CS 599: Formal Methods in Security and Privacy
Proofs of Protocol Security in Real/Ideal Paradigm

Marco Gaboardi
gaboardi@bu.edu

Alley Stoughton
stough@bu.edu
Real/Ideal Paradigm

• We proved IND-CPA (indistinguishability under chosen plaintext attack) security of a symmetric encryption scheme built from a pseudorandom function plus randomness.

• Now, we’re going to consider a proof in the Real/Ideal Paradigm of the security of a three party cryptographic protocol.

• In the real/ideal paradigm there are two games:
  • A “real” game based on how the actual protocol works.
  • An “ideal” game that is secure by construction.

• In the security proof we show that an Adversary can’t distinguish the two games—or can only distinguish them with negligible probability.
Private Count Retrieval Protocol

- The Private Count Retrieval (PCR) Protocol involves three parties:
  - a **Server**, which holds a database
  - a **Client**, which makes queries about the database
  - an *untrusted Third Party (TP)*, which mediates between the Server and Client

- A **database** is one-dimensional: it consists of a list of **elements**

- Each **query** is also an element, and is a request for the count of the number of times it occurs in the database
Private Count Retrieval Protocol

- For example, suppose the database is $[0; 2; 0; 4; 2]$.
- If the query is $0$, the answer is:
  - $2$
- If the query is $4$, the answer is: $1$
- If the query is $3$, the answer is:
  - $0$
Security Goals for PCR

• Informally, the goal is for:
  • Client to only learn the counts for its queries, not anything else about the database (we’ll limit how many queries it can make)
  • Server to learn nothing about the queries made by the Client other than the number of queries that were made
  • TP to learn nothing about the database and queries other than certain element patterns
Hashing

- The PCR protocol makes use of \textit{hashing}, a process transforming a value of some type into a bit string of a fixed length
  - When distinct inputs are hashed, it should be very unlikely that the resulting bit strings are equal
  - Given a bit string, it should be hard to find an input that hashes to it
- In an implementation, we might use a member of the SHA family of hash functions
- But in our proofs, we’ll model hashing via a \textit{random oracle}
  - Like the true random function of the IND-CPA example, but directly accessible to the adversary.
PCR Protocol Operation

Environment

Server

TP

Client

Main

sec

tag

count

qry

count

db

res

res

db

db
PCR Protocol Operation

![Diagram showing the PCR Protocol Operation]

- **Server**
  - db
  - hdb

- **TP**
  - sec
  - tag
  - count

- **Client**
  - qry
  - count

- **Main**
  - db
  - res
  - res
PCR Protocol Operation

Environment

Server

TP

Client

Main

hdb

sec

tag

qry

count

count

db

db

res

res

secrets are bit strings of length sec_len
PCR Protocol Operation

Environment

Server
  hdb
  db

TP
  sec
  tag
  count

Client
  qry
  count

Main
  res
  res
  db
PCR Protocol Operation

random shuffle

Server

hdb

db

TP

sec

tag

count

Client

qry

count

Main

db

res

res
PCR Protocol Operation

tags are bit strings of length $\text{tag\_len}$

**Diagram:**

- **Server**
  - hash
  - elem/sec
  - pairs
  - hdb
- **TP**
  - sec
  - tag
  - count
- **Client**
  - qry
  - count
- **Main**
  - db
  - res
  - res
PCR Protocol Operation

Environment

Server → TP → Client

- hdb → TP
- tag → count
- sec
- qry → count
- db → Server
- res → Main
- res → Main
PCR Protocol Operation

![Diagram of PCR Protocol Operation]

- Server
- TP
- Client
- Environment
- Main
- db
- res
- hdb
- tag
- count
- sec
- qry
- count
PCR Protocol Operation

Server -> TP
  hdb

TP -> Client
  tag
  count

Client -> Server
  qry/sec
  hash

Main
  db
  res
  res

Environment
  sec
PCR Protocol Operation

Environment

Server ➔ TP

TP ➔ Client

Client ➔ qry

Main

Server ➔ db

TP ➔ count, tag

Client ➔ count

Server ➔ hdb
PCR Protocol Operation

![Diagram of PCR Protocol Operation](image)

- **Server**
  - `hdb`
  - `db`

- **Main**
  - `db`
  - `res`

- **Client**
  - `qry`
  - `count`

- **Environment**
  - `sec`
  - `tag`
  - `count`
  - `qry`
PCR Protocol Operation

Environment

Server -> TP

TP -> Client

Main

Server -> db

TP -> hdb

Client -> count

Client -> qry

Client -> final result

Server -> db

TP -> res

Client -> res
Protocol Example

- E.g., suppose the original database was \([0; 1; 1; 2]\) and the queries are 1, 2 and 3
- The Server’s shuffled database might be \([1; 0; 2; 1]\)
- TP will get a hashed database \([t_2; t_1; t_3; t_2]\) and hash tags \(t_2, t_3\) and \(t_4\), and so will return to Client counts 2, 1 and 0 (assuming no hash collisions)
EasyCrypt Code

• On GitHub you can find:
  • All the EasyCrypt definitions and proofs
  • A link to a conference paper about PCR and its proofs
    • Joint work with Mayank Varia

https://github.com/alleystoughton/PCR
Elements, Secrets and Hashing in EasyCrypt

- Elements (type `elem`) may be anything
- Secrets (type `sec`) are bits strings of length `sec_len`
- Hash tags (type `tag`) are bit strings of length `tag_len`
- Hashing is done using a random oracle in which element/secret pairs are hashed to hash tags
Random Oracle Theory

module type OR = {
    proc init() : unit
    proc hash(inp : input) : output
}.

module Or : OR = {
    var mp : (input, output) fmap

    proc init() : unit = {
        mp <- empty;
    }

    proc hash(inp : input) : output = {
        if (! dom mp inp) {
            mp.[inp] <$> output_distr;
        }
        return oget mp.[inp];
    }
}.
clone RandomOracle as RO with
  type input <- elem * sec,
  op input_default <- (elem_default, zeros_sec),
  op output_len <- tag_len,
  type output <- tag,
  op output_default <- zeros_tag,
  op output_distr <- tag_distr
proof *.
(* realization *) ... (* end *)

Thus R0.0r with module type R0.OR is the random oracle
Random Shuffling

module Shuffle = {
    proc shuffle(xs : elem list) : elem list = {
        var ys : elem list; var i : int;
        ys <- [];
        while (0 < size xs) {
            i <$> [0 .. size xs - 1];
            ys <- ys ++ [nth elem_default xs i];
            xs <- trim xs i;
        }
        return ys;
    }
}. 

each of the (size xs)! reorderings of xs are equally possible (because of duplicates, some of these reorderings may be the same)
PCR Protocol

type db = elem list. type hdb = tag list.

... type server_view = server_view_elem list. type tp_view = tp_view_elem list. type client_view = client_view_elem list.

module type ENV = {
  proc init_and_get_db() : db option
  proc get_qry() : elem option
  proc put_qry_count(cnt : int) : unit
  proc final() : bool
}.

Each party has a view variable that records everything it sees
module Protocol (Env : ENV) = {
    module Or = RO.Or
    ...

    proc main() : bool = {
        var db_opt : db option; var b : bool;
        init_views(); Or.init();
        server_gen_sec(); client_get_sec();
        db_opt @ Env.init_and_get_db();
        if (db_opt <> None) {
            server_hash_db(oget db_opt);
            tp_get_hdb();
            client_loop();
        }
        b @ Env.final();
        return b;
    }
}.
PCR Protocol

proc client_loop() : unit = {
    var cnt : int; var tag : tag;
    var qry_opt : elem option;
    var not_done : bool <- true;
    while (not_done) {
        qry_opt <-@ Env.get_qry();
        cv <- cv ++ [cv_got_qry qry_opt];
        if (qry_opt = None) {
            not_done <- false;
        } else {
            tag <-@ Or.hash((oget qry_opt, client_sec));
            cnt <-@ tp_count_tag(tag);
            cv <- cv ++
                 [cv_query_count(oget qry_opt, tag, cnt)];
            Env.put_qry_count(cnt);
        }
    }
}
Adversarial Model

• We are modeling what is called *semi-honest* or *honest-but curious* security.

• In this model, the Adversary is given access to a given protocol party’s *view*—the party’s data—but it is not allowed to modify that data.

• The Adversary is also given access to the *hash* procedure of the random oracle — this is different from having access to its map.

• The Real and Ideal games for each protocol party are parameterized by the Adversary.
  
  • The Adversary tries to learn more from the protocol’s view plus the *hash* procedure’s view of the random oracle than it *should*.

• At the end of the games, the Adversary returns a boolean judgement, trying to make the probability it returns *true* be as different as possible in the Real and Ideal games.
Real Games

- The Real Games for the Server, Third Party and Client are formed as specializations of Protocol.
- For a given party, we define the module type ADV of Adversaries for that party.
  - In calls to the Adversary, the party’s current view is supplied.
- The Real Game $G_{\text{Real}}$ is parameterized by $\text{Adv} : \text{ADV}$.
  - defined by giving Protocol an environment Env made out of Adv.
Example: Adversary for Server

module type ADV(O : RO.OR) = {
  proc init_and_get_db(view : server_view) : 
    db option {O.hash}
  proc get_qry(view : server_view) : elem option {O.hash}
  proc qry_done(view : server_view) : unit {O.hash}
  proc final(view : server_view) : bool {O.hash}
}.

• Adversary can do hashing when deciding which database and queries to choose

• Queries are chosen one by one — adaptively

• \texttt{qry\_done} is called with server view, which does not include the count for the query

• Each time the Adversary is called, it can do hashing to try to increase its knowledge
Example: Real Game for Server

module GReal(Adv : ADV) = {
  module Or = RO.Or
  module A  = Adv(Or)

  module Env : ENV = {
    proc init_and_get_db() : db option = {
      var db_opt : db option;
      db_opt <= A.init_and_get_db(Protocol.sv);
      return db_opt;
    }

    proc get_qry() : elem option = {
      var qry_opt : elem option;
      qry_opt <= A.get_qry(Protocol.sv);
      return qry_opt;
    }

    proc put_qry_count(cnt : int): unit = {
      A.qry_done(Protocol.sv);
    }
  }
}
Real Game for Server

proc final() : bool = {
    var b : bool;
    b <$> A.final(Protocol.sv);
    return b;
}

proc main() : bool = {
    var b : bool;
    b <$> Protocol(Env).main();
    return b;
}. 
Ideal Games

• A party’s Ideal Game is also parameterized by a Simulator (in addition to the Adversary)

• Simulator’s job is to convince the Adversary it’s interacting with the real game: it must simulate the party’s view and the hashing function’s view of the random oracle state

• Because we are working information-theoretically, when assessing the information leakage from the Ideal Game to the Simulator (and thus Adversary), we don’t have to scrutinize its Simulator
  
  • It can’t learn more about the database or queries by brute force computation

• In fact, in our EasyCrypt security theorems, the Simulators are existentially quantified
Two Dimensional Sequences of Games

• When proving security against a protocol party, we use EasyCrypt’s pRHL and ambient logics to connect the party's Real and Ideal Games via a sequence of games.

• Upper bound on distance between source and target games is sum of intermediate transitions’ upper bounds.

• We can prove a game transition using a previously proved sequence of games.

![Diagram showing the sequence of games](https://via.placeholder.com/150)

\[ G_{\text{Real}} \xrightarrow{H_1} G_1 \xrightarrow{H_2} G_2 \xrightarrow{H_3} G_{\text{Ideal}} \]

\[ H_1 \quad H_2 \quad H_3 \]

cryptographic reduction
Reminder: Real Game for Server

Environment discards count before calling Adversary
Real Game for Server

• What (if anything) can the Server learn about the queries and their counts?

• We formalize this by asking what can be learned from the Server views that are passed to the Adversary — plus the ability to run the hash procedure of the random oracle

  • We can think that each time the Adversary is called, the Server is woken up

• To answer and prove this, we need to formalize an Ideal Game
Ideal Game for Server

- Simulator
  - Generates secret, does shuffling, hashing
- Adversary
- Client
  - No hashing

Main

- db
- res

?r
- done
Ideal Game for Server

• The Simulator doesn’t directly learn anything about the queries, and so the Server views it simulates can’t convey anything about them either

• And the query loop doesn’t modify the random oracle, so experimentation with the random oracle won’t learn anything either

• But because the Server is woken up each iteration of the query loop, the Server does learn the number of queries
Proof of Security Against Server

• We are able to prove perfect security: Real/Ideal games equally likely to return true:

\[
\text{lemma GReal}_{-}G\text{Ideal} : \\
\begin{align*}
\exists (\text{Sim} &: \text{SIM}\{-G\text{Real}, -G\text{Ideal}\}), \\
\forall (\text{Adv} &: \text{ADV}\{-G\text{Real}, -G\text{Ideal}, -\text{Sim}\}) \& m, \\
\Pr[\text{GReal}(\text{Adv}).\text{main}\left(\right) @ & m : \text{res}] = \\
\Pr[\text{GIdeal}(\text{Adv}, \text{Sim}).\text{main}\left(\right) @ & m : \text{res}].
\end{align*}
\]

• The only challenge is dealing with the redundant hashing performed by the Client in the Real but not the Ideal Game

• We remove it using a variation of a technique due to Benjamin Grégoire
Redundant Hashing

module type HASHING = {
  proc hash(inp : input) : output
  proc rhash(inp : input) : unit
}.

module type HASHING_ADV(H : HASHING) = {
  proc main() : bool {H.hash H.rhash}
}.

Two implementations of HASHING, both built from a random oracle 0:

- **NonOptHashing** ("non optimized hashing"), in which rhash hashes its input, but discards the result
- **OptHashing** ("optimized hashing"), where rhash does nothing
module GNonOptHashing(HashAdv : HASHING_ADV) = {
    module H = NonOptHashing(Or)
    module HA = HashAdv(H)
    proc main() : bool = {
        var b : bool;
        Or.init(); b <$> HA.main();
        return b;
    }
}.

module GOptHashing(HashAdv : HASHING_ADV) = {
    module H = OptHashing(Or)
    module HA = HashAdv(H)
    proc main() : bool = {
        var b : bool;
        Or.init(); b <$> HA.main();
        return b;
    }
}.
Redundant Hashing

lemma GNonOptHashing_GOptHashing
  (HashAdv <: HASHING_ADV{Or}) &m :
  Pr[GNonOptHashing(HashAdv).main() @ &m : res] =
  Pr[GOptHashing(HashAdv).main() @ &m : res].

• Proof intuition: redundant hashing can be put off until it’s superseded by hash or no longer necessary

• Proof uses EasyCrypt’s eager tactics

• To use in Server proof, we define a concrete adversary HashAdv in such a way that the left side of the gap in the sequence of games proof can be connected with GNonOptHashing(HashAdv), and GOptHashing(HashAdv) can be connected with the right side of the gap
Ideal Game for Third Party

Server/Client hash **elems** in private random oracle
Ideal Game for Third Party

• The Adversary is invoked with the TP’s view when the database and queries are requested by the game and client loop

• In the Ideal Game, Adversary only learns patterns, not anything more about the database and queries
  • It doesn’t have access to the private random oracle used by Server/Client
  • So even though the database and queries were used to derive the hashed database \([t_1; \ldots; t_n]\) and query tags \(s_1, \ldots, s_m\), these tags were all randomly (but consistently) chosen, and so convey no information about the particular elements
  • And the Server’s random shuffling means it doesn’t learn anything about the order of the database
Security Against Third Party

• E.g., suppose the original database was $[0; 1; 1; 2]$

• The Server’s shuffled database might be $[1; 0; 2; 1]$

• In the Real Game, TP will get a hashed database $[t_2, t_1, t_3, t_2]$, where $t_1 = \text{hash}(0, \text{sec})$, $t_2 = \text{hash}(1, \text{sec})$ and $t_3 = \text{hash}(2, \text{sec})$ — for the shared Server/Client $\text{sec}$

• In the Ideal Game, TP will get a hashed database with the same pattern, $[s_2; s_1; s_3; s_2]$, but where the $s_i$ have no connection with hash or $\text{sec}$

• In order to tell the games apart, we can prove it has to guess $\text{sec}$, i.e., call hash with a pair whose second component is $\text{sec}$
To try to differentiate the games, the Adversary can pick a database with a large number of distinct elements, where each element appears a different number of times (e.g., \([0; 1; 1; 2; 2; 2; \ldots]\)).

When given (in TP’s view) the hashed database that was created in the Real or Ideal Game from shuffling the database and then hashing its elements (either paired with \(\text{sec}\) in the random oracle, or in the private random oracle), it can (assuming no hash collisions) match the resulting tags \(t\) with their elements \(e\).

Given a particular \((e, t)\) pair, it can search for a \(\text{sec}'\) such that hashing \((e, \text{sec}')\) results in \(t\). When it finds one, it can check that the rest of the hashed database is consistent with \(\text{sec}'\). Otherwise it can try another choice of \(\text{sec}'\).
Security Against Third Party

- This process is guaranteed to succeed in the Real Game, it’s highly unlikely to succeed in the Ideal Game

- In any event, if the Adversary never calls the random oracle with a pair whose second component is \texttt{sec}, we can prove it will fail to distinguish the Real and Ideal Games
Proof of Security Against Third Party

- To obtain a security theorem, we must limit (limit) the number of distinct inputs the Adversary may hash

- The Server and Client are unrestricted

- We use a cryptographic reduction to bridge the Real and Ideal Games — one proved with up-to-bad reasoning, and so — that makes us assume the Adversary’s procedures are lossless (always terminating), and prove that the Client Loop always terminates

- When we form $G_{\text{Real}}$ and $G_{\text{Ideal}}$, we terminate the Client Loop after $\text{qrys}_\text{max}$ steps (in $G_{\text{Real}}$, by returning None from the environment’s get_qry procedure)
Proof of Security Against Third Party

• Here is the relevant part of the Environment for GReal:

```ocaml
module Env : ENV = {
  var qrys_ctr : int ...
  proc get_qry() : elem option = {
    var qry_opt : elem option;
    qry_opt <@ A.get_qry(Protocol.tpv);
    if (qry_opt <> None) {
      if (qrys_ctr < qrys_max) { qrys_ctr <- qrys_ctr + 1; }
      else { qry_opt <- None; }
    }
    return qry_opt;
  }
}
```
Third Party Proof

• We reduce security against TP to the security of a new abstraction, “Secrecy Random Oracles”

• They offer *limited* (limit) hashing of element/secret pairs (what Adversary does), as well as *unlimited* hashing of elements (what Server and Client do)

• “Dependent” implementation with single map, where hashing an element is same as hashing pair of it and sec — connection with Real Game

• “Independent” implementation with separate maps — connection with Ideal Game

• We prove that a Secrecy Adversary can only tell the games involving the two implementations apart if it does limited hashing of a pair whose second component is sec
Third Party Proof

• The Secrecy Random Oracles proof is carried out using up-to-bad reasoning

• As long as the Secrecy Adversary doesn’t do limited hashing with a pair with right side sec (the “bad” event), we can maintain an invariant:

  • keeping the non-sec-part of the map of the dependent implementation in sync with the non-sec-part of the elem sec map of the independent implementation; and

  • keeping the sec-part of the map of the dependent implementation in sync with the elem map of the independent implementation
Third Party Proof

• We reduce the upper-bounding of the probability of the bad event holding to a lemma about another new abstraction, “Secret Guessing Oracles”

• It gives the adversary limited ($\text{limit}$) number of chances to guess $\text{sec}$

• EasyCrypt’s pHL is used to upper bound the probability of the adversary winning by

\[
\frac{\text{limit}}{2^{\text{sec\_len}}}
\]

• Both the Secrecy Random Oracles and Secret Guessing Oracles definitions and proofs are packaged up into reusable theories
Third Party Proof

• The theorem for security against the TP upper-bounds the distance between the probabilities of the Real and Ideal Games returning \texttt{true} by

\[
\text{limit} / 2^{\text{sec\_len}}
\]
Reminder: Real Game for Client

Environment discards count before calling Adversary
**Ideal Game for Client**

**SIG** = Simulator’s Interface to Game

* no shuffling or hashing uses elems counts map

---

Adversary generates secret, does hashing

---

**Main**

- **SIG**
  - qry
  - done
  - db

- **Simulator**
  - qry/count
  - done
  - db

- **res**
Proof of Security Against Client

• The Adversary can distinguish the Real and Ideal Games by causing or forcing a hash collision

• If it can find distinct \(\text{elem}\) and \(\text{elem}'\) such that \((\text{elem}, \text{sec})\) and \((\text{elem}', \text{sec})\) hash to the same hash tag, \(\text{tag}\), then it can let \(\text{db} = [\text{elem}]\) and the only query be \(\text{elem}'\)

• In Real Game, count will be
  
  • 1

• In Ideal Game, count will be
  
  • 0

• It can let \(\text{db}\) be a list of distinct elements of greater length than number of distinct hash tags, and work through that same list of elements as queries
Proof of Security Against Client

• Thus we must impose a hashing budget on the Adversary — not just on the hashing it does directly, but also on the hashing it makes Server and Client do:

  • **adv_budget** — distinct hashing done by Adversary
  • **db_uniqs_max** — maximum number of distinct elements in database
  • **qrys_max** — maximum number of queries
  • **budget = adv_budget + db_uniqs_max + qrys_max**
  • If Adversary doesn’t respect budget, we terminate game early (we terminate the Client Loop after **qrys_max** steps)
  • Because the proof uses up-to-bad reasoning, we need that Adversary is always terminating and Client Loop terminates
Proof of Security Against Client

- We have **Budgeted Random Oracles**, which provide:
  - *separate* budgeted hashing functions for the Adversary, Server and Client
  - set a flag when over budget, but keep working
  - for Adversary and Server, only distinct inputs matter, but for Client its the number of hashes
  - ordinary (unrestricted) hashing (which the Adversary uses before making its final judgement)

- There are two implementations of budgeted random oracles:
  - a "collision-possible" one in which hash collisions may occur
  - a "collision-free-while-within-budget" one in which hash collisions don’t happen if only budgeted hashing is done and the individual budgets are respected
Proof of Security Against Client

- Each move back and forth between the collision-possible and collision-free-while-within-budget versions incurs a penalty of
  \[
  \frac{(\text{budget} \times (\text{budget} - 1))}{2^{\text{tag_len}} + 1}
  \]
- This is proved using up-to-bad reasoning, where the “bad” event is when a collision occurs
- EasyCrypt’s failure event lemma and pHL are used to bound the probability that failure occurs
- The proof is packaged into a reusable theory
Client Proof

• Move to collision-possible budgeted random oracle
• Move to collision-free-while-within-budget random oracle
• Use complex relational invariant to switch to Server, TP and Client using an elements counts map instead of hashed database (but Server still does hashing)
• Switch back to collision-possible budgeted random oracle
• Switch back to ordinary random oracle (Adversary still subjected to budget)
• Get rid of Server’s hashing, which is now seen to be redundant
• Show that computing elements counts map works out same without first shuffling database
• Final refactoring
Client Proof

- Theorem for security against the Client upper bounds the distance between the probabilities of the Real and Ideal Games returning true by

\[
(budget \times (budget - 1)) / 2^{\text{tag\_len}}
\]

which is two times

\[
(budget \times (budget - 1)) / 2^{\text{tag\_len} + 1}
\]
Summary/Lessons Learned

• Size of EasyCrypt formalization:
  
  • About **380 lines** of theorem statements and relevant definitions (random oracles, protocol definition, etc.)
  
  • About **5,275 lines** of proof (which one can trust EasyCrypt to check)

• Two-dimensional game structure very useful

• Formalizing Protocol once — parameterized by Environment — and then specializing to Real Games works well

• Because we work information-theoretically, Simulators are existentially quantified (so part of proof, not specification)

• Removing redundant hashing was crucial, and our version of Grégoire’s technique was proved once and used twice
Summary/Lessons Learned

• Use of budgeted random oracles in Client proof let us do the hard step of the proof without worrying about hash collisions.

• EasyCrypt made it easy to obtain concrete upper bounds in terms of game parameters on the distances between real and ideal games.
Discussion

• Q: In the PCR Protocol, does the Client always get correct counts for its queries?
  • A: Not in the highly unlikely event of hash collisions

• Q: Why do we let the Adversary choose the database and queries?
  • A: This models how it may have inside information about what elements (e.g., people's names) are likely to appear in the database or in queries
    • E.g., TP, when analyzing the tags it sees, might guess that a tag appearing numerous times corresponds to “Alice”, based on knowledge of an organization. But it won’t be able to confirm that guess.
Discussion

• Q: Is it realistic to assume two parties can communicate, without the other one eavesdropping?
  
  • A: Yes. The Adversary works on behalf of a given party, and has no special access to the network
Discussion

• Q: Are the restrictions we place on the Adversary realistic?
  • A: Server:
    • No restrictions
  • A: TP:
    • Limit on distinct hashes
  • A: Client:
    • Budget for Adversary’s distinct hashing
    • Budget on number of distinct elements in database
    • Budget on number of queries

in reality, the Adversary doesn’t choose the database or queries
Questions?