept iff
  $MAC(t(Austin), C_1, C_3, C_5) = tag^*$

Outline

(1) Introduction

(2) Our (previous) verifiable SSE

(3) Extend it to a dynamic SSE scheme
(but not yet fully verifiable)

(4) Verifiable dynamic SSE scheme

(5) Summary

Extending the previous scheme

- When we consider extending
  our previous scheme to a dynamic scheme,
  we noticed that the scheme cannot
  modify documents efficiently.

Suppose the client wants to modify C_1 to C'_1

- The client must store C'_1 and two updated authenticators to the server.

label	document ID	Authenticator
t(Austin)	(1, 3, 5)	tag_{Austin} =MAC(t(Austin), (C_1, C_3, C_5))
t(Boston)	(2, 4)	tag_{Boston} =MAC(t(Boston), (C_2, C_4))
t(Washington)	(1, 2, 4)	$tag_{Washington}$ =MAC(t(Washington), (C_1, C_2, C_4))

- If C_1 includes more keywords, then the client must update more authenticators.

To update efficiently

- Authenticate
  (t(Austin), (1, 3, 5))
  and
  each (i, C_i)
  separately.

	Authenticator for Document
1	MAC(1, C_1)
2	MAC(2, C_2)
3	MAC(3, C_3)
4	MAC(4, C_4)
…	…

label	doc ID	Authenticator for index
t(Austin)	(1, 3, 5)	tag_{Austin} =MAC(t(Austin), (1, 3, 5))
t(Boston)	(2, 4)	tag_{Boston} =MAC(t(Boston), (2, 4))
t(Washington)	(1, 2, 4)	$tag_{Washington}$ =MAC(t(Washington), (1, 2, 4))

To update C_1 to C'_1

- the client has only to store just one authenticator on $(1, C'_1)$ no matter how many keywords are included in C_1 .

	Authenticator for Document
1	MAC(1, C'_1)
2	MAC(2, C_2)
3	MAC(3, C_3)
4	MAC(4, C_4)
…	…

label	doc ID	Authenticator for index
t(Austin)	(1, 3, 5)	tag_{Austin} =MAC(t(Austin), (1, 3, 5))
t(Boston)	(2, 4)	tag_{Boston} =MAC(t(Boston), (2, 4))
t(Washington)	(1, 2, 4)	$tag_{Washington}$ =MAC(t(Washington), (1, 2, 4))

Deletion and Addition

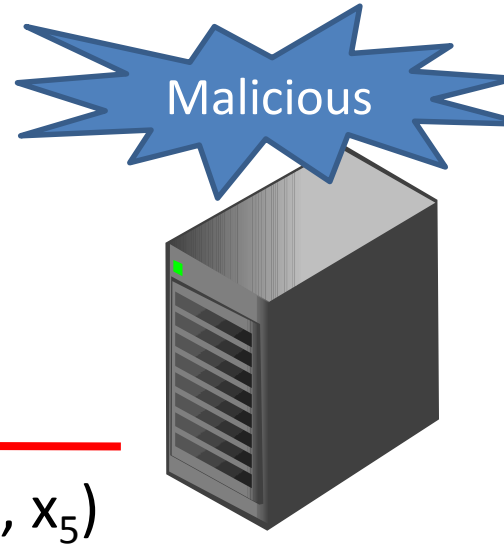
- To *delete* a document C_2
  the client runs *update* with a special symbol
  C'_2 = *delete* .

- To *add* a new document D_6
  which includes Austin,
  the client updates the authenticator
  tag_{Austin} = MAC(t(Austin), (1, 3, 5))
  to
  tag'_{Austin} = MAC(t(Austin), (1, 3, 5, 6)).

However,

- This scheme is vulnerable against replay attack.
- A malicious server may return
  old
  $(1, C_1, MAC(1,C_1))$
  instead of the latest
  $(1, C'_1, MAC(1,C'_1))$.

Client



Server

tag_{Austin}, ~~$(1, C'_1, x'_1)$~~, $(3, C_3, x_3)$, $(5, C_5, x_5)$

$(1, C_1, x_1)$

So the problem is

- How to timestamp on the *current/updated* (i, C_i)
- How to timestamp on the *current/updated* ($label_{Austin}$, (1, 3, 5))

We solved this problem by

- using an authentication scheme
  which posses the timestamp functionality.
  - Merkle hash tree
  - Authenticated skip list
  - RSA accumulator

These schemes

- Allows one to hash a set of inputs
  $E = \{ x_1, x_2, \ldots, x_n \}$
  into one short accumulation value
  $Acc(E)$.

- Infeasible to find a set E'
  such that $Acc(E) = Acc(E')$, $E \neq E'$.

- Able to construct a "witness" $\square_j$
  that a given input x_j was incorporated into Acc(E).

RSA accumulator

- Let f be a function randomly chosen from a two-universal function family
  $F = \{f_a : \{0,1\}^{3\lambda} \longrightarrow \{0,1\}^{\lambda}\}$.

- For any $y \in \{0,1\}^{\lambda}$,
  we can compute a prime $x \in \{0,1\}^{3\lambda}$
  such that $f(x)=y$ by sampling $O(\lambda^2)$ times
  with overwhelming probability.

RSA accumulator

- Let $p=2p'+1$ and $q=2q'+1$ be two large primes such that p' and q' are also primes and $|pq|>3\lambda$.
- Let $N=pq$.
- Let $QR_N=\{a \mid x^2 \bmod N \text{ for some } x \in Z^*_N\}$. Then QR_N is a cyclic group of size $(p-1)(q-1)/4$.

- Let g be a generator of QR_N.

RSA accumulator

- For a set $E=\{y_1, y_2, ..., y_n\}$ with $y_i \in \{0,1\}^{\lambda}$,
  RSA accumulator works as follows:
  - For each y_i, Alice chooses a prime $x_i \in \{0,1\}^{3\lambda}$ randomly.
    Let *prime*(y_i) denote such a prime x_i.
    x_i = *prime*(y_i)
  - Alice computes accumulated value of set E as
    $Acc(E) = g^{x_1 x_2 \cdots x_n} \bmod N$
    and sends Acc(E) to Bob.

RSA accumulator

- Later, Alice proves that $y_j \in E$ to Bob as follows:
  - She computes

    $\pi_j = g^{x_1 x_2 \dots x_{j-1} x_{j+1} \dots x_n} \mod N$

    $x_j = prime(y_i)$

    and sends π_j and x_j to Bob
  - Bob verifies that

    $Acc(E) = \pi_j^{x_j} \mod N$

To time stamp in our scheme

- We apply the RSA accumulator to the sets

$E_c = \{ (i, C_i) \mid i=1, \ldots, n\}$

$E_I = \{ (\text{label}_i, j, [\overline{\text{index}_i}]_j) \mid i=1,\ldots,m,\ j=1,\ldots,n\}$

and compute their accumulated values

$A_C = \text{Acc}(E_C)$

$A_I = \text{Acc}(E_I)$

$[x]_j$: j-th bit of x

Update and Verification

- The client updates A_C each time when he *modifies* or *deletes* a document.
- He also updates A_I each time when he *adds* a document.

- In the search phase, the client checks whether the server returned the valid latest ciphertexts, based on A_C and A_I.

Outline

(1) Introduction

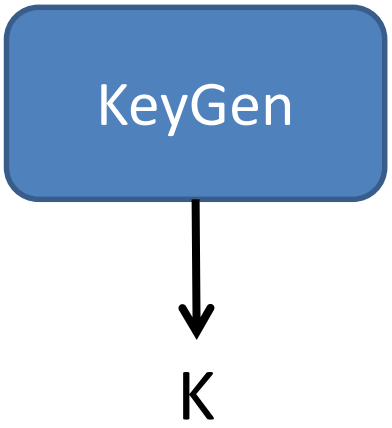
(2) Our (previous) verifiable SSE

(3) Extend it to a dynamic SSE scheme
(but not yet fully verifiable)

(4) Verifiable dynamic SSE scheme

(5) Summary

Key generation



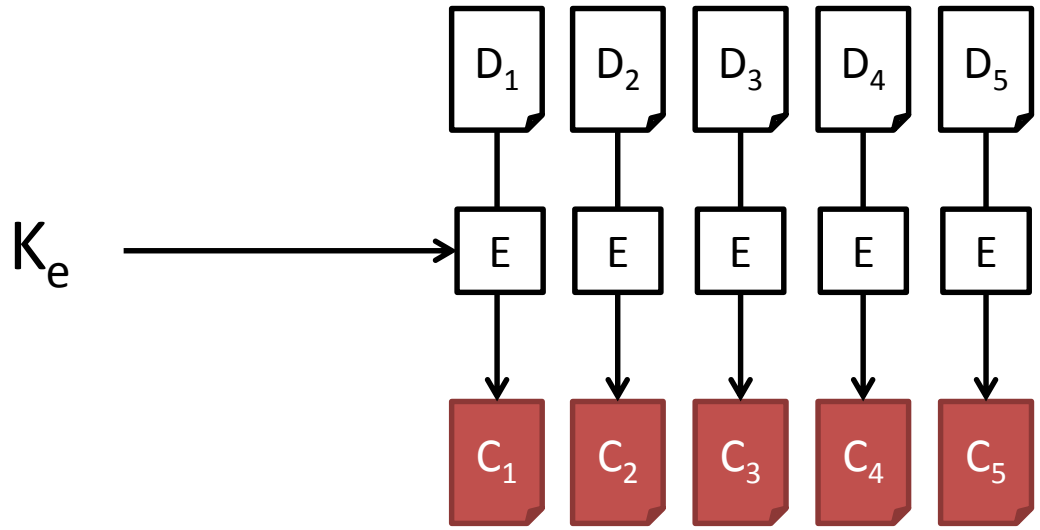
The client first generates set of keys

K_e	key of SKE
K_0, K_1	keys of PRF

and keeps them secret.

Store phase: File encryption

The client first encrypts each document.

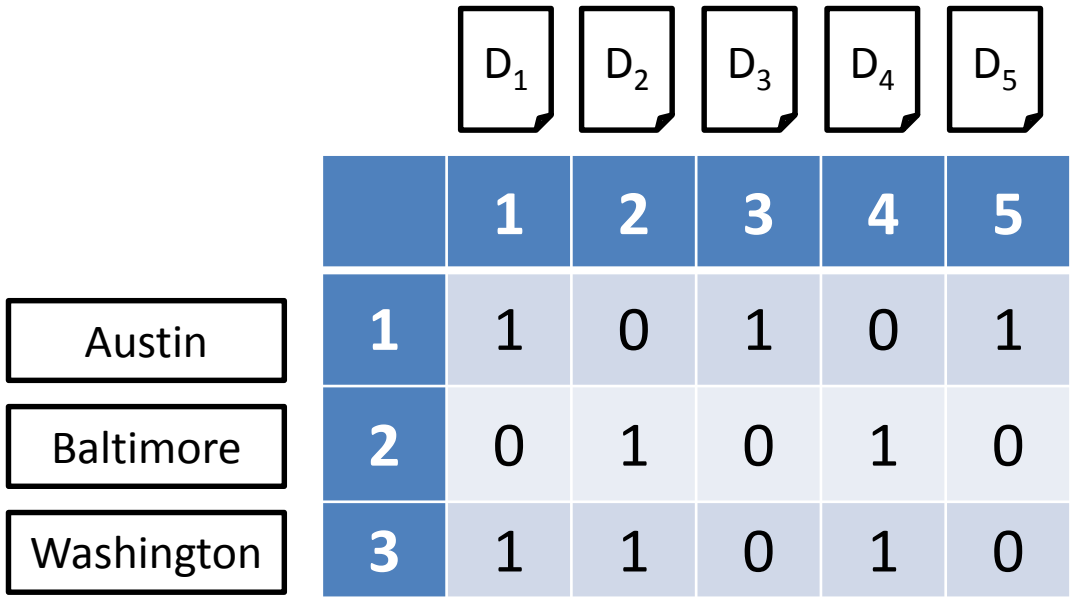


Store phase: Index encryption

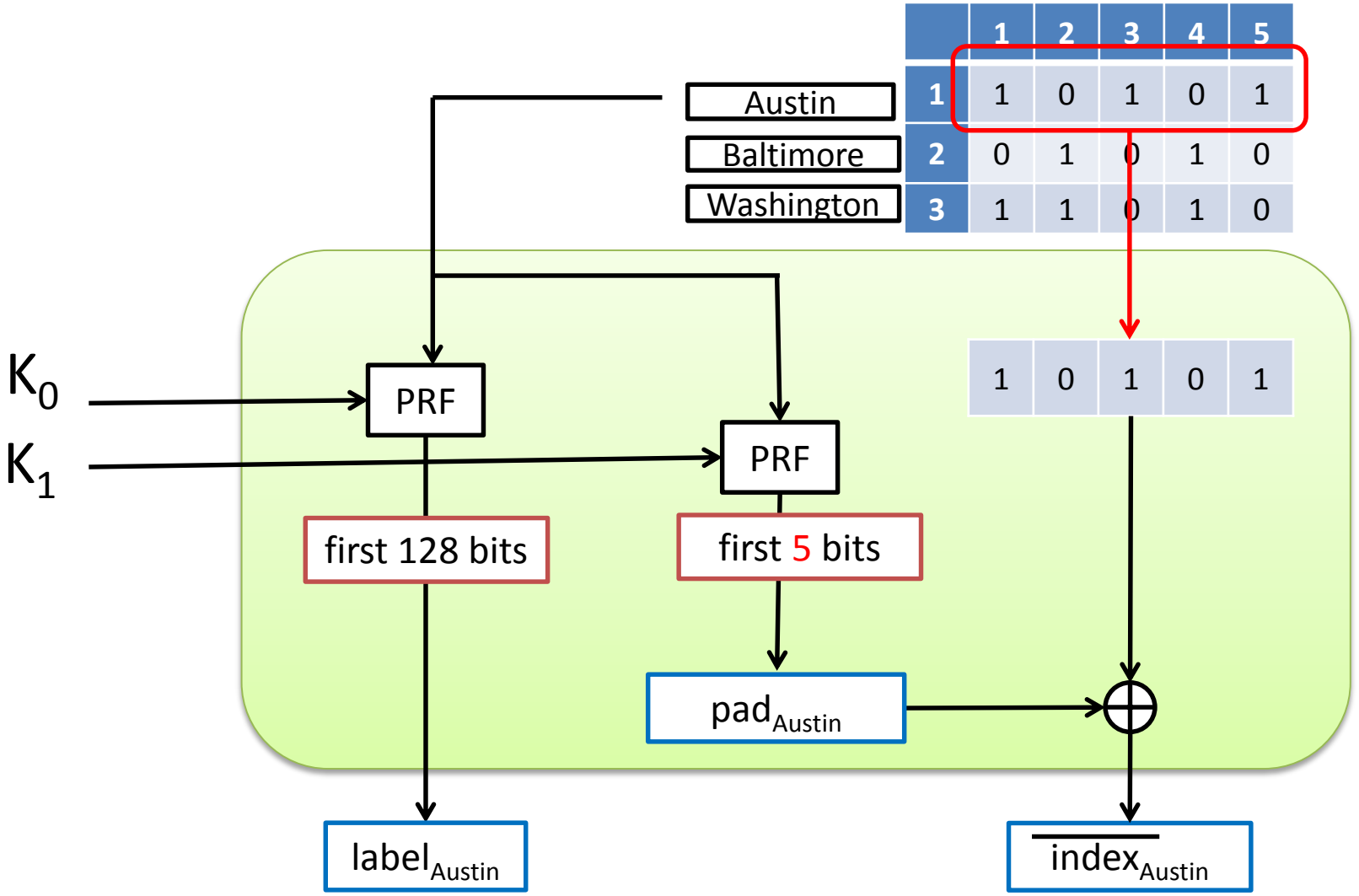
The plain index is an 5 x 3 binary matrix where

$e_{i,j}$ =1 if keyword$_i$ is contained in D_j,

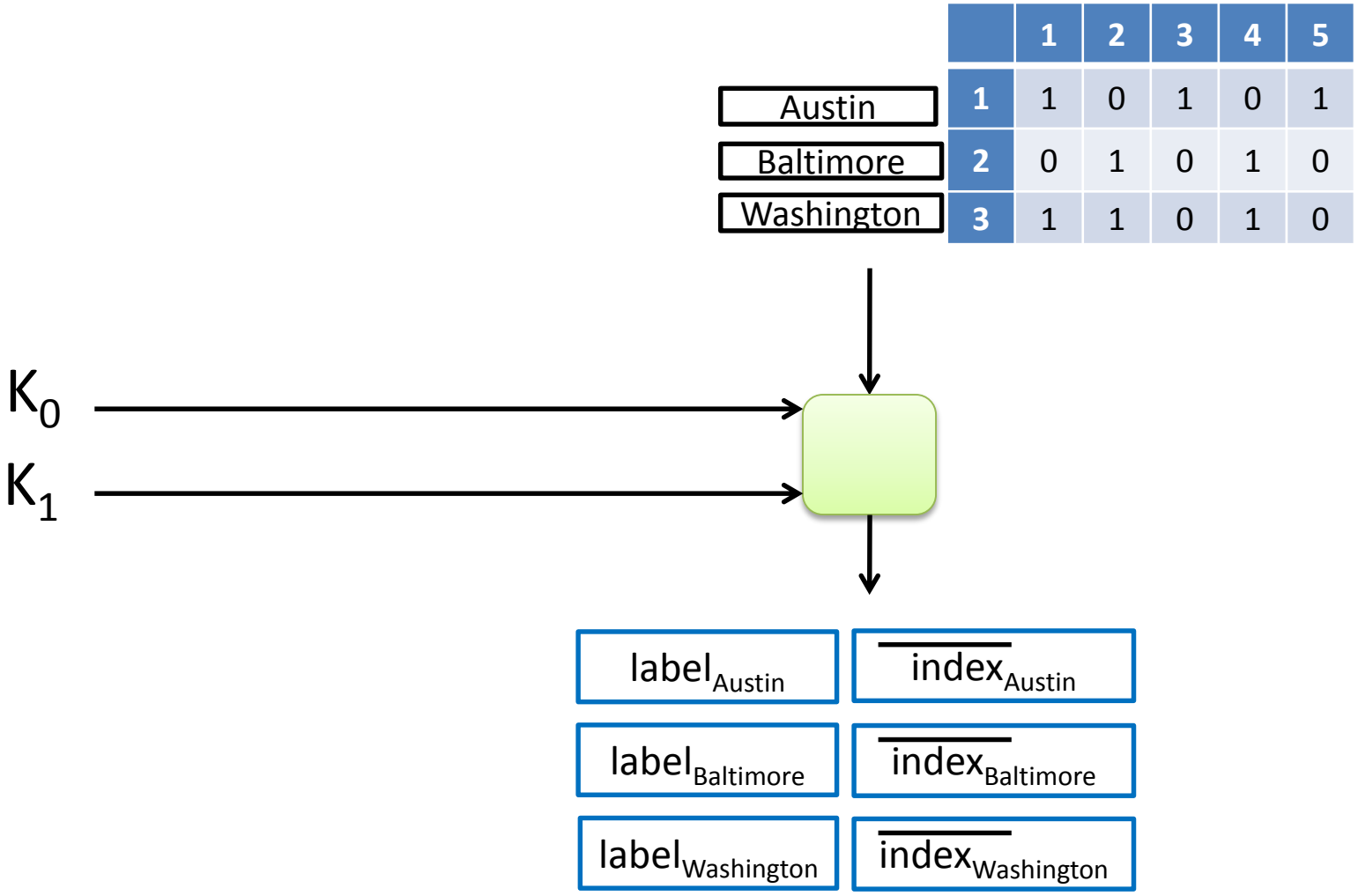
0 otherwise



Store phase: Index encryption



Store phase: Index encryption



Store phase: Compute A_C

The client map each document status into a prime number and computes A_C

$x_1 = prime(H(1, H(C_1)))$

$x_2 = prime(H(2, H(C_2)))$

$x_3 = prime(H(3, H(C_3)))$

$x_4 = prime(H(4, H(C_4)))$

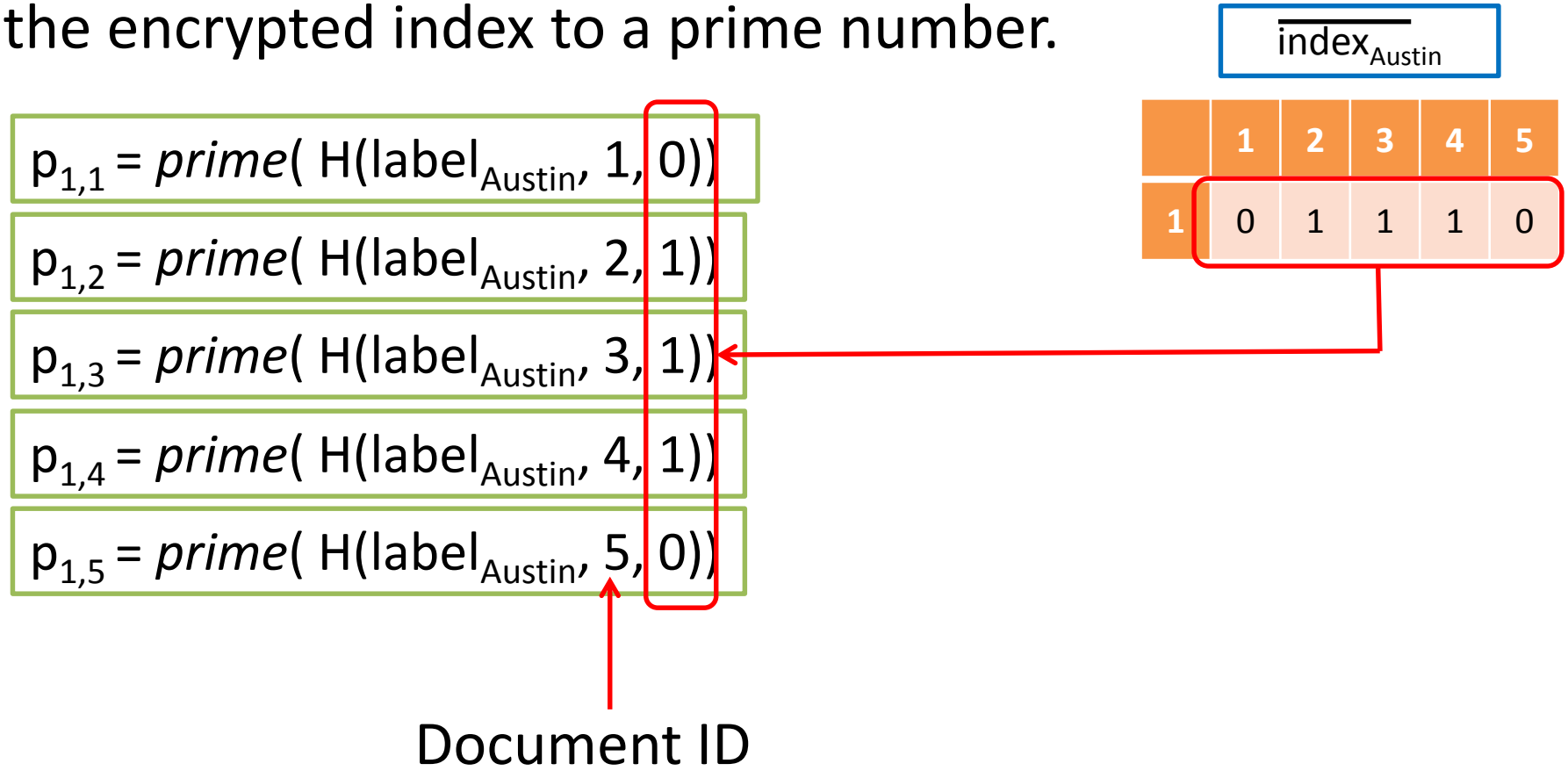
$x_5 = prime(H(5, H(C_5)))$

H() : collision-resistant hash function

$$A_C = g^{x_1 x_2 x_3 x_4 x_5} \mod N$$

Store phase: Compute A_I

The client also maps each element of the encrypted index to a prime number.



Store phase: Compute A_I

Do same for all encrypted index element.

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	1
3	1	0	1	0	0

$p_{1,1} = prime(\ H(label_{Austin}, 1, 1))$

$p_{1,2} = pr$

$p_{1,3} = pr$

$p_{1,4} = pr$

$p_{1,5} = pr$

$p_{2,1} = prime(\ H(label_{Baltimore}, 1, 0))$

$p_{2,2} = prime(\ H(label_{\ldots}$ 2, 0))

$p_{2,3} = prim$

$p_{2,4} = prim$

$p_{2,5} = prim$

$p_{3,1} = prime(\ H(label_{Washington}, 1, 1))$

$p_{3,2} = prime(\ H(label_{Washington}, 2, 0))$

$p_{3,3} = prime(\ H(label_{Washington}, 3, 1))$

$p_{3,4} = prime(\ H(label_{Washington}, 4, 0))$

$p_{3,5} = prime(\ H(label_{Washington}, 5, 0))$

Store phase: Compute A_I

The client computes A_I

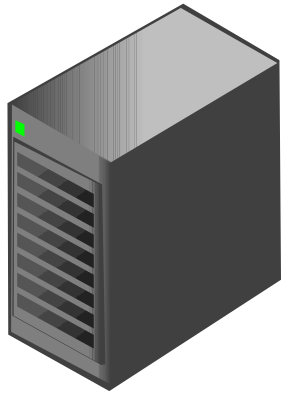
$$A_I = g^{p_{1,1}p_{1,2}\cdots p_{1,5}\ p_{2,1}\cdots p_{2,5}\ p_{3,1}\cdots p_{3,5}} \bmod N$$

The client keeps $n(=5)$, A_I and A_c .

Store phase: Store data

$C = \begin{matrix} (1, C_1) \\ (2, C_2) \\ (3, C_3) \\ (4, C_4) \\ (5, C_5) \end{matrix}$

$I = \begin{matrix} (\text{label}_{\text{Austin}}, \overline{\text{index}}_{\text{Austin}}) \\ (\text{label}_{\text{Baltimore}}, \overline{\text{index}}_{\text{Baltimore}}) \\ (\text{label}_{\text{Wasington}}, \overline{\text{index}}_{\text{Washington}}) \end{matrix}$

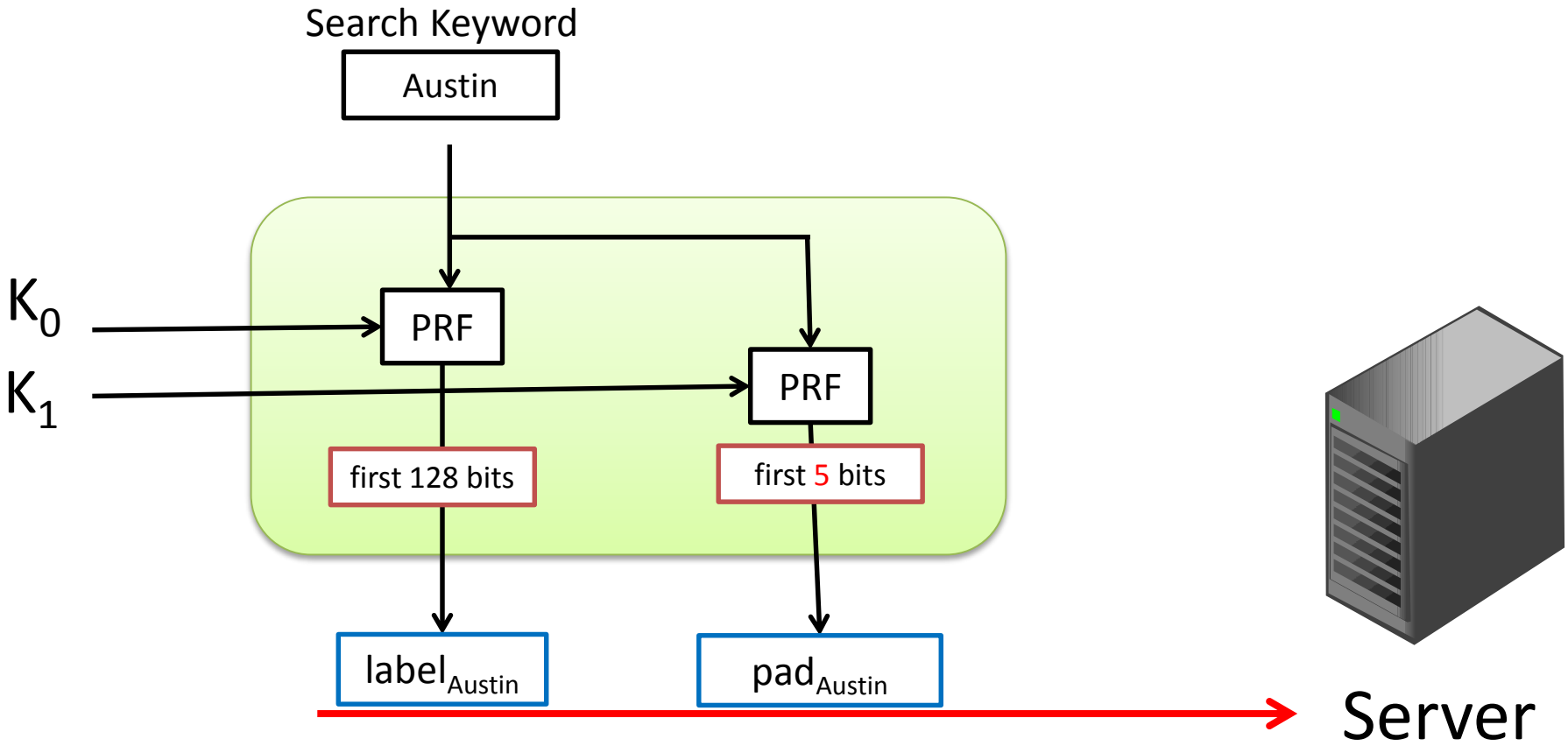


Client

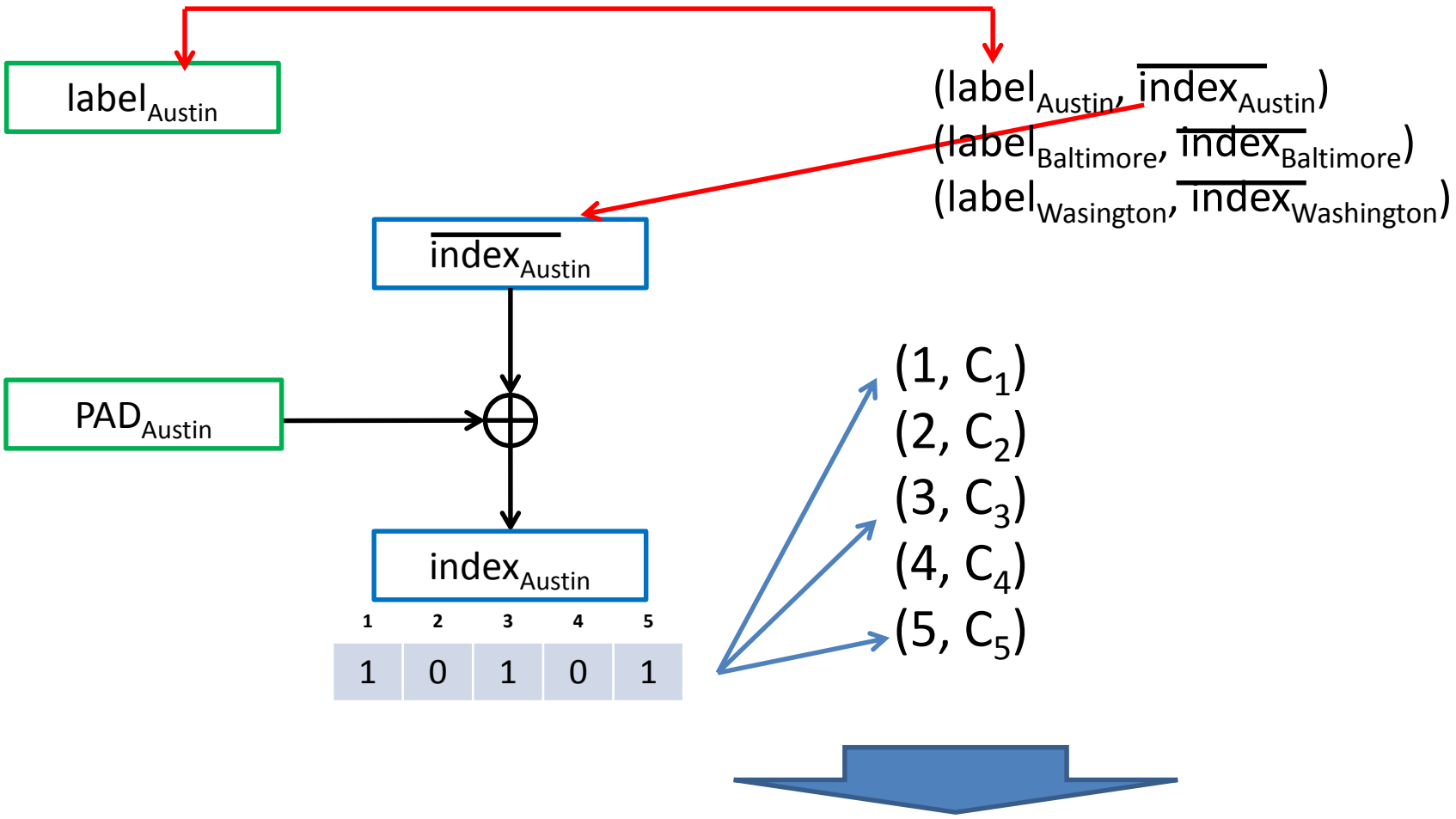
Server

Search phase: Keyword encryption

The client computes the label and pad for the keyword.

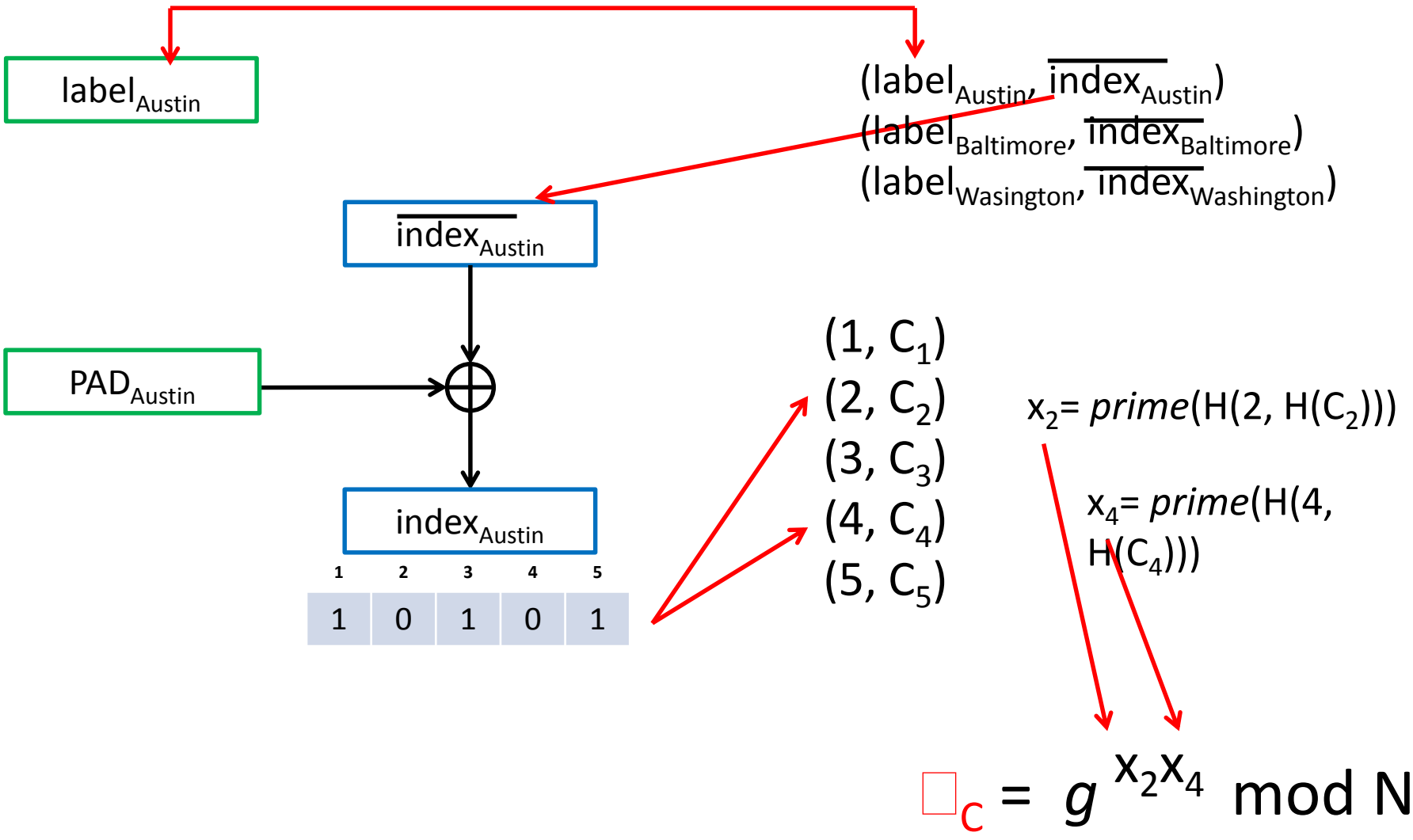


Search phase: Find C(austin)



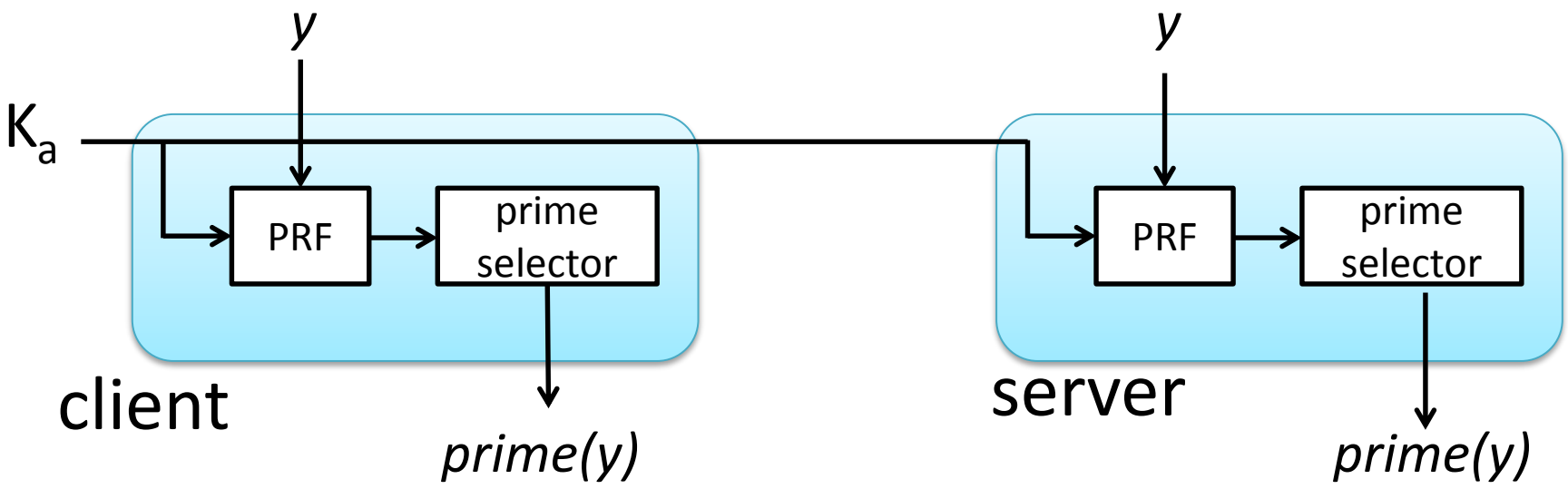
$C(Austin) = (1, C_1), (3, C_3), (5, C_5)$

Search phase: Compute $\square_C$

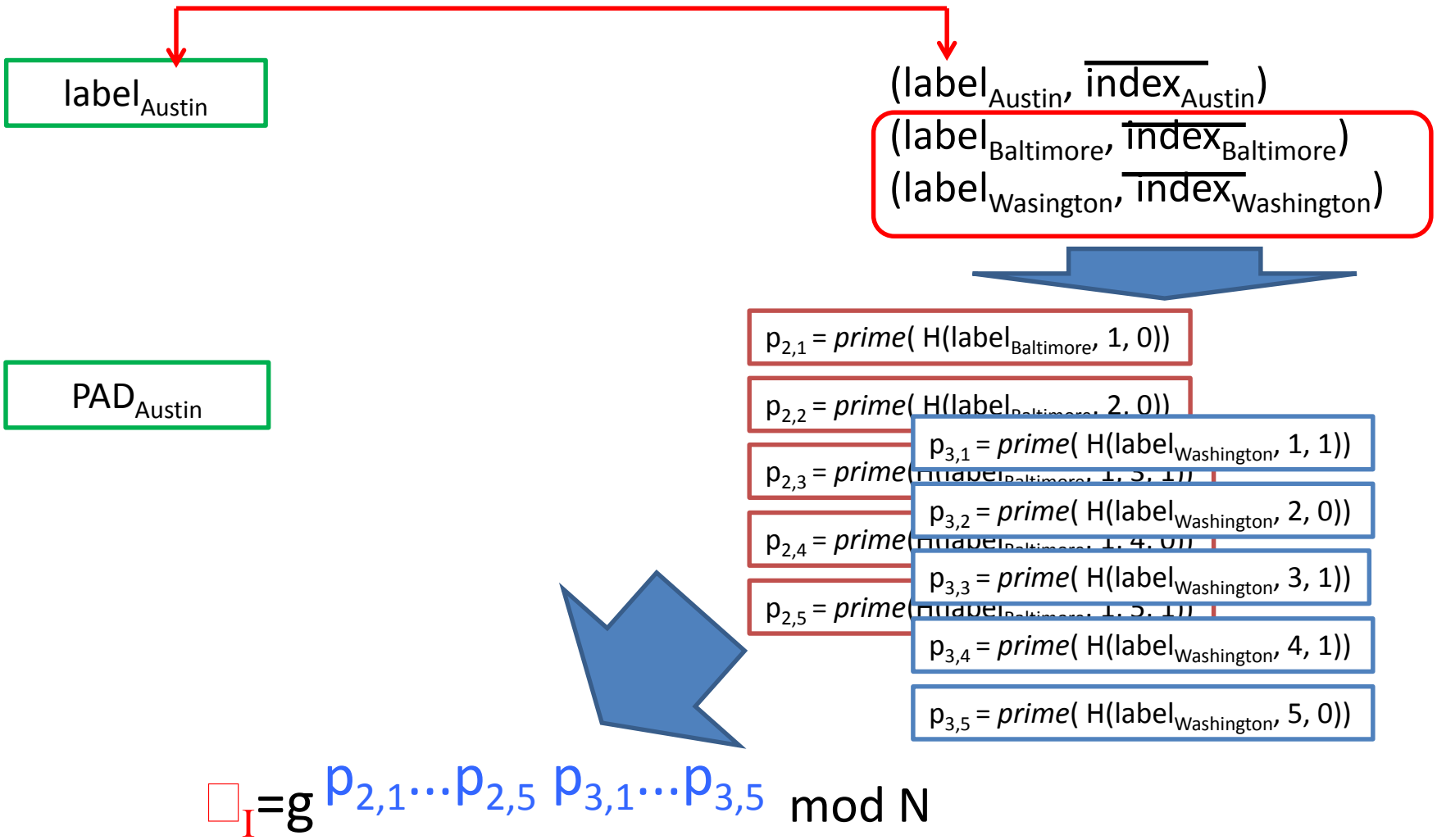


How can the server calculate *prime*() locally?

- *prime(y)* should be a prime number chosen *randomly* by the client.
- In our scheme, we use PRF with a key K_a chosen randomly by the client .



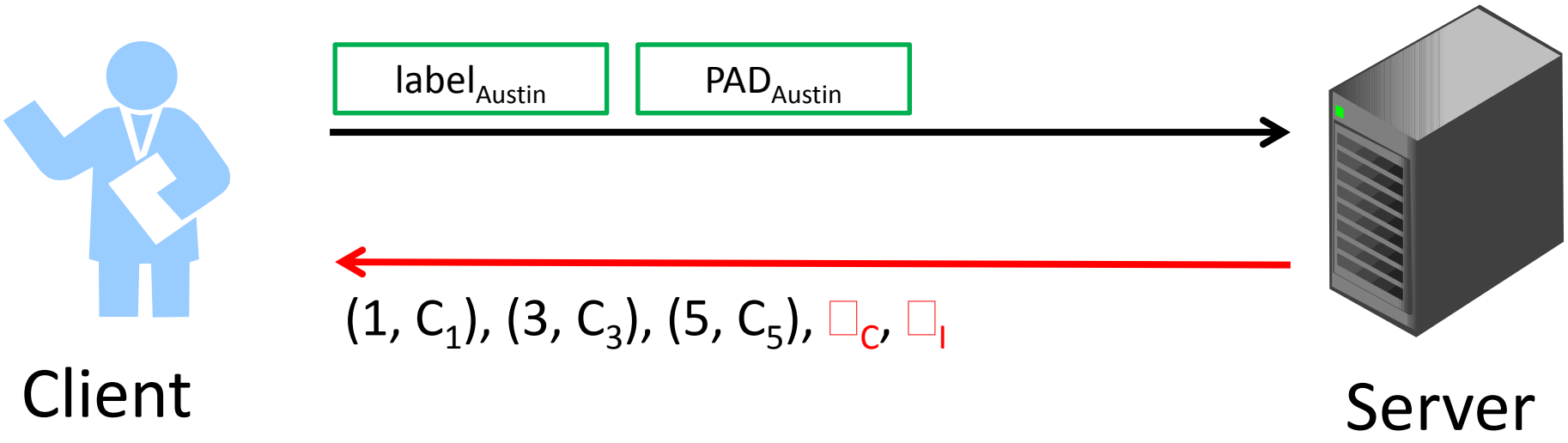
Search phase : Compute Δ_I



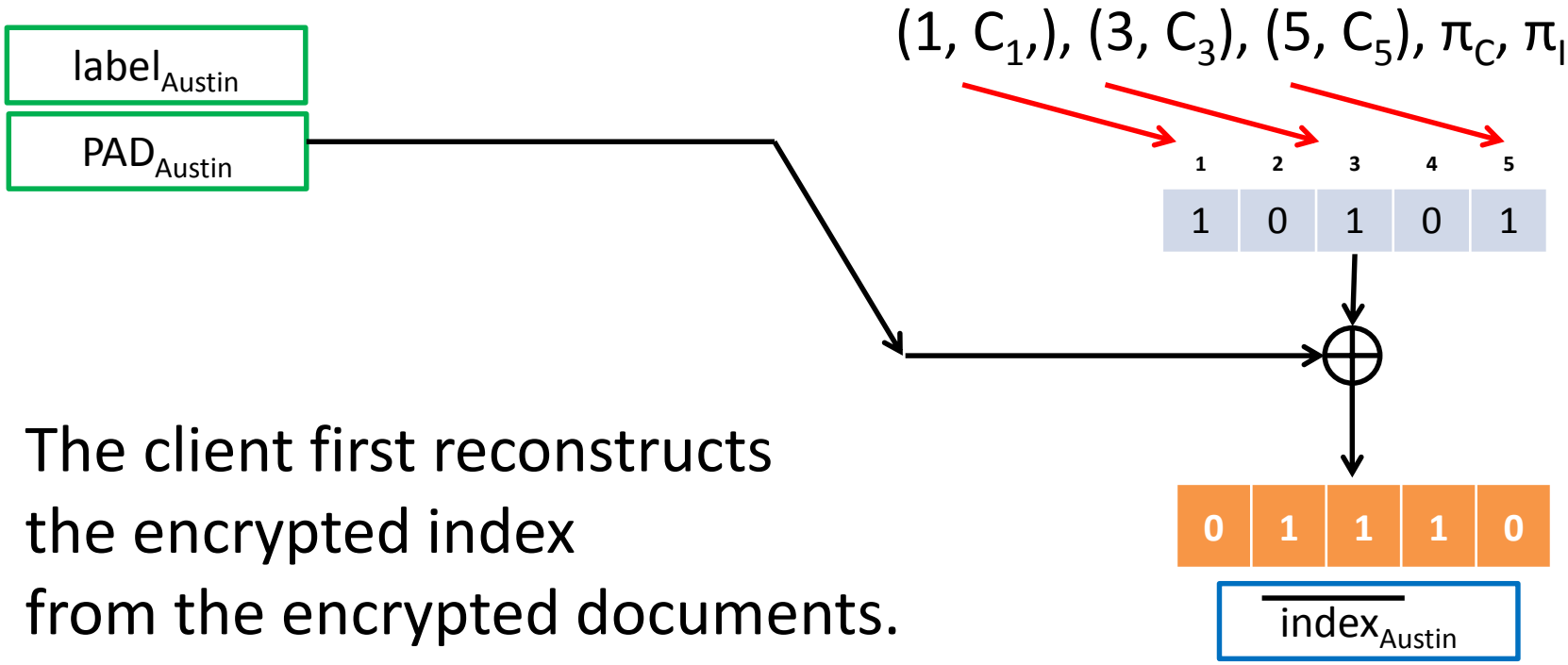
$$\Delta_I = g^{p_{2,1}\cdots p_{2,5}\ p_{3,1}\cdots p_{3,5}} \bmod N$$

Search phase : Sending the results

The server send the encrypted documents
and two witnesses
to the client.



Search phase: Verify (1/4)

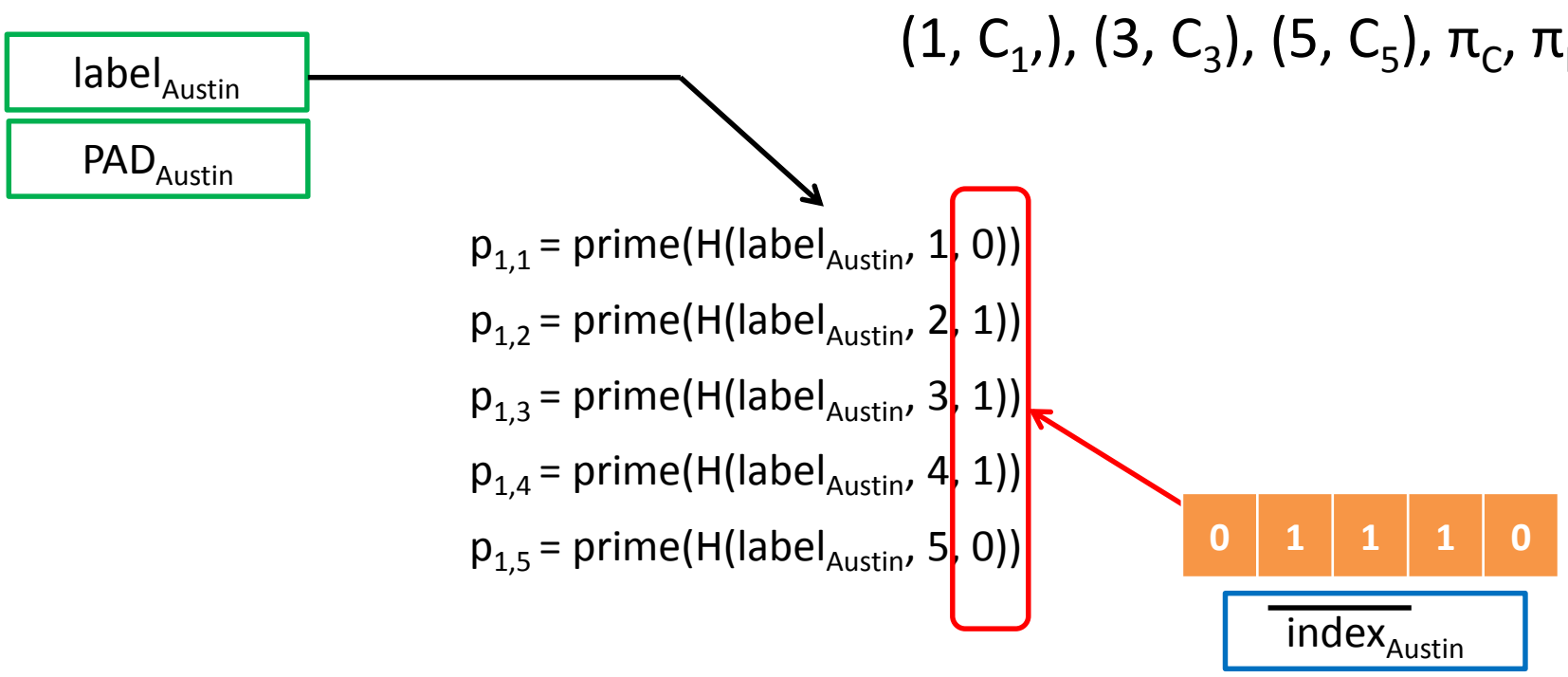


The client first reconstructs
the encrypted index
from the encrypted documents.

A_C

A_I

Search phase: Verify (2/4)

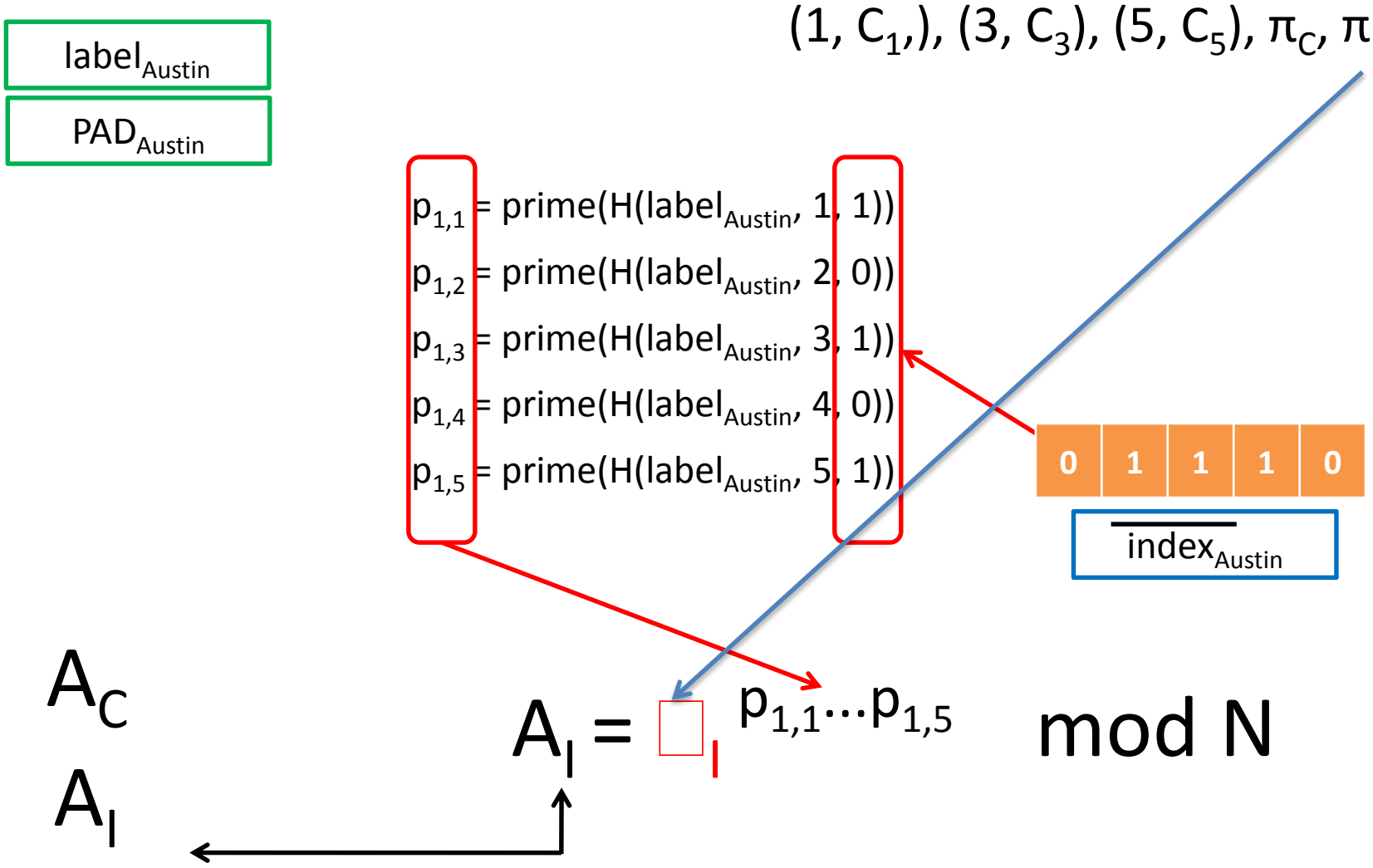


A_C

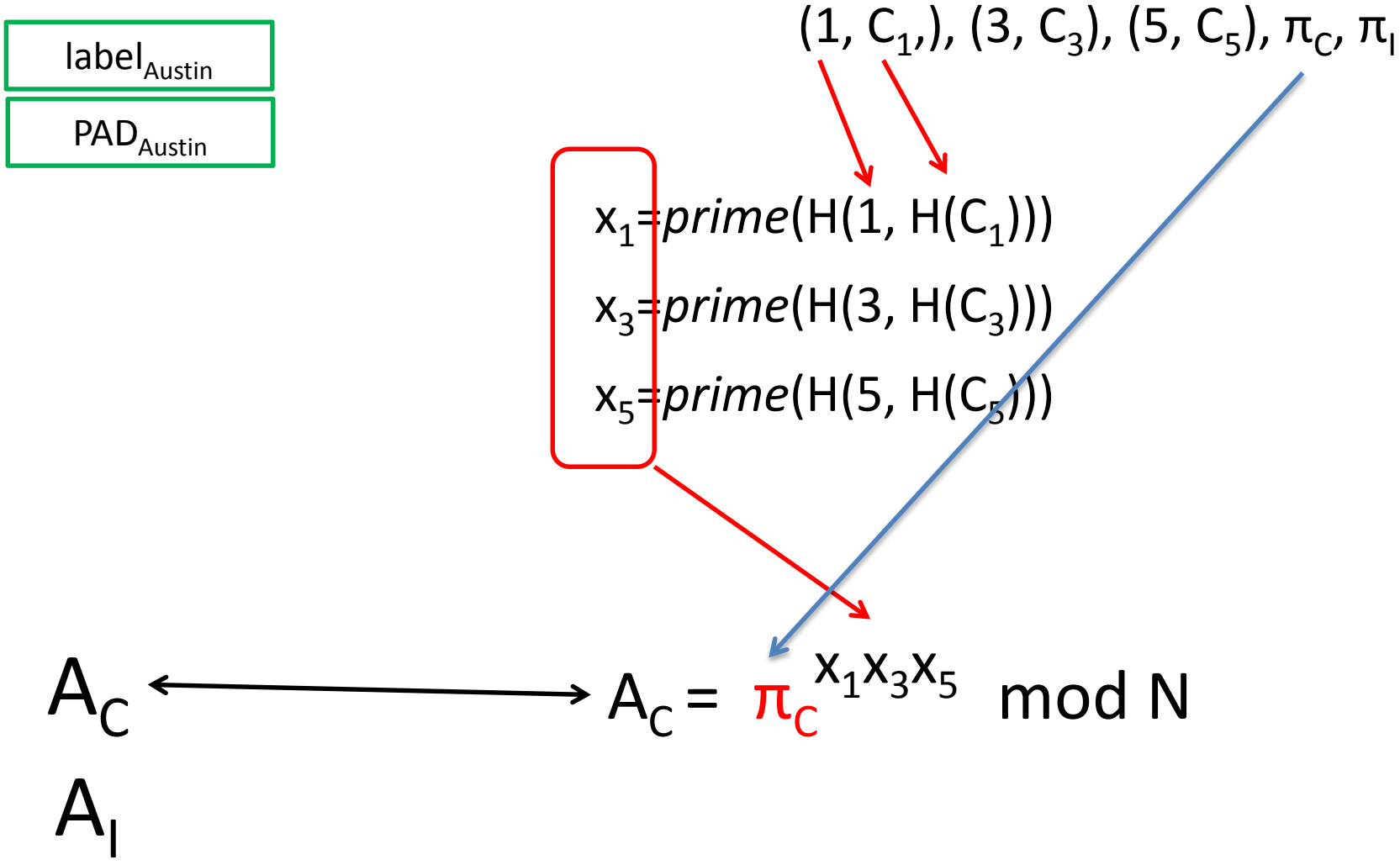
A_I

Then he can compute the primes
for the index elements related to the keyword.

Search phase: Verify (3/4)

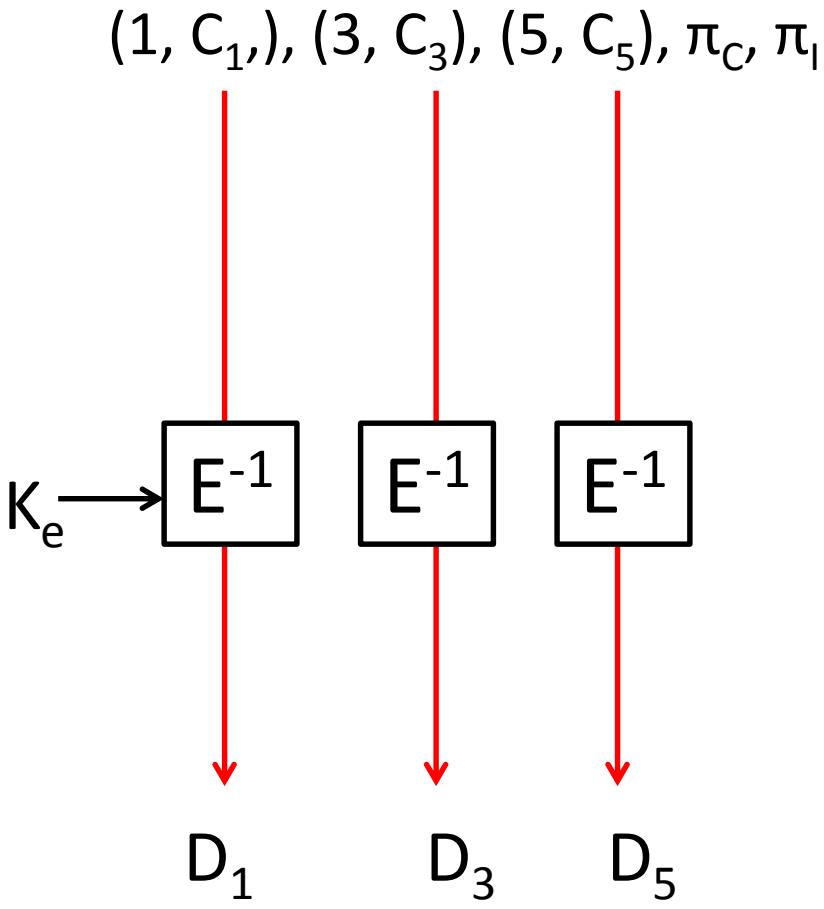


Search phase: Verify (4/4)



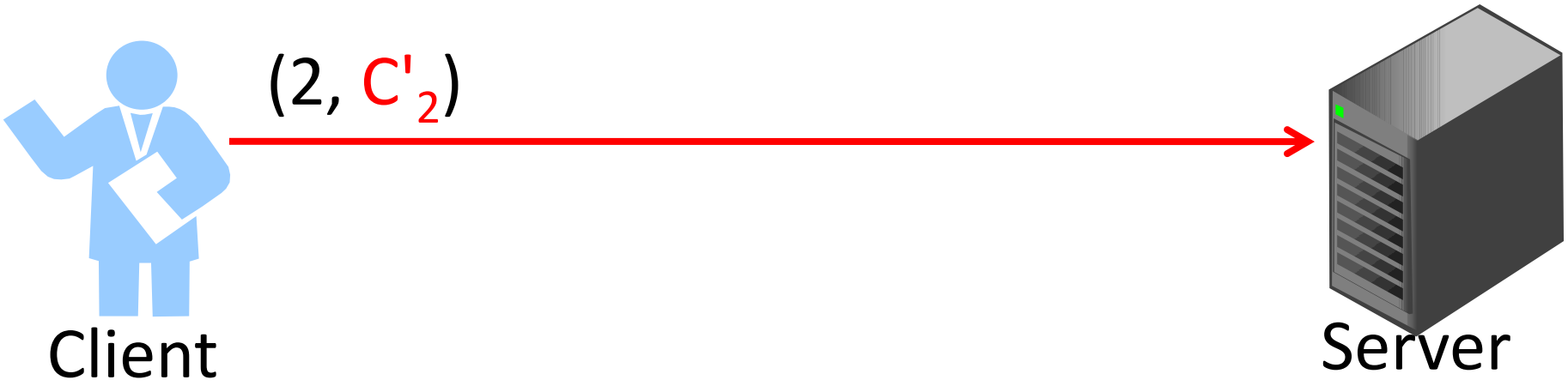
Search phase : Decrypt

- If all the checks succeed, then the client decrypts and outputs the documents.
- otherwise outputs *reject*.



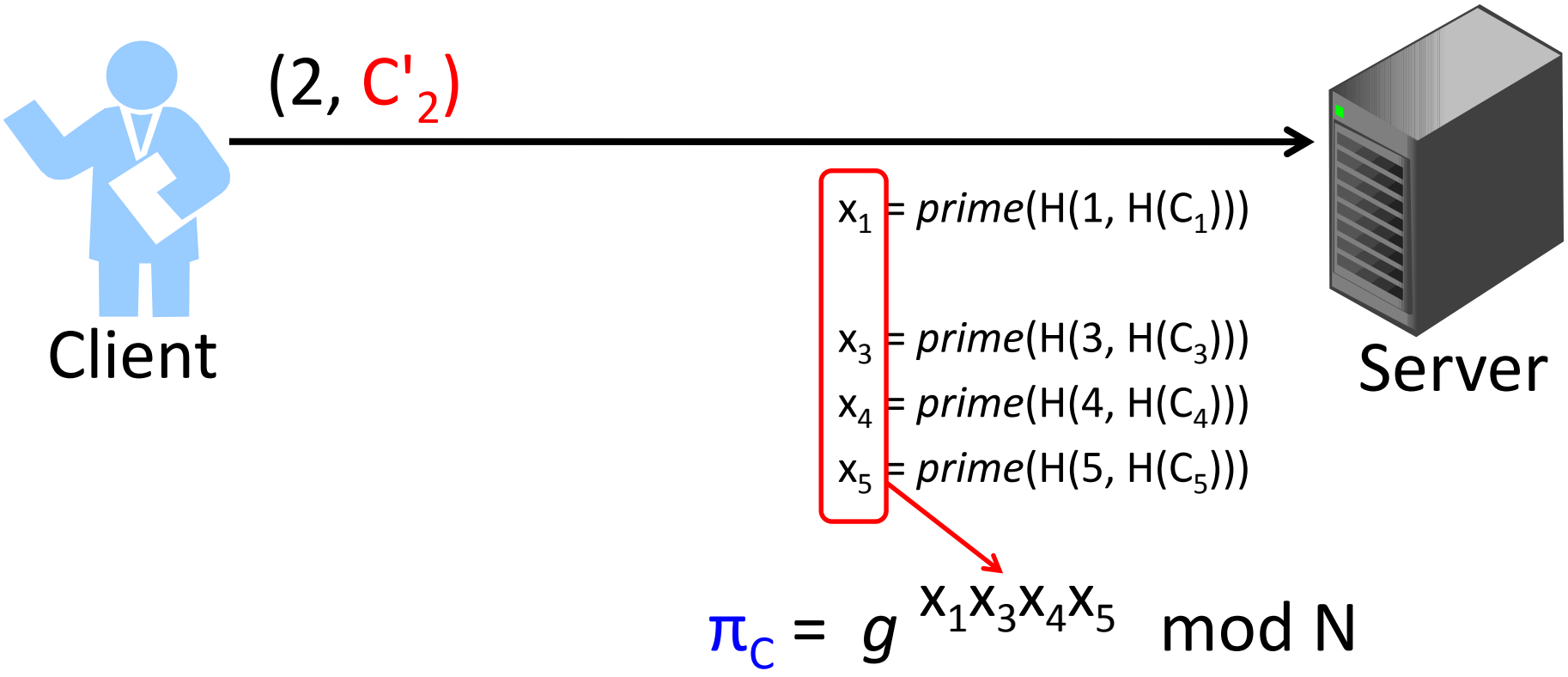
How to modify C_2 to C'_2 (1/5)

- the client sends (2, C'_2) to the server.



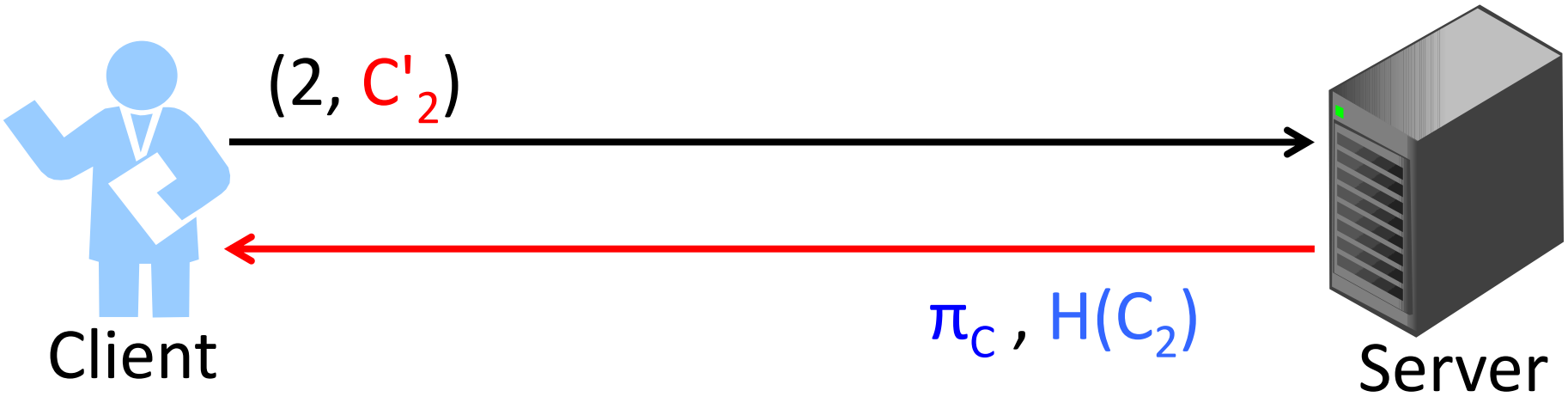
How to modify C_2 to C'_2 (2/5)

- the server computes π_C.



How to modify C_2 to C'_2 (3/5)

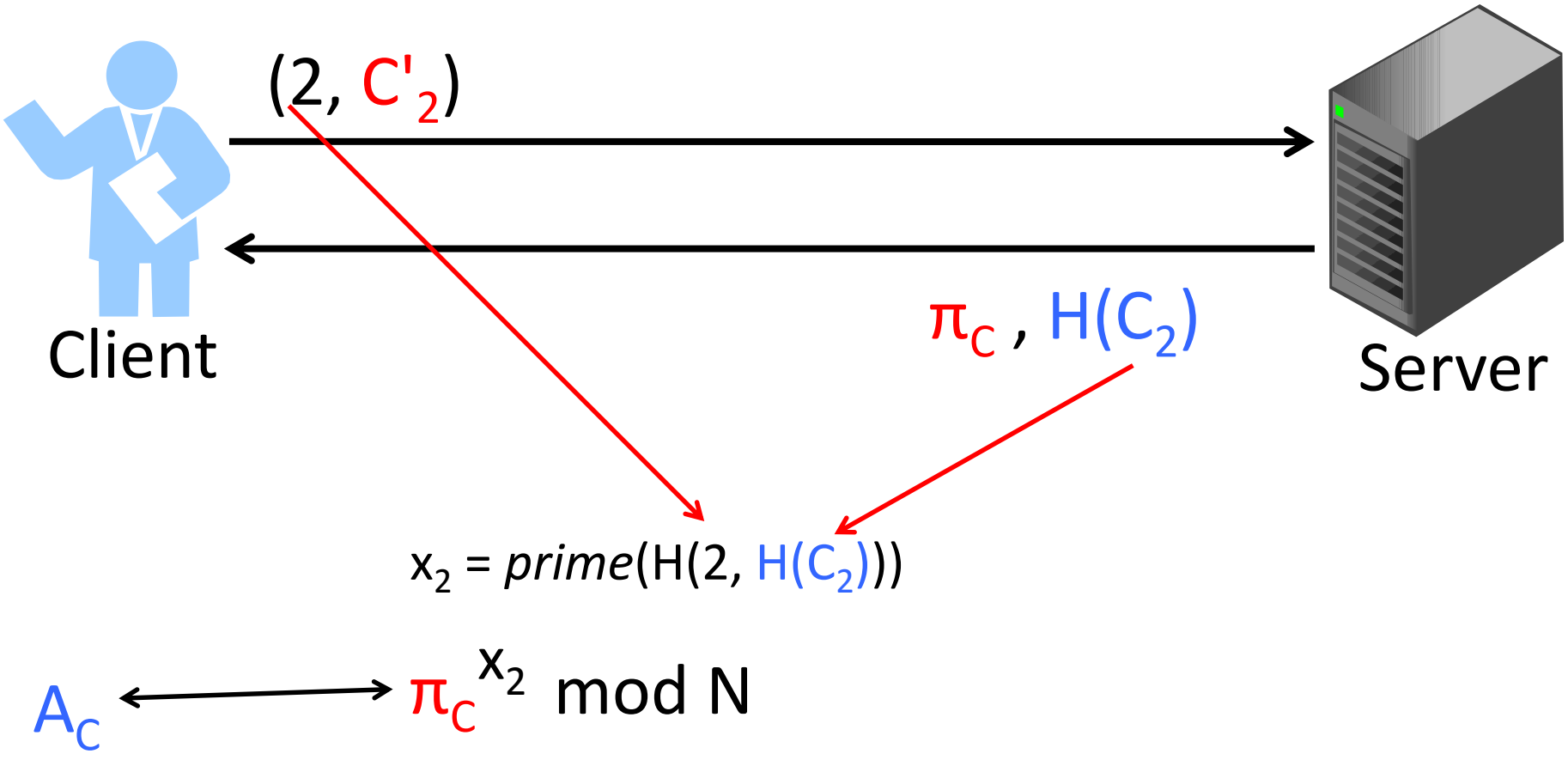
- the server returns π_C and $H(C_2)$ and then updates $(2, C_2)$ to $(2, C'_2)$.



$$\pi_C = g^{x_1 x_3 x_4 x_5} \bmod N$$

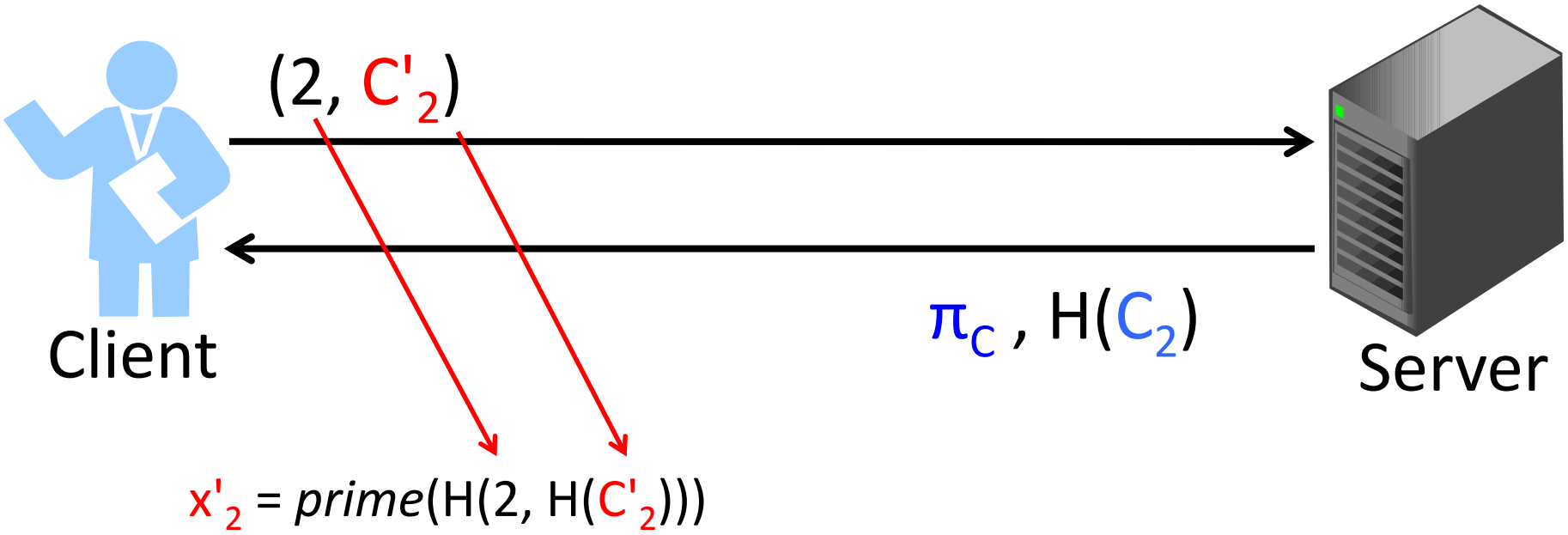
How to modify C_2 to C'_2 (4/5)

- the client verifies the latest C_2 the server had



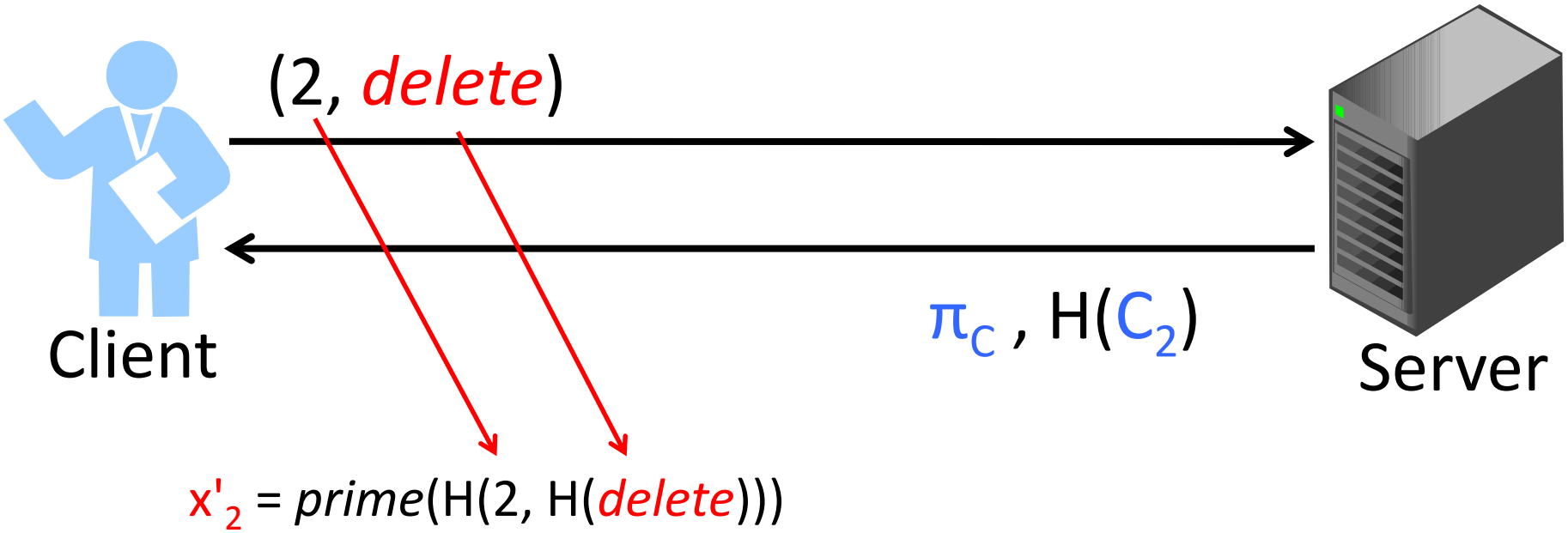
How to modify C_2 to C'_2 (5/5)

- and then updates A_c to A'_C



How to *delete* C_2

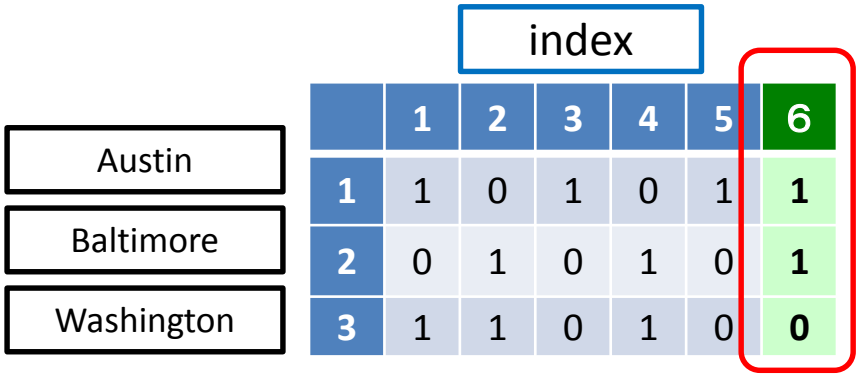
- Do the *modify* with C'_2 = *delete*.



A'_C ⟵ $A'_C = \pi_C^{x'_2} \bmod N$

To *add* a new document D_6 (1/5)

$C_6 = E_{Ke}(D_6)$

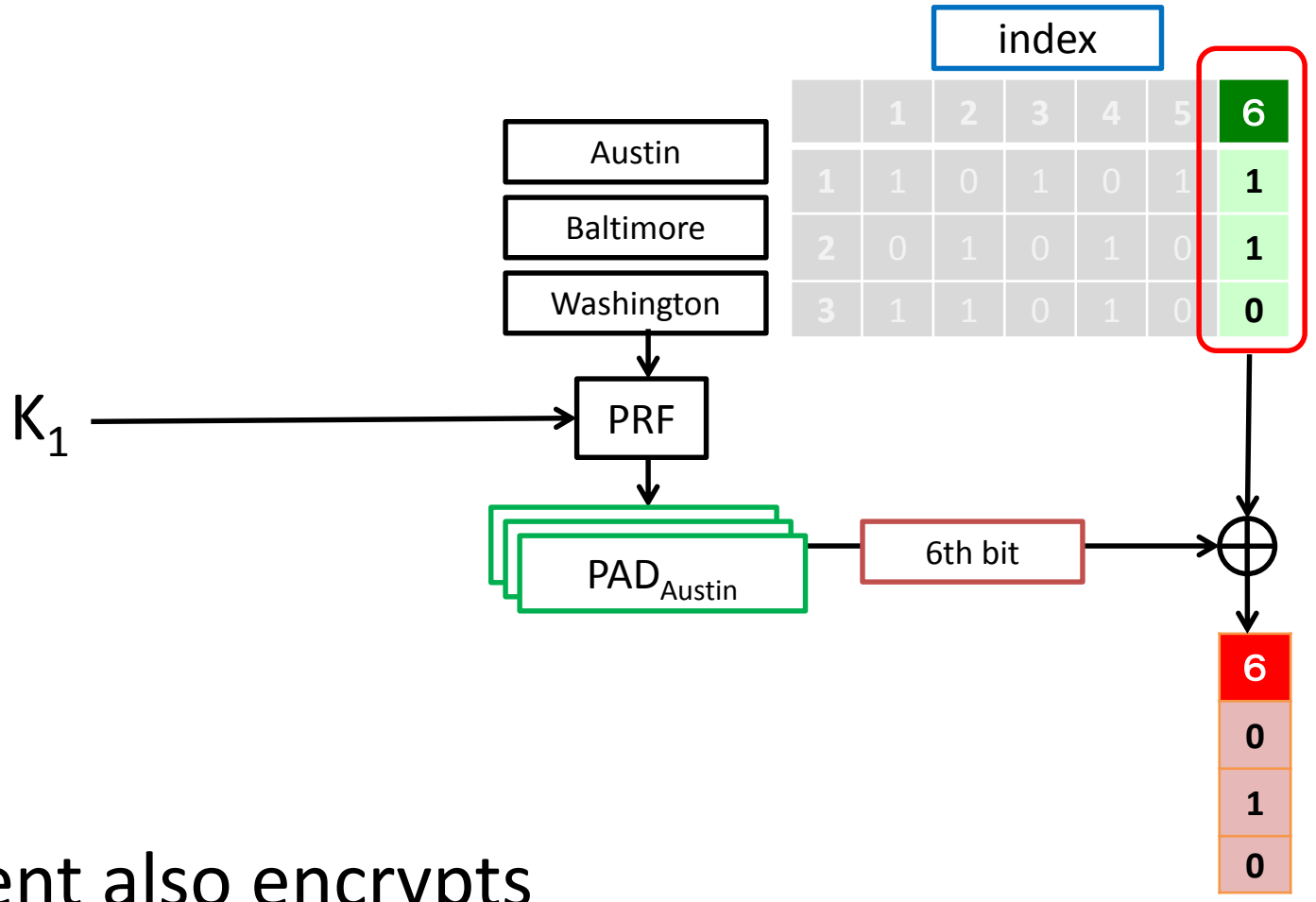


$x_6 = prime(H(6, H(C_6)))$

$A'_C \longleftarrow A'_C = A_C^{x_6} \mod N$

- The client encrypts the document and assign a new prime x_6 and updates A_C to A'_C.

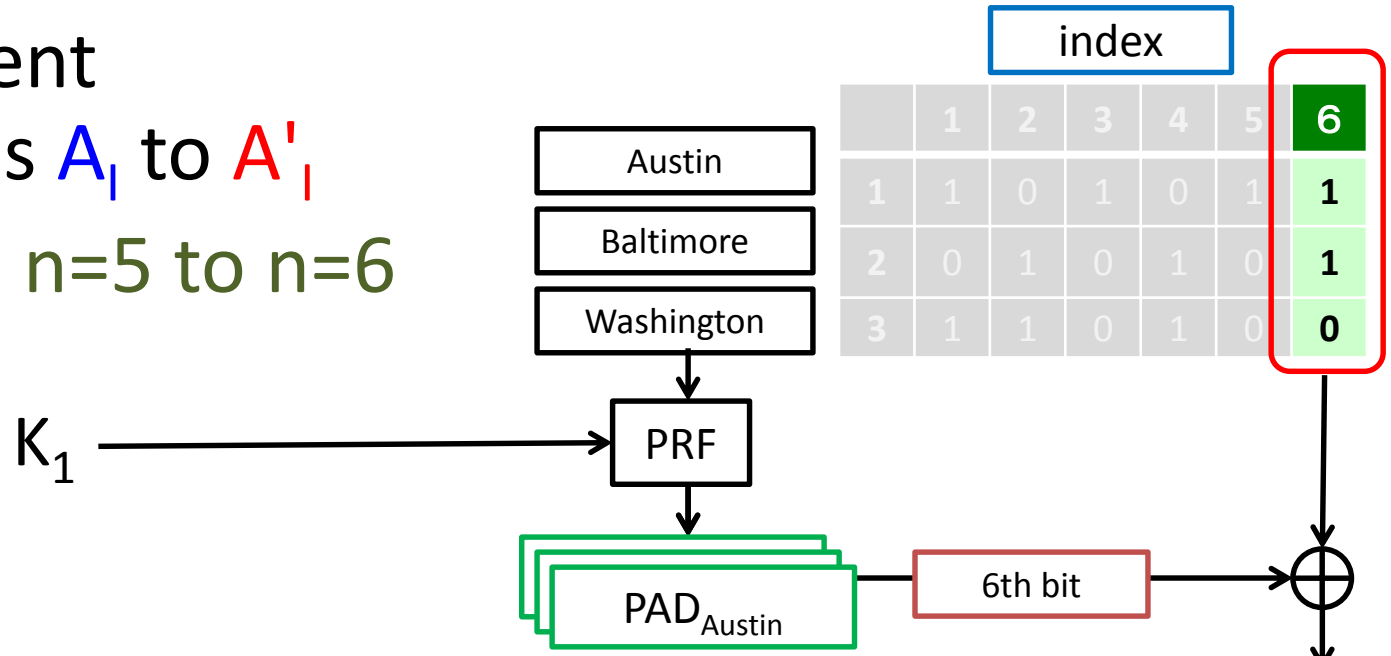
To *add* a new document D_6 (2/5)



- The client also encrypts
  the index elements related to the new document.

To *add* a new document D_6 (3/5)

- The client updates A_l to A'_l
- update n=5 to n=6



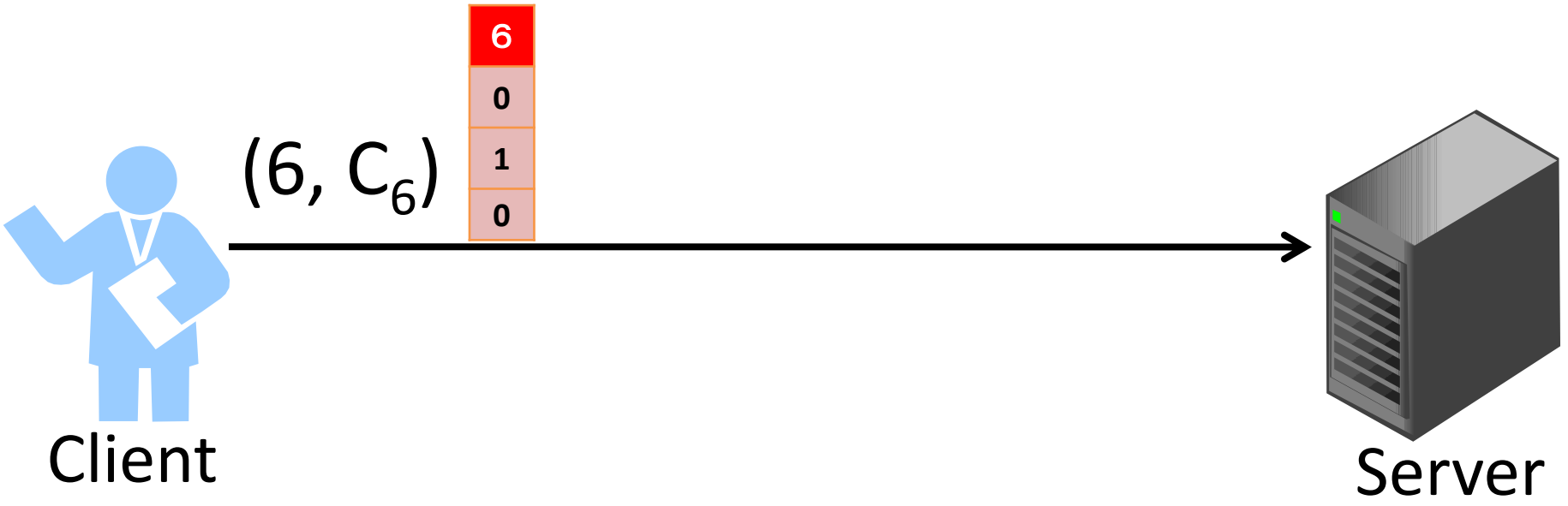
$p_{1,6}$= *prime*(H($label_{Austin}$, 6, 0))

$p_{2,6}$= *prime*(H($label_{Baltimore}$, 6, 1))

$p_{3,6}$= *prime*(H($label_{Washington}$, 6, 0))

$$A_l' = A_l^{p_{1,6}p_{2,6}p_{3,6}} \mod N$$

To *add* a new document D_6 (4/5)

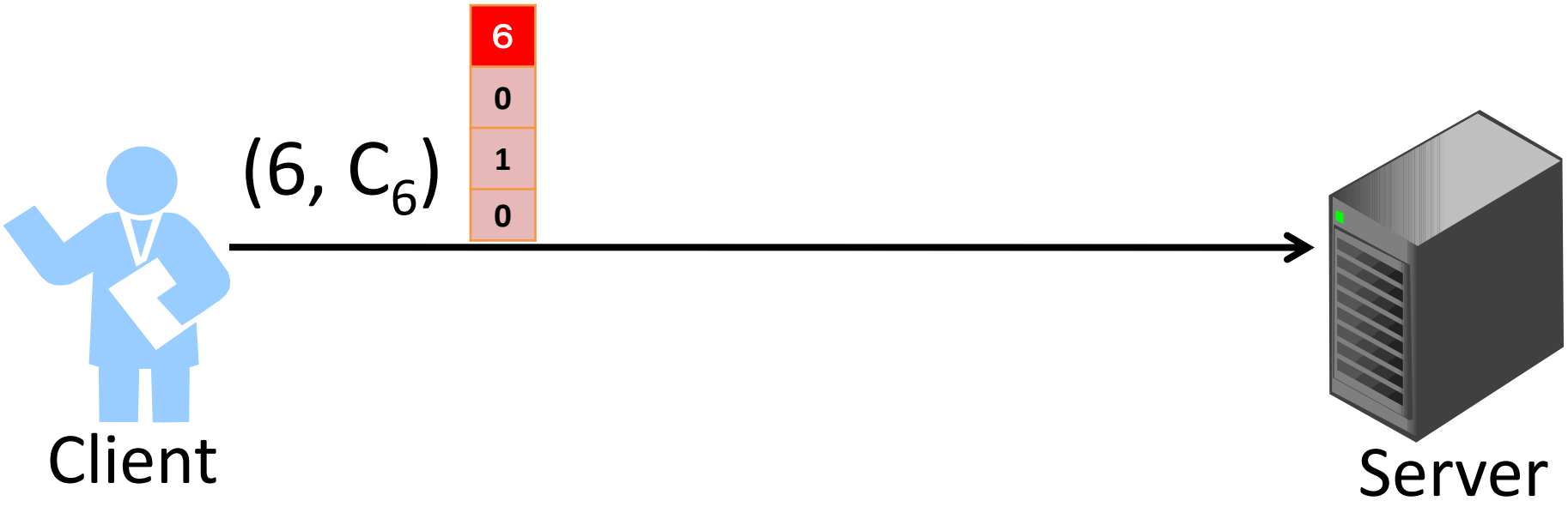


- Finally the client sends the encrypted document and the corresponding index.

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	1
3	1	0	1	0	0

index

To *add* a new document D_6 (5/5)



- The server
  stores the encrypted document and
  append the index to the 6th-column.

	1	2	3	4	5	6
1	0	1	1	1	0	0
2	1	0	0	1	1	1
3	1	0	1	0	0	0

index

Security

- The proposed scheme is UC-secure against non-adaptive adversaries.
- Assumptions
  - the strong RSA assumption
  - SKE is CPA-secure
  - PRF is a pseudo-random function
  - H is a collision-resistant hash function

Please refer to the paper for the details.

Efficiency

	search	modify	delete	add
communication overhead	n+O(λ)	O(λ)	O(λ)	m
computation cost of the Server	O(nm)	O(n)	O(n)	O(m)
computation cost of the Client	O(n)	O(1)	O(1)	O(m)

λ : security parameter
n : number of Documents
m : number of Keywords

Summary

- Proposed a Verifiable Dynamic SSE scheme
- Construction based on RSA Accumulator
- UC-Secure

Thank you