Sophos and Diane

Searchable Symmetric Encryption with (Very) Low Overhead

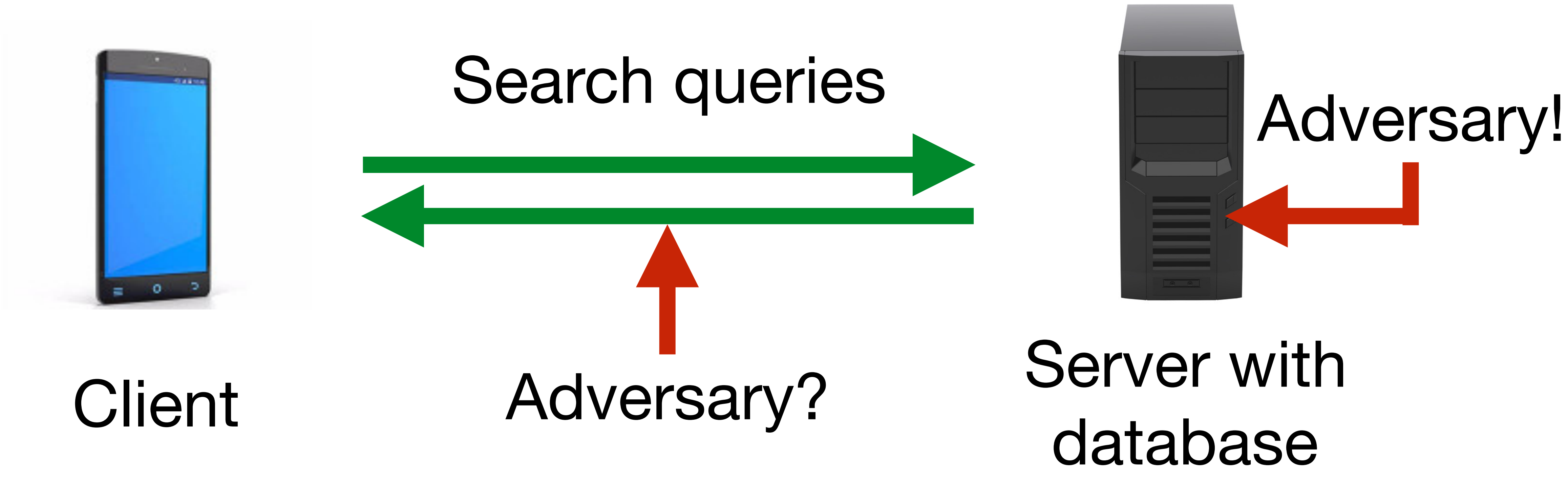
Raphael Bost, Brice Minaud

RHUL ISG seminar, November 24th 2016

Plan

1. Symmetric Searchable Encryption.
2. Leakage and Forward-Privacy.
3. Sophos and Diane schemes.
4. Proof Models.

Symmetric Searchable Encryption



Search queries

Adversary!

Client

Adversary?

Server with
database

- Client stores encrypted database on server.
- Client can perform search queries.
- Privacy of data and queries is retained.

Example: private email storage.

- *Dynamic* SSE: also allows update queries.

Symmetric Searchable Encryption

Two databases:

- **Document** database.

  Encrypted documents d_i for $i \leq D$.

- (Reverse) **Index** database DB.

  Pairs (w,i) for each keyword w and each document index i such that d_i contains w.

$$\mathrm{DB} = \{(w,i) : w \in d_i\}$$

Symmetric Searchable Encryption

- **Search**(w) query:

  Retrieve DB(w) = $\{i : w \in d_i\}$.

- **Update**(w,i) query:

  Add (w,i) to DB.

After getting DB(w) from a **search** query, the client is likely to retrieve documents in DB(w) from the **document** database.

- This leaks DB(w).

Is leakage necessary?

Leaking DB(***w***) for search queries is nearly unavoidable.

In a nutshell, ORAM approaches either leak it or are very inefficient [Nav15].

Note: still feasible in some restricted settings.

How bad is leakage?

- Assume a priori knowledge of frequency and correlation of keywords.
  - ▹ IKK12 (NDSS'12) and CGPR15 (CSS'15) show how to identify (most) keywords.

- Assume the adversary can inject arbitrary documents.
  - ▹ CGPR15 and ZKP16 (USENIX Sec'16) show how to immediately identify searched keywords.

File injection

	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7
File A	✓	✓	✓	✓				
File B			✓	✓			✓	✓
File C		✓		✓		✓		✓

Idea of ZKP16: for *W* keywords, inject log(*W*) files containing W/2 keywords each as above.

When **Search**(w) is searched, DB(w) directly leaks w.

E.g. DB(w) contains A, B but not C, then $w = w_2$.

Adaptive file injection

Proposed countermeasure: at most T keywords/file.

▹ Attacke requires $(K/T) \cdot \log(T)$ injections.

Adaptive version: enhancement of frequency attack:

▹ Adaptive attack requires less injections, e.g. $\log(T)$, assuming some prior knowledge.

This last attack uses update leakage:

Most SE schemes leak if a newly inserted document matches a **previous** search query.

▹ Need **forward privacy**: oblivious updates.

Forward Privacy

Forward privacy: **Update** queries leak nothing.

- The encrypted database can be securely built online.
- Only one existing scheme SPS14 (NDSS'14):

  ORAM-like construction.

  Inefficient updates.

  Large client storage.

Sophos (Σοφος) and Diane

Sophos: introduced at CCS'16 [Bost16]:

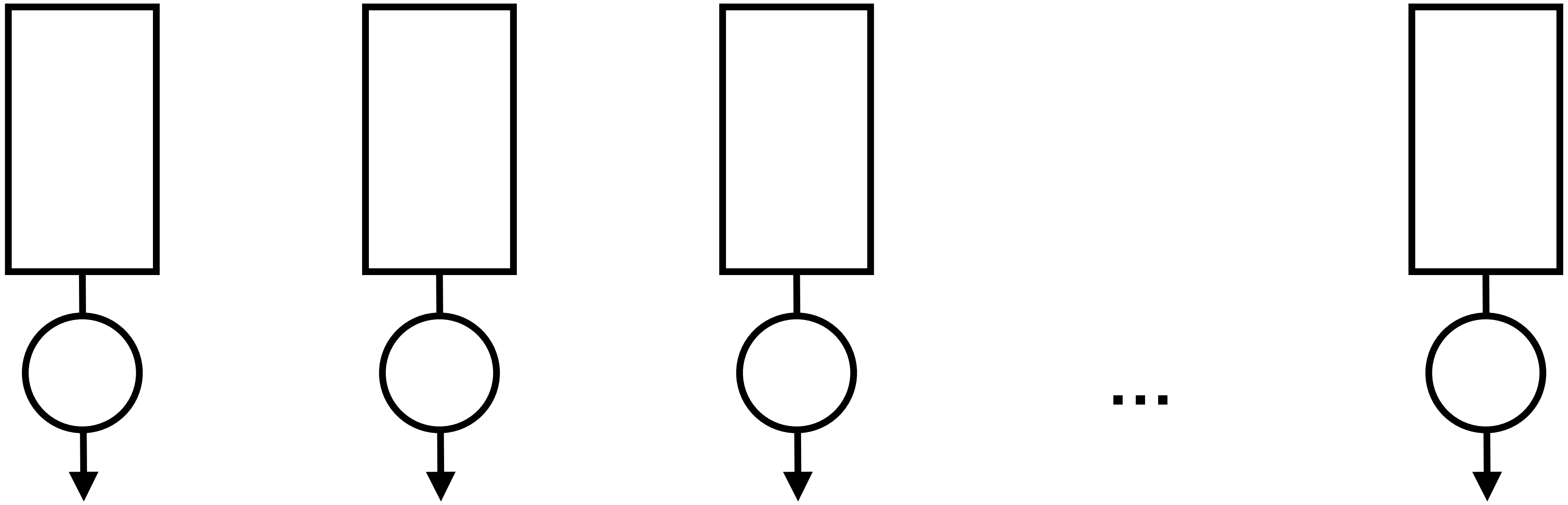
- Dynamic, forward-private SSE scheme.
- Low overhead.
- Simple.

Diane: work-in-progress.

Sophos (Σοφος)

Fix a keyword w.

Let i_k be the k-th document containing w.



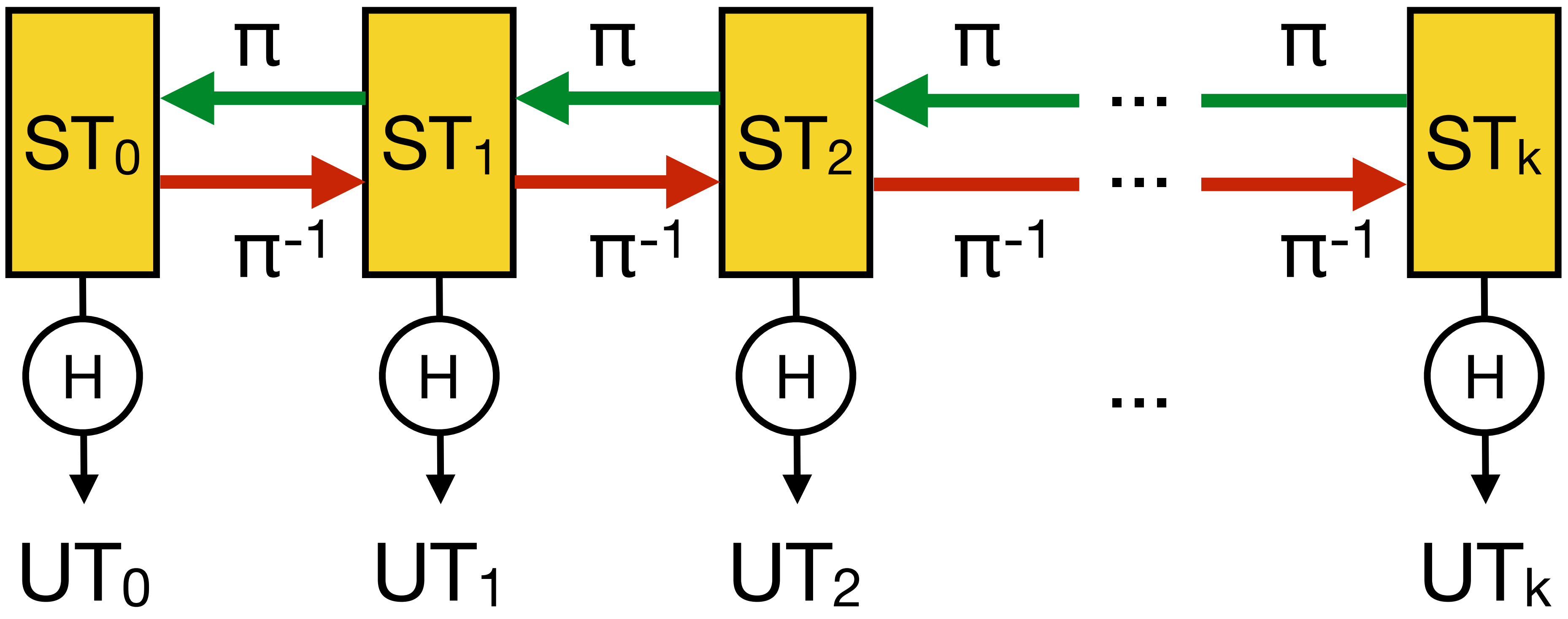
UT_0 UT_1 UT_2 … UT_k

DB stores enc(i_k) at position UT_k.

Sophos (Σοφος)

Fix a keyword ***w***.

Let $\boldsymbol{i}_k$ be the k-th document containing ***w***.



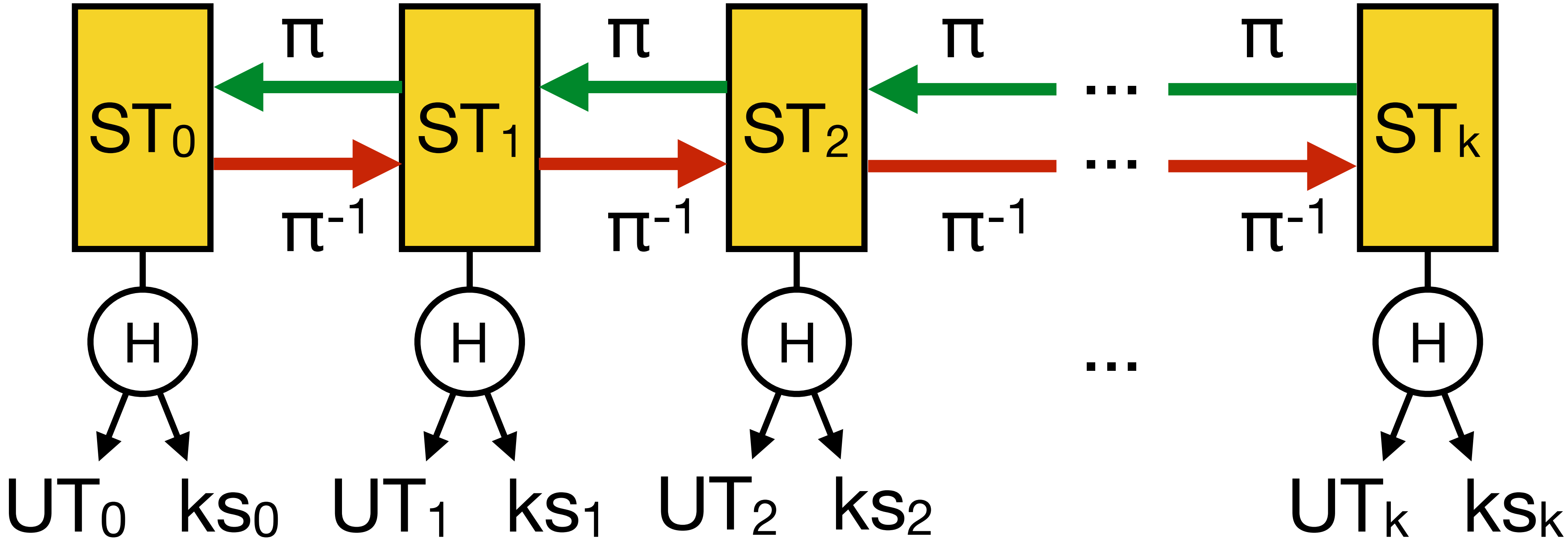
DB stores enc($\boldsymbol{i}_k$) at position UT_k.

Let π be a trapdoor permutation (e.g. RSA).

Sophos (Σοφος)

Fix a keyword ***w***.

Let $\boldsymbol{i}_k$ be the k-th document containing ***w***.



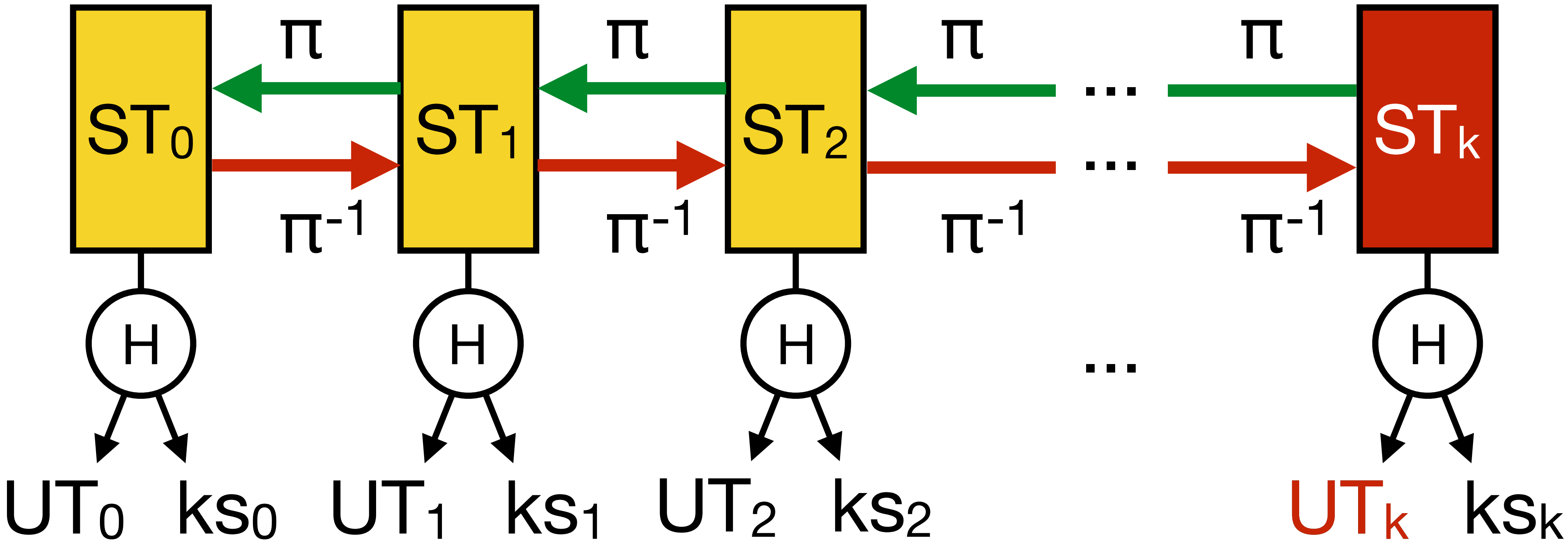
DB stores $\mathrm{enc}(\boldsymbol{i}_k) = \boldsymbol{i}_k \oplus ks_k$ at position UT_k.

Let π be a trapdoor permutation (e.g. RSA).

Sophos (Σοφος)

Fix a keyword ***w***.

Let i_k be the k-th document containing ***w***.



- **Update**(*w*,*i*): send (UT_k, $i \oplus ks_k$).
- **Search**(*w*): send ST_k.

Client Storage

Sophos assumes the client stores $c_{w} = |DB(w)|$ for every keyword.

- ▹ Client-side storage: W · log(D), with:

  W = #keywords D = #documents

This is enough!
Everything else is generated pseudo-randomly.

Nice feature of RSA:

$$x^{d \cdot d \cdots d} = x^{d^c \bmod \phi(N)} \bmod N$$

Makes computing ST_c faster.

Summary of Sophos

	Computation		Communication		Client Storage	FS
	Update	Search	Update	Search		
[CJJ+14]	$\mathcal{O}(1)$	$\mathcal{O}(c_w)$	$\mathcal{O}(1)$	$\mathcal{O}(c_w)$	$\mathcal{O}(1)$	✗
[SPS14]	$\mathcal{O}(\log^2 N)$	$\mathcal{O}(c_w+\log^2 N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(c_w+\log N)$	$\mathcal{O}(N^a)$	✓
Sophos	$\mathcal{O}(1)$	$\mathcal{O}(c_w)$	$\mathcal{O}(1)$	$\mathcal{O}(c_w)$	$\mathcal{O}(W\log(D))$	✓

optimal

Leakage:

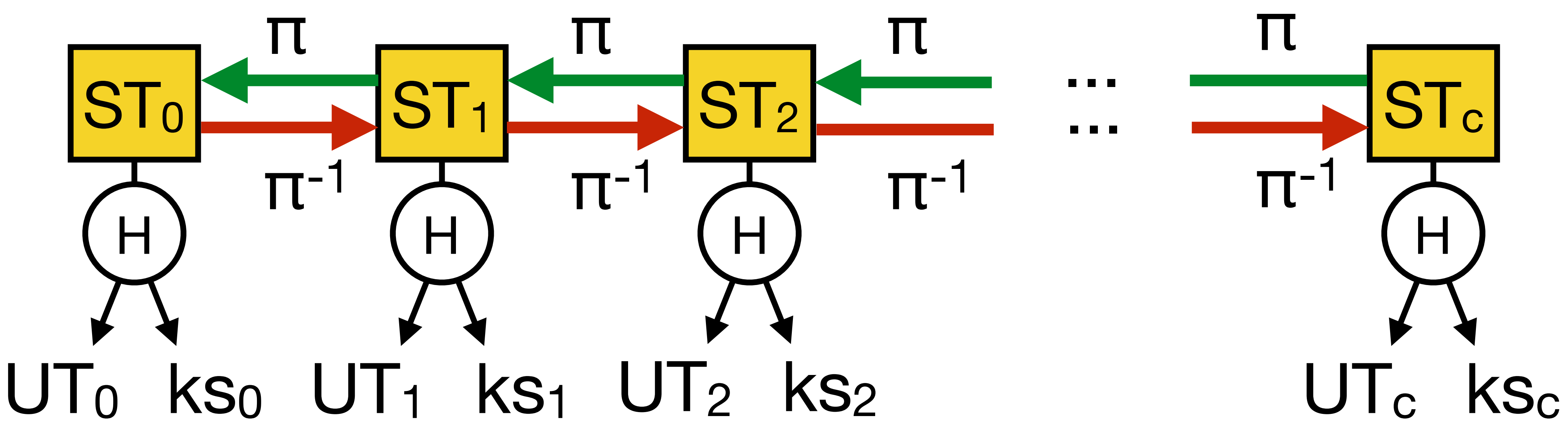
- $\mathcal{L}^{\text{Search}}(w) = \text{DB}(w)$ and content of previous search and update queries on w.
- $\mathcal{L}^{\text{Update}}(w,i) = \emptyset$. **Forward-private!**

Summary of Sophos

- Provable forward-privacy.
- Very simple.
- Efficient search (IO bounded).
- Asymptotically efficient update (optimal).

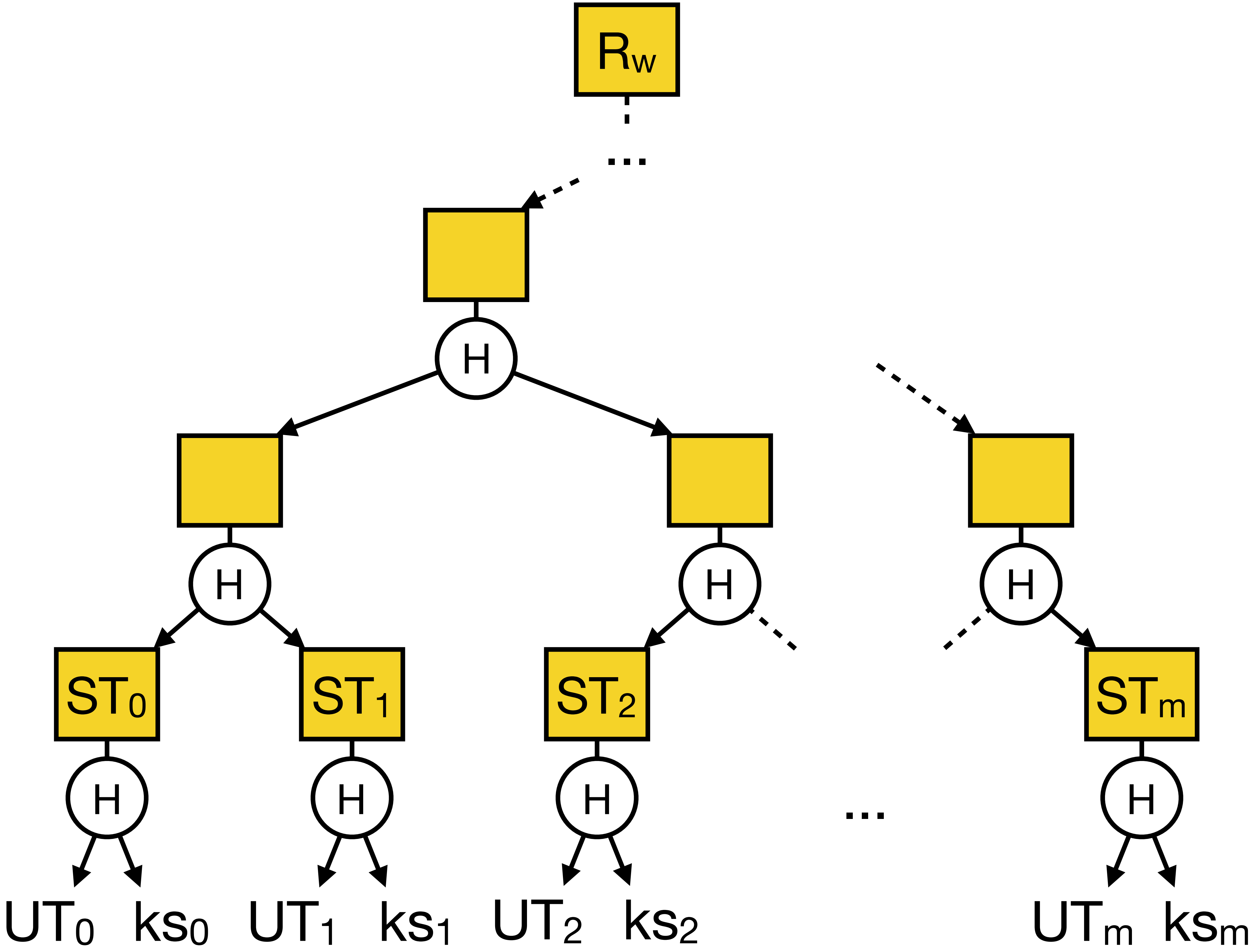
  In practice, very low update throughput (20x slower than prior work).

Diane

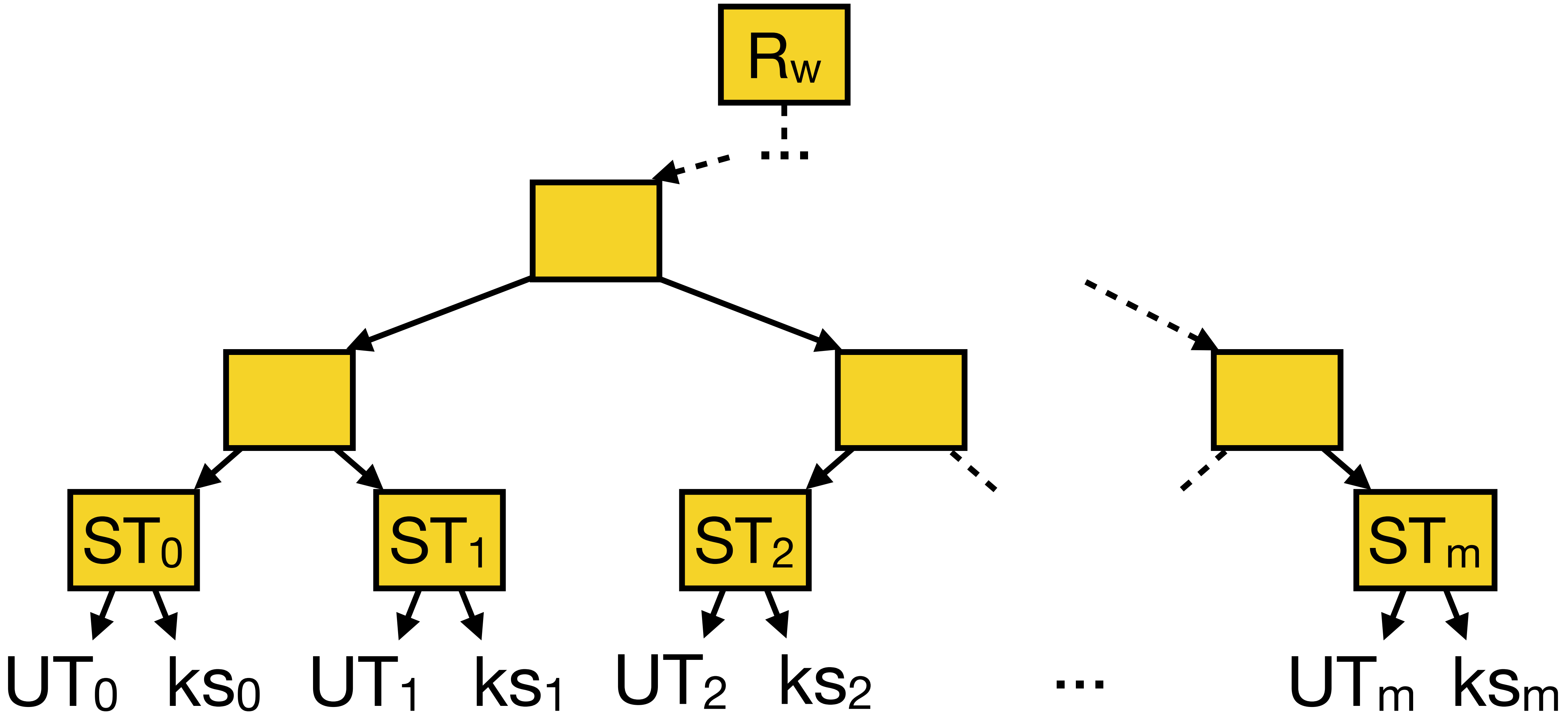


ST_0
ST_1
ST_2
ST_c
π
π
π
π
π^{-1}
π^{-1}
π^{-1}
π^{-1}
…
…
H
H
H
H
UT_0
ks_0
UT_1
ks_1
UT_2
ks_2
UT_c
ks_c

Diane



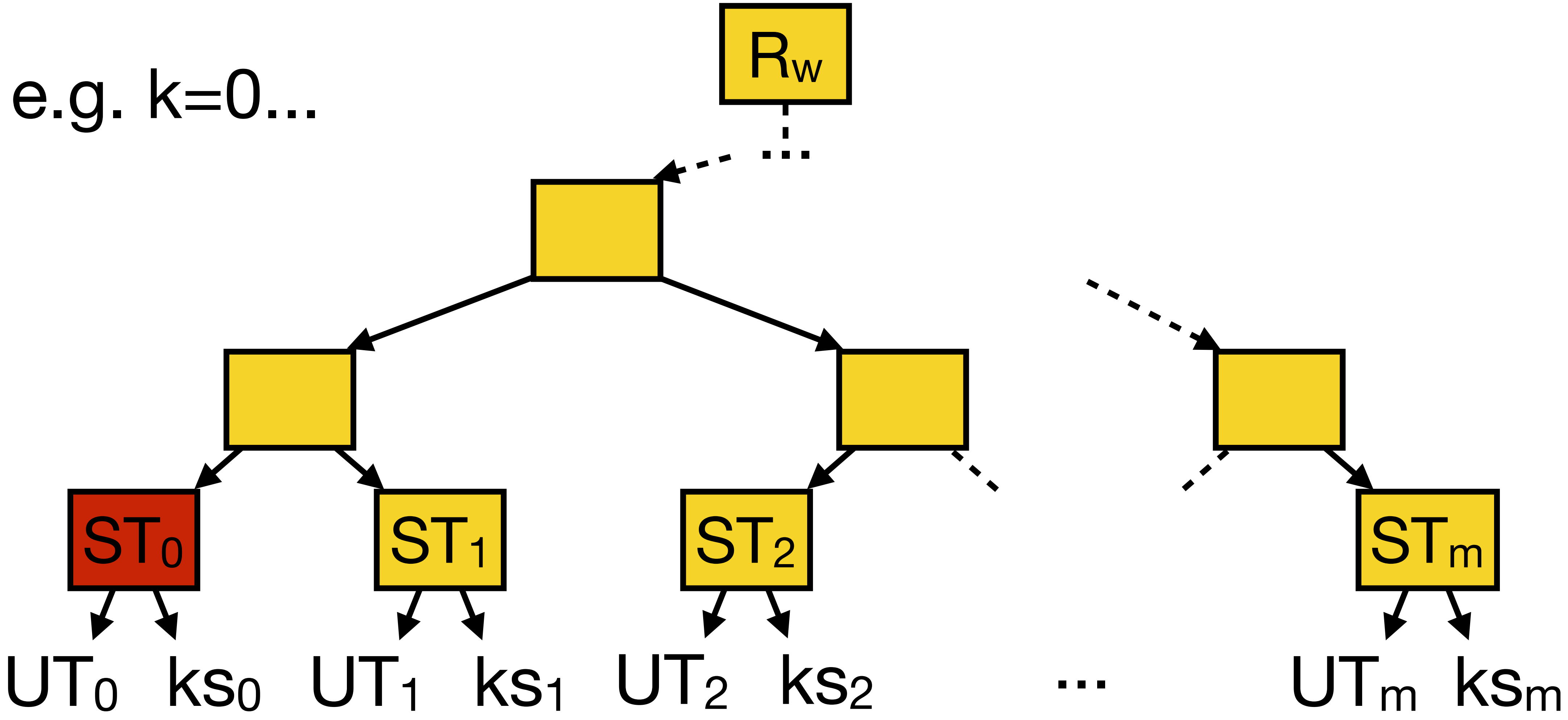
Diane



- **Update**(***w***,***i***): send (UT_c, ***i*** ⊕ ks_c).
- **Search**(***w***): send *covering set* of ST_0, ..., ST_c.

Diane

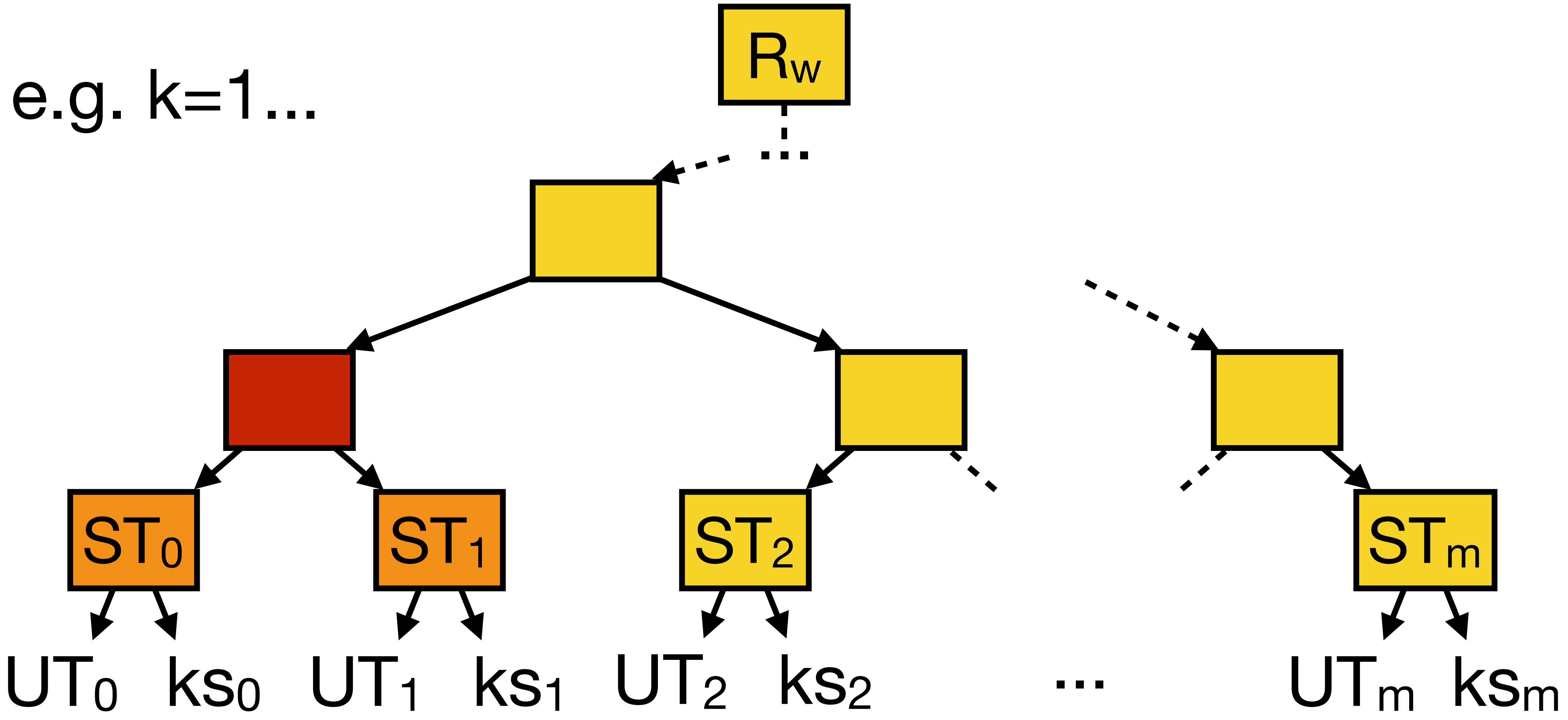
e.g. k=0...



- **Update**(*w*,*i*): send (UT_c, ***i*** $\oplus$ ks_c).
- **Search**(*w*): send *covering set* of ST_0, ..., ST_c.

Diane

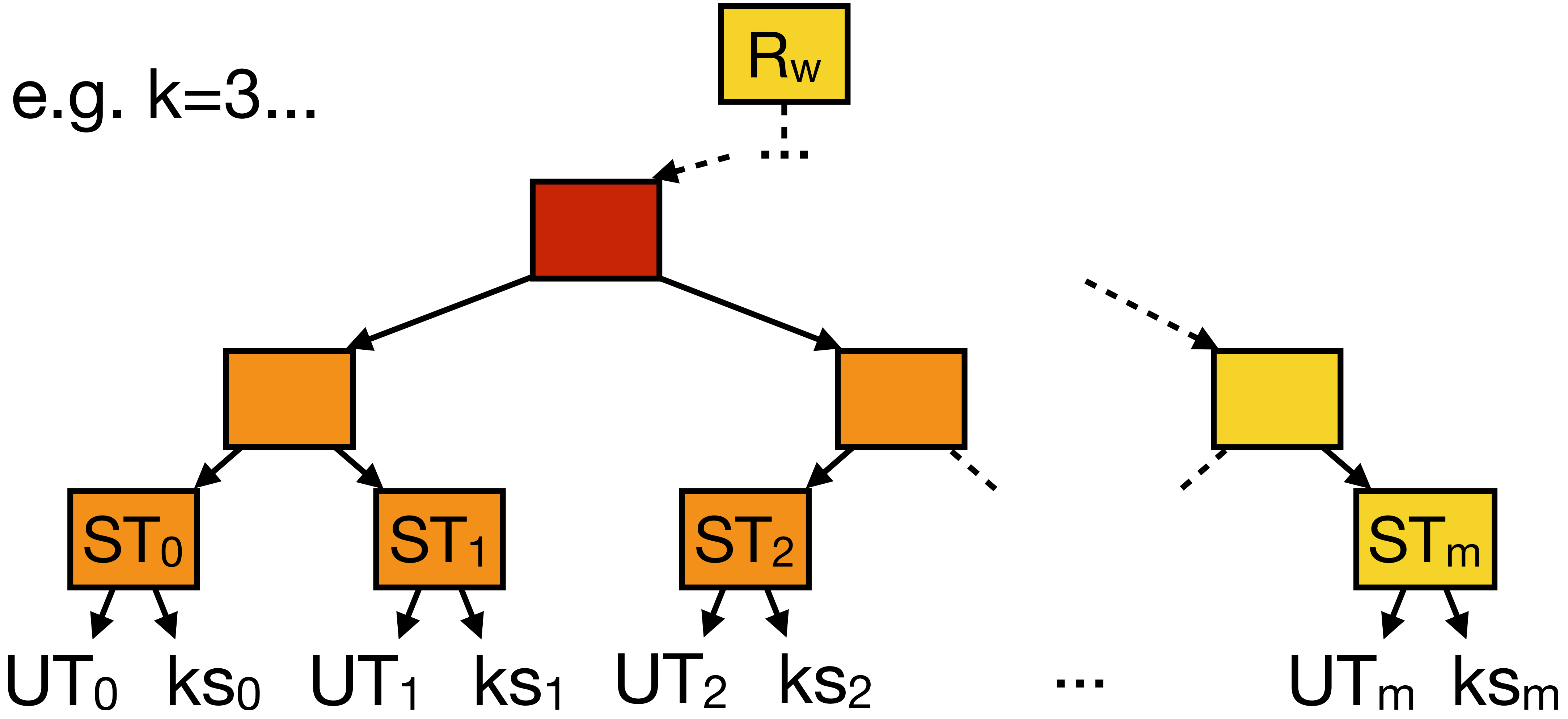
e.g. k=1...



- **Update**(***w***,***i***): send (UT_c, ***i*** $\oplus$ ks_c).
- **Search**(***w***): send *covering set* of ST_0, ..., ST_c.

Diane

e.g. k=3...



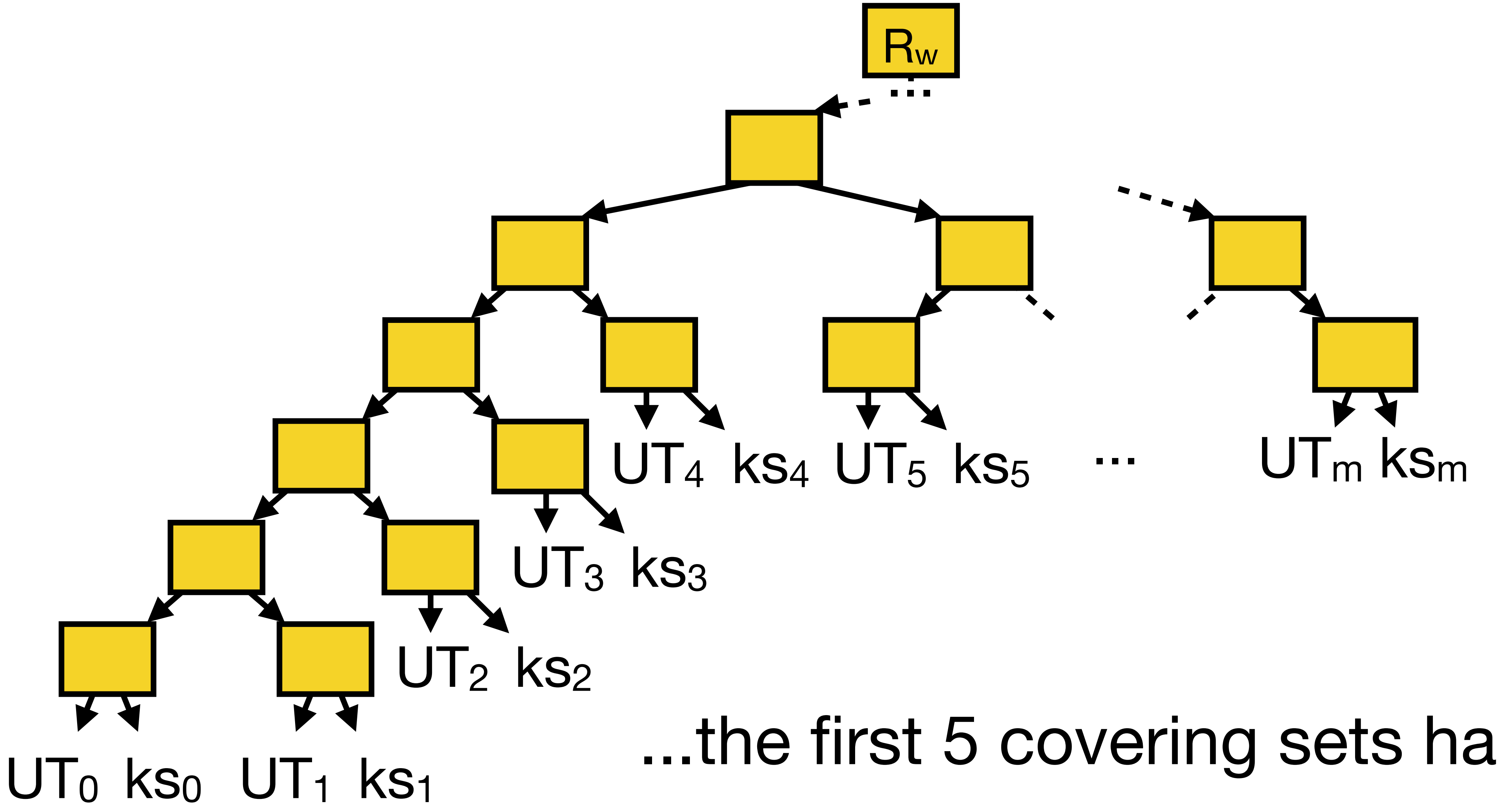
- **Update**(*w*,*i*): send (UT_c, *i* ⊕ ks_c).
- **Search**(*w*): send *covering set* of ST_0, ..., ST_c.

The size of the covering set is logarithmic in c.

Tweaking the Tree

The tree does not have to be balanced.

- e.g. if most keywords have ≤ 5 matches:

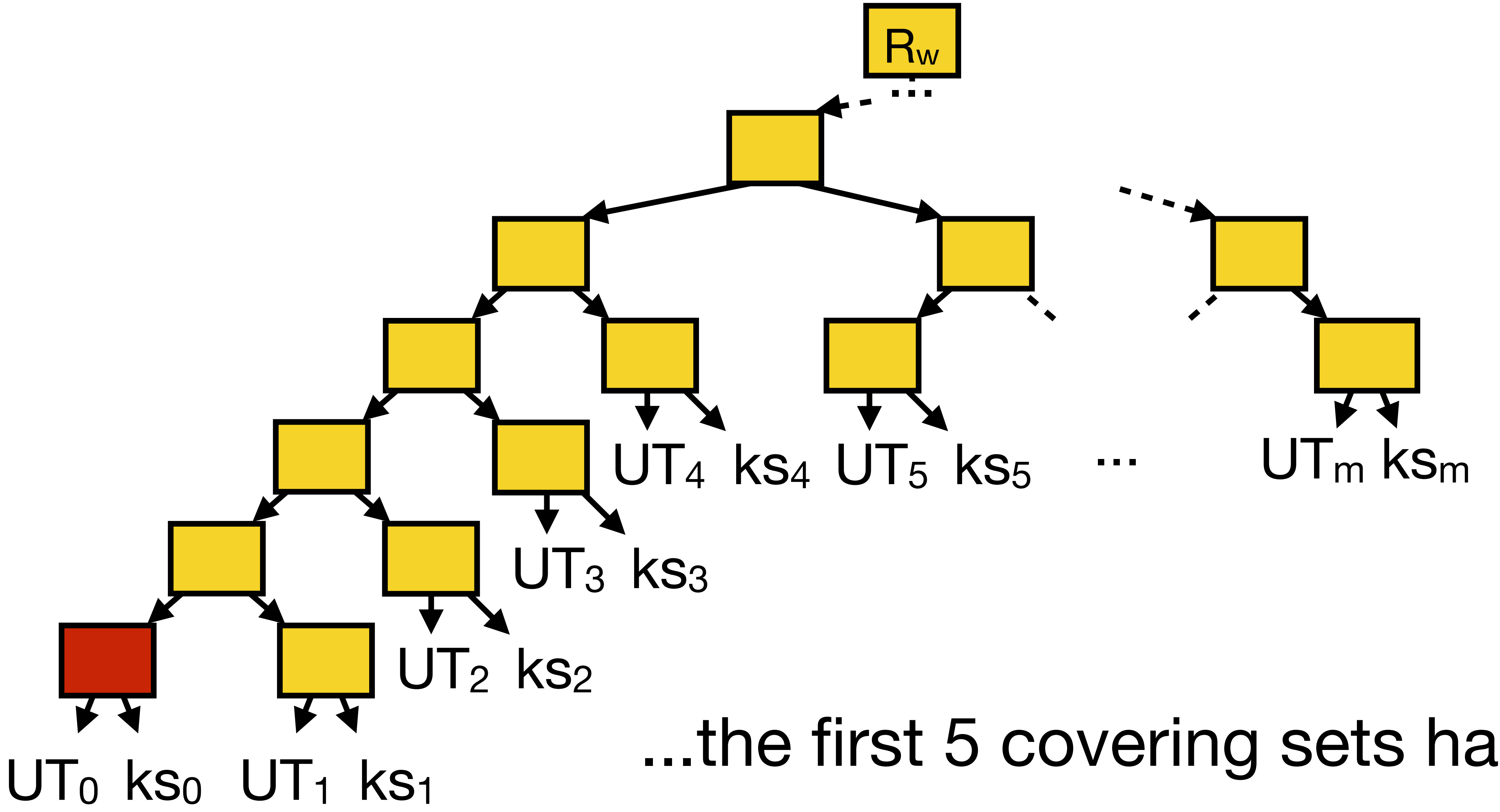


...the first 5 covering sets have size 1.

Tweaking the Tree

The tree does not have to be balanced.

- e.g. if most keywords have ≤ 5 matches:

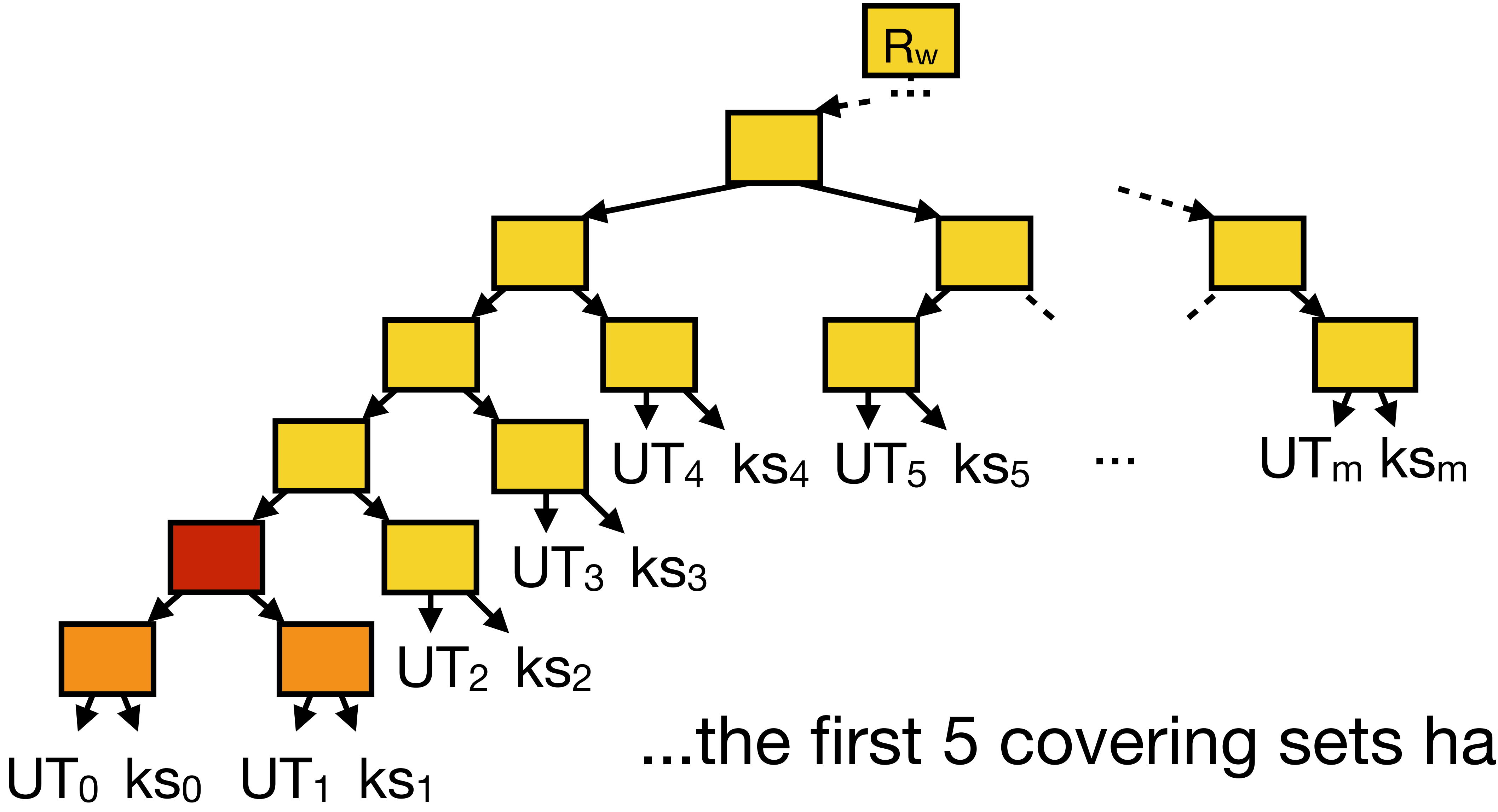


...the first 5 covering sets have size 1.

Tweaking the Tree

The tree does not have to be balanced.

- e.g. if most keywords have ≤ 5 matches:

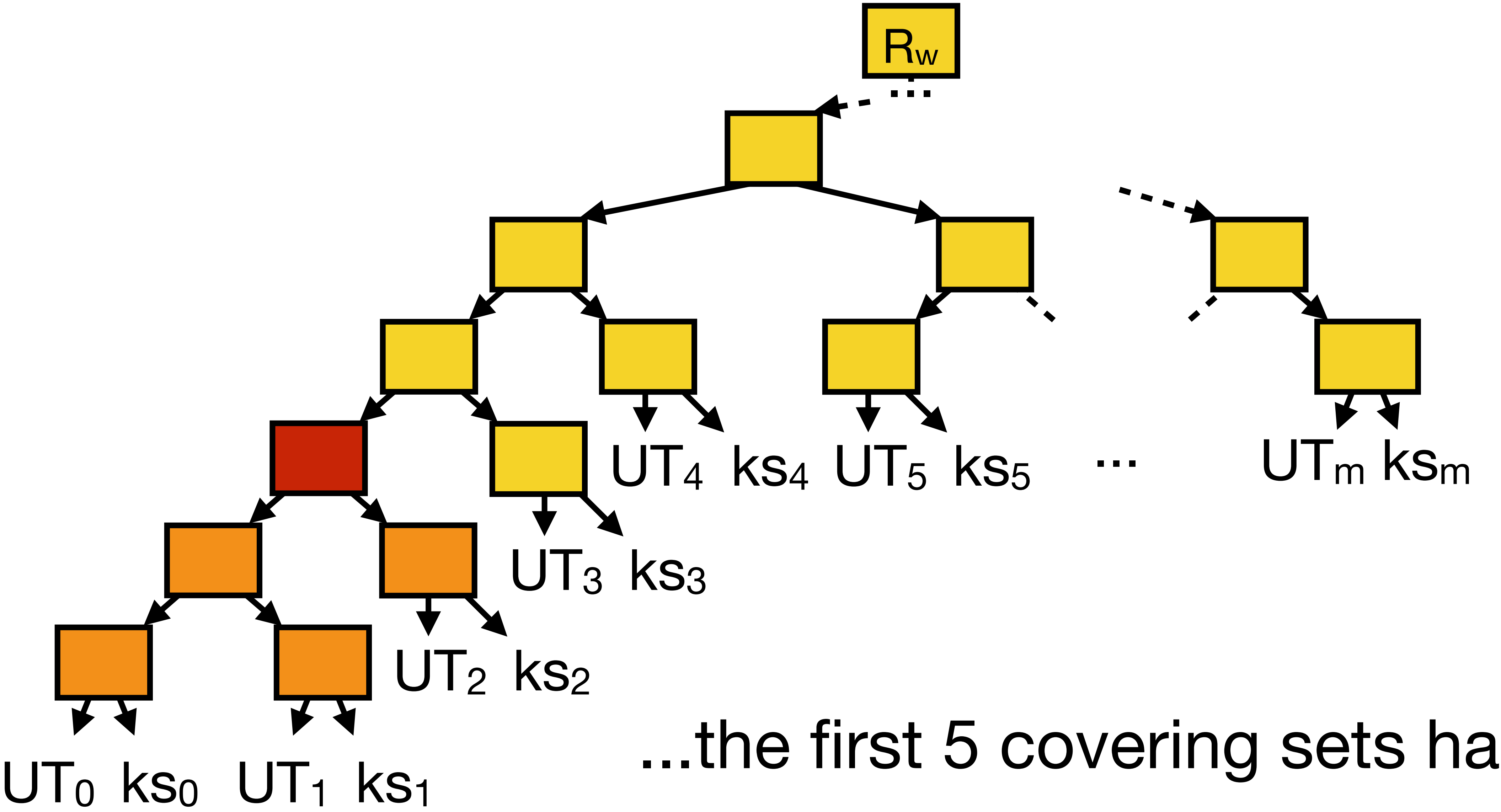


...the first 5 covering sets have size 1.

Tweaking the Tree

The tree does not have to be balanced.

- e.g. if most keywords have ≤ 5 matches:

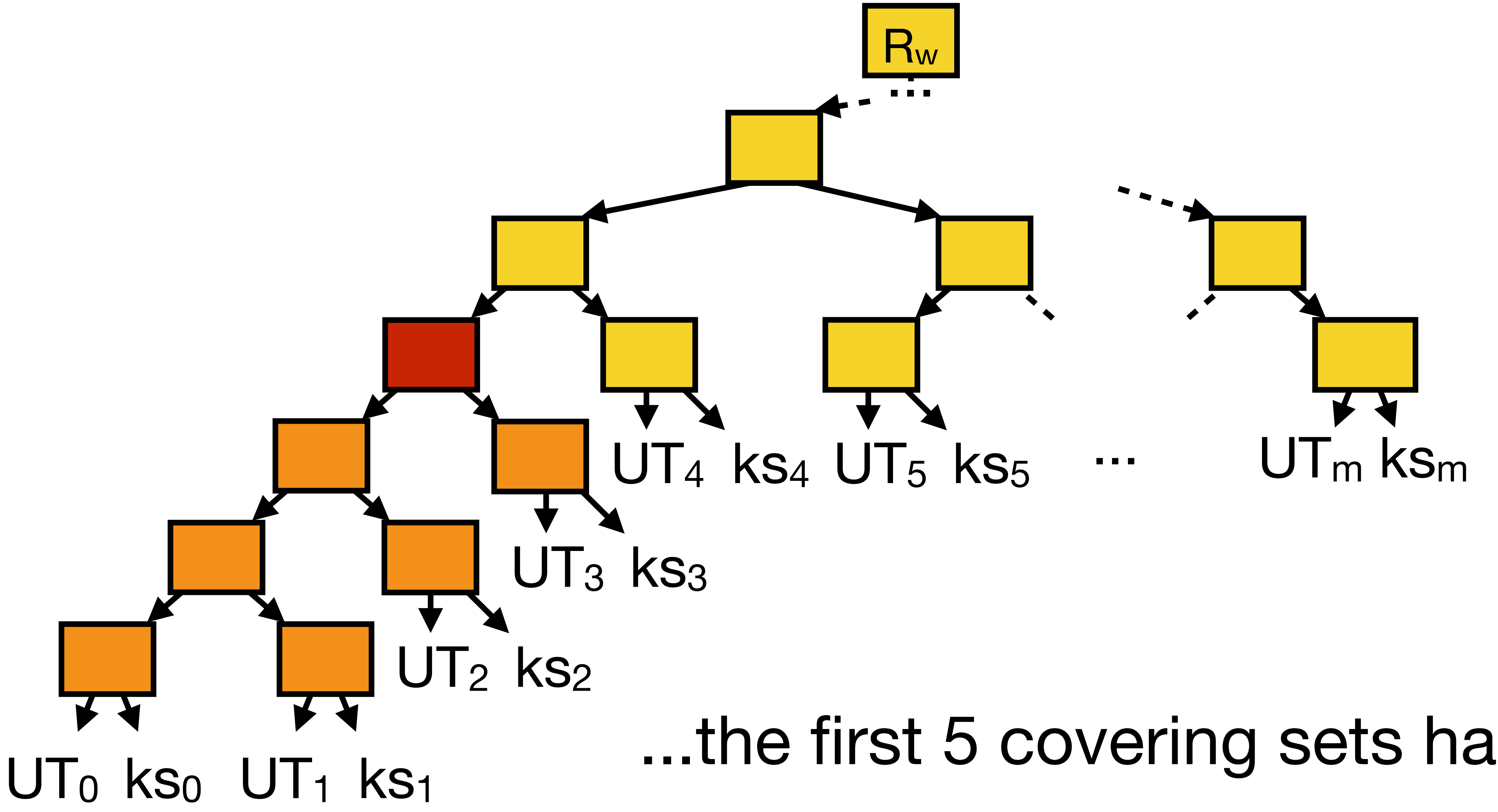


...the first 5 covering sets have size 1.

Tweaking the Tree

The tree does not have to be balanced.

- e.g. if most keywords have ≤ 5 matches:

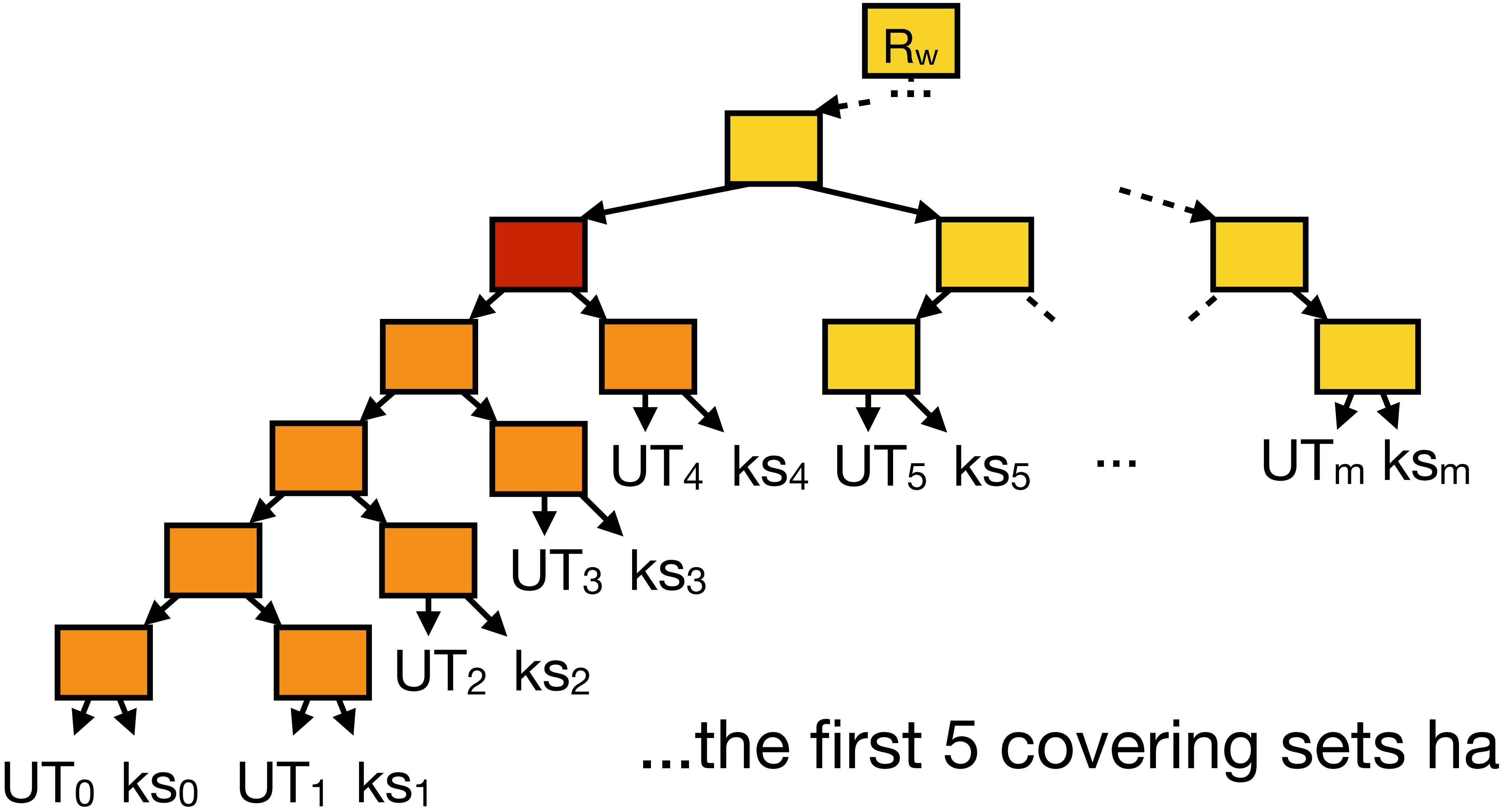


...the first 5 covering sets have size 1.

Tweaking the Tree

The tree does not have to be balanced.

- e.g. if most keywords have ≤ 5 matches:

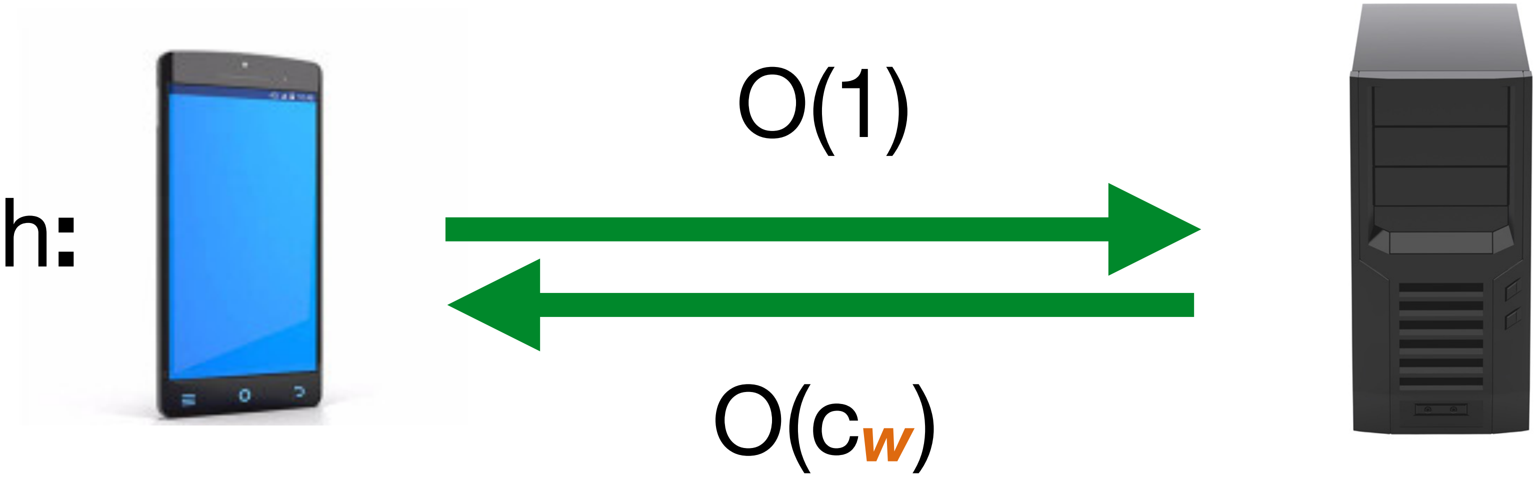


...the first 5 covering sets have size 1.

The tree also does not have to be finite (no last leaf).

Communication Complexity

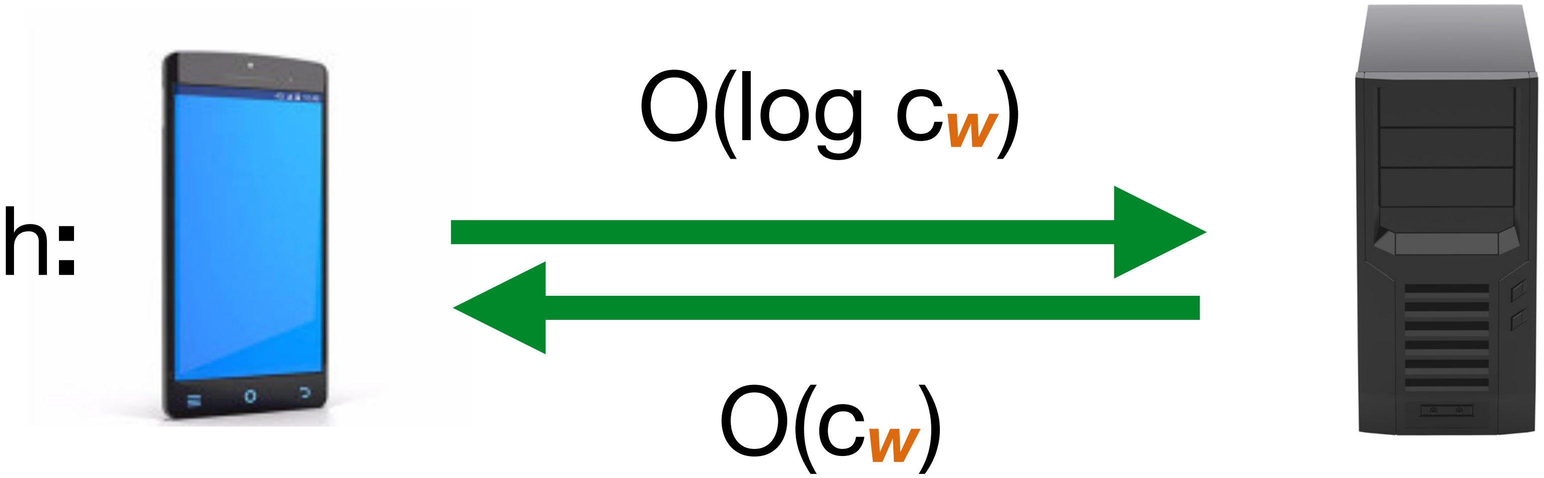
Sophos Search:



$O(1)$

$O(c_w)$

Diane Search:



$O(\log c_w)$

$O(c_w)$

However...

$O(1)$ for Sophos is 2000+ bits (RSA).

$O(\log c_w)$ for Diane is $128 \log c_w$ bits.

Computational Complexity

	Computation		Communication		Client Storage	FS
	Update	Search	Update	Search		
Sophos	$\mathcal{O}(1)$	$\mathcal{O}(c_w)$	$\mathcal{O}(1)$	$\mathcal{O}(c_w)$	$\mathcal{O}(W\log(D))$	✓
Diane	$\mathcal{O}(1)$	$\mathcal{O}(c_w)$	$\mathcal{O}(1)$	$\mathcal{O}(c_w)$	$\mathcal{O}(W\log(D))$	✓

Asymptotically equivalent to Sophos.
Practically much faster: removes RSA bottleneck.

Overall, "crypto" overhead is negligible: IO and memory accesses dominate.

Security model

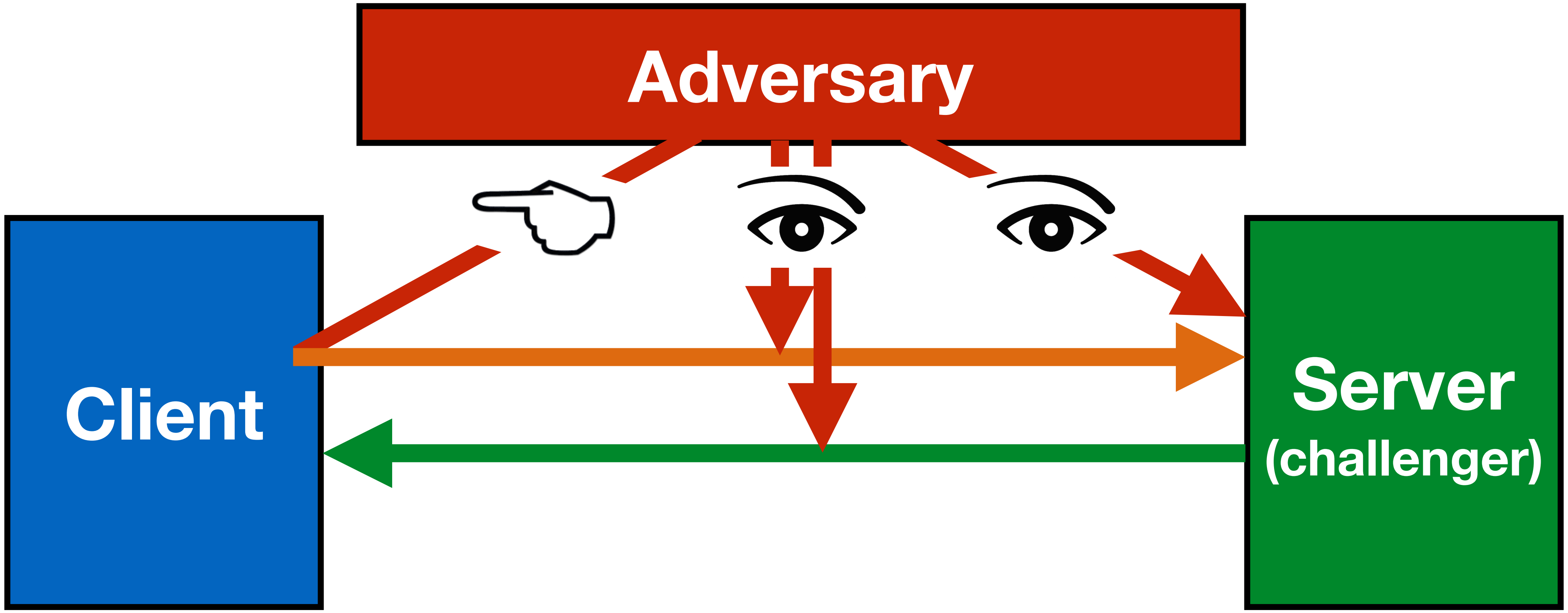
Security is parametrized by a leakage function.

Search(*w*) leaks $\mathcal{L}^{\text{Search}}(w)$.

Update(*w*,*i*) leaks $\mathcal{L}^{\text{Update}}(w,i)$.

Intuition: the adversary should learn no more than this leakage.

Simulation-based security

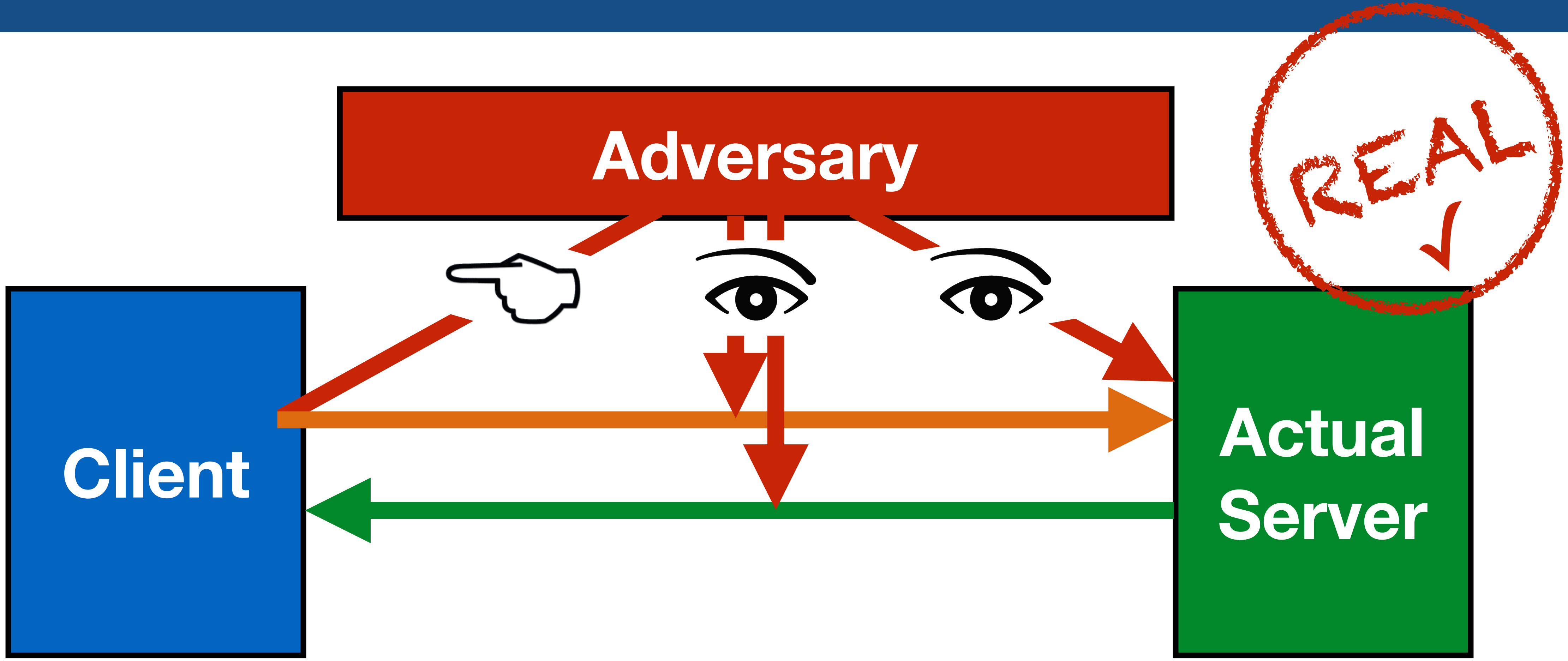


The adversary can:

- adaptively trigger **Search**(*w*) and **Update**(*w*,*i*) queries.
- observe all traffic and server storage.

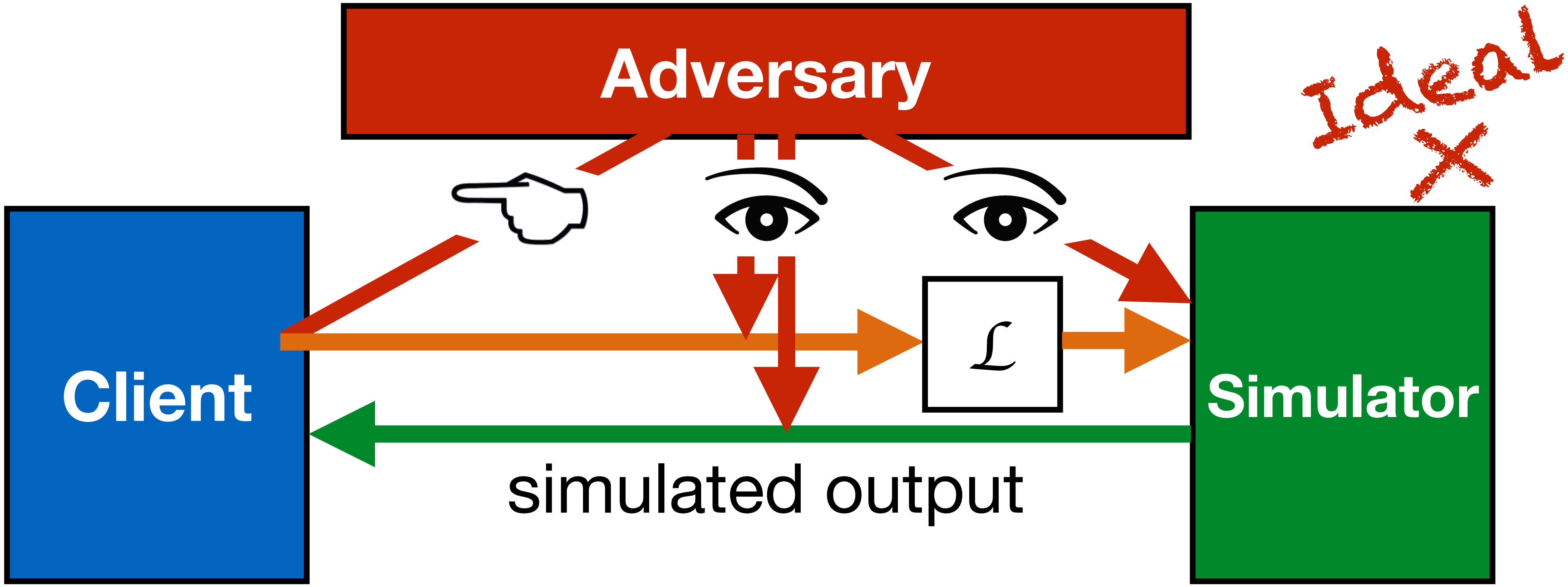
The adversary attempts to distinguish a real and ideal world.

Simulation-based security



In the **real** world, the server receives the actual queries and implements the actual scheme.

Simulation-based security



In the **ideal** world, the server receives only the **leakage** of queries and attempts to mimick a real server.

$\mathcal{L}$-security: there exists a simulator s.t. no adversary can distinguish the two worlds with significant probability.

Random oracle

Assume the adversary triggers:

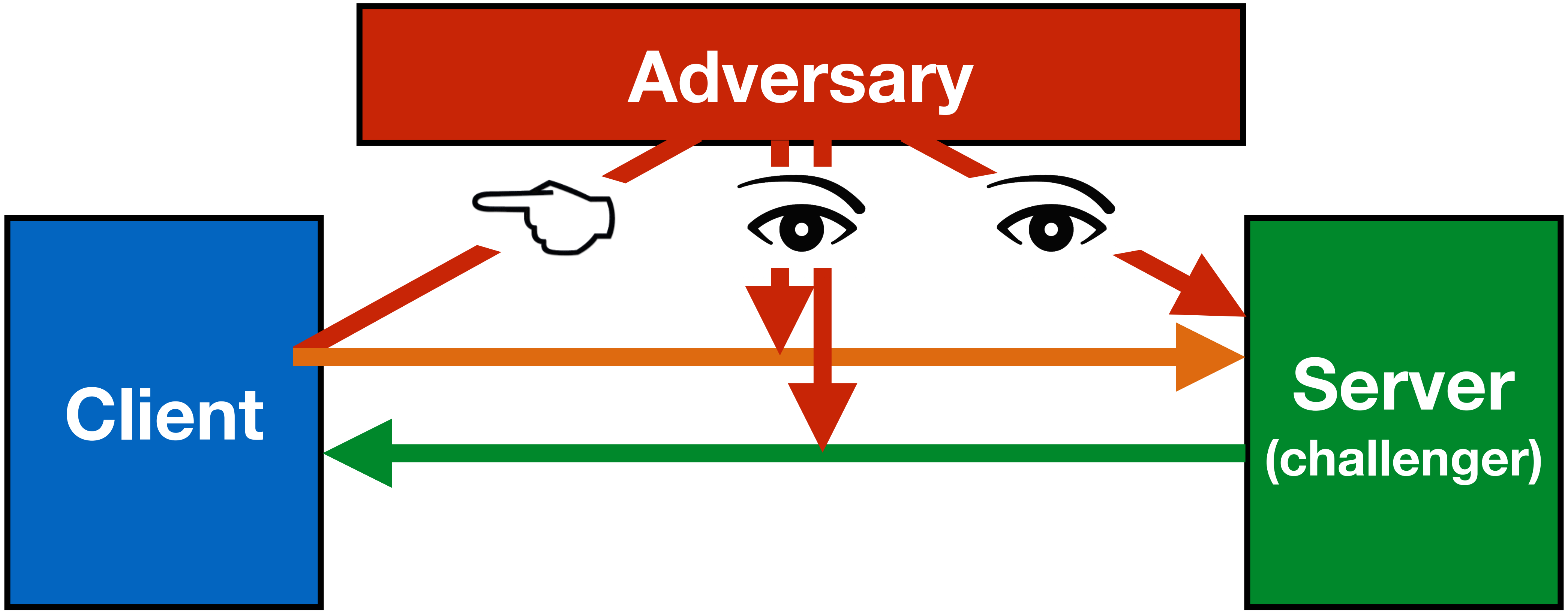
Update(w_0,0)
Update(w_1,1)
Update(w',2)
Search(w')

Depending on $w' = w_0$ or $w' = w_1$, different tree, UT's for w' will have to be in a tree with either w_0 or w_1.

...but the simulator has to commit before knowing.

▹ ROM required.

Indistinguishability security



Adversary

Client

Server

(challenger)

The adversary (adaptively) triggers **pairs** of queries.

World 0	**World 1**
Query(0)	Query(0)'
Query(1)	Query(1)'
...	...

Same leakage

The challenger chooses ***b*** and runs **World *b***.

Security of Diane

In the end:

- Diane is provable in the simulation setting using ROM.
- It is also provable in the indistinguishability setting without ROM (with worse bounds).

Malicious Adversaries

The server could lie when answering **Search** queries.

Generic solution:

For each keyword, the client stores and updates a *set hash* of matching documents.

Example of set hash: XOR of hashes of indices.

- **Update**(w,i): $h_w \leftarrow h_w \oplus H(i)$. Initially $h_w = 0$.
- **Search**(w): upon receiving i_0, ..., i_c, check $h_w = \sum H(i_k)$.

Allowing Deletions

Generic solution:

For **Update** queries, let op = *add* or *del.*
Send (UT_c, enc(***i*** || op)) instead of (UT_c, enc(***i***)).

During a **Search** query, the server retrieves op and can cancel out *add*'s and *del*'s.

Reducing Client Storage

Diane uses 1 round-trip for **Search** queries and W log(D) client storage.

If we allow 2 round-trips:

- **honest-but-curious** setting: O(1) storage is easy (outsource the c_w's).
- **malicious** setting: trade-offs are possible using Merkle trees.

  α W log(D) storage at the cost of log($1/\alpha$) extra communication.

Locality

Diane's crypto is almost free w.r.t. computation and communication.

Hidden cost: non-locality.

- ▻ In an **unencrypted** database: DB(***w***) would be stored contiguously.
- ▻ In **SE** schemes it is spread across |DB(***w***)| random locations.

This is cost is (mostly) inherent [CT14].

Summary of Diane

- Provable forward-privacy.
- Simple.
- Efficient search (IO bounded).

  Asymptotically non-optimal outgoing communication (but very good in practice).
- Efficient update.

Open problems: mitigating inherent issues.

- ▹ Leakage-abuse attacks.
- ▹ Non-locality.

Thank you!