Mini-APP: Proving a Simple Private Information Retrieval Protocol Secure in EasyCrypt

Alley Stoughton

First EasyCrypt Summer School
University of Pennsylvania
July 18, 2013

LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

This work is sponsored by the Director of National Intelligence under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author, and are not necessarily endorsed by the United States Government.

Acknowledgements

It's a pleasure to acknowledge helpful discussions with:

- **SPAR Formal Methods Team**

  Jonathan Herzog (MIT Lincoln Laboratory), Aaron D. Jaggard (Naval Research Laboratory), Jonathan Katz (University of Maryland), Catherine Meadows (Naval Research Laboratory), Adam Petcher (MIT Lincoln Laboratory);

- **MIT Lincoln Laboratory SPAR Team**

  Emily Shen, Mayank Varia, Arkady Yerukhimovich;

- **EasyCrypt Team**

  Gilles Barthes, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, Pierre-Yves Strub, Santiago Zanella-Béguelin.

SPAR Formal Methods Project

- **As part of the IARPA SPAR Project, the SPAR Formal Methods Team undertook the verification in EasyCrypt of the APP Protocol developed by the University of California, Irvine.**
- **The UCI APP Protocol is a three party Private Information Retrieval (PIR) protocol.**
- **Its verification required our learning about—or developing—various sophisticated techniques:**
  - **constraining oracle use by adversaries;**
  - **reasoning up to failure;**
  - **working with complex relational invariants;**
  - **loop fission;**
  - **eager/lazy random sampling.**

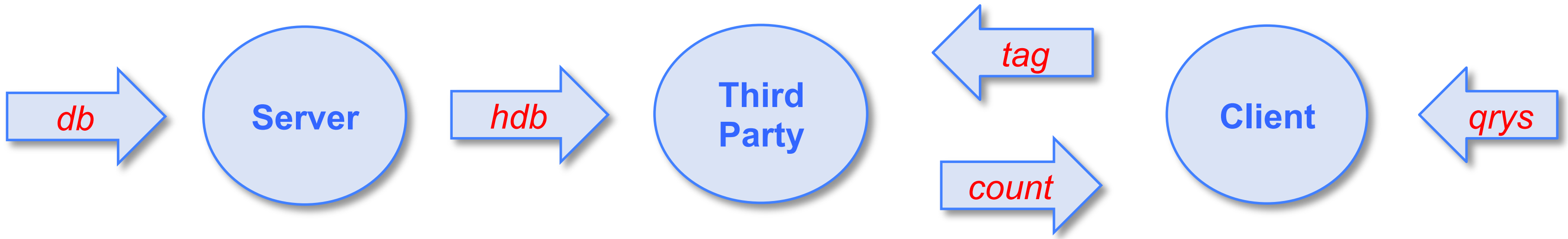
Genesis of Mini-APP

- **Because of the complexity of the UCI APP protocol, it was sub-optimal to be learning/developing these techniques in the context of the full UCI APP proofs.**
- **Consequently, I began working on proving the security of a simpler protocol—which I've named *Mini-APP*—in parallel with my work on the UCI Isolated Box proof.**

Mini-APP Protocol

Mini-APP is a three-party PIR protocol:

- **The Server takes in a database consisting of a list of attributes, hashes each attribute, producing a hashed database, and sends the hashed database to the Third Party (TP).**
- **The Client works its way through a list of queries—also attributes—hashing each query, and asking the TP for the number of occurrences of this hash tag in the hashed database.**



Random Oracle

- **Hash tags are bit strings, all of the same length.**
- **Hashing is done using a random oracle, consisting of a map to which new attribute/tag pairs are added, dynamically.**
- **It is consistent with our axioms that there are many fewer hash tags than attributes.**
- **Because of the possibility of hash collisions, the results computed by the Client may include false positives.**
  - **E.g., if the database consists of attributes** x **and** y**, but** x **and** y **hash to the same hash tag, then the count for query** x **will be** 2 **not** 1**.**

Real and Ideal Games

- We formalize the security of the Client’s view of the protocol using a pair of cryptographic games.
- The “real” game is based on the protocol as described above.
- The “ideal” game is based on a variant of the protocol in which:
  - the query counts are in perfect agreement with the database, and
  - it’s obvious that nothing other than the results is leaked from the database to the client.

Adversary Model

- Our games are parameterized by an adversary with access to the random oracle.
- Both real and ideal games begin with an initial call to the adversary in which the adversary picks a database and list of queries (possibly hashing various attributes in the process).
- Both games end with a final call to the adversary, in which the adversary is called with the Client *view*: a record of what the Client saw during the game. The adversary returns a boolean judgment (possibly hashing various attributes in the process), and this boolean is returned as the result of the game.
- The Client view of the protocol is said to be *secure* iff the adversary can't distinguish the real and ideal games, i.e., the probabilities of the games returning `true` differ by a negligible amount.

Infeasible Adversaries

- Because EasyCrypt
  - allows for computationally infeasible adversaries, and
  - the adversary has access to the oracle at the beginning of the game,

  unless we restrict the adversary, it will be able to distinguish the real and ideal games.
- For example, the initial call to the adversary can hash attributes until it finds a pair (x, y) of attributes hashing to the same hash tag, and then return [x, y] as the database and [x] as the list of queries.
- Alternatively, the initial call could return a database consisting of more distinct attributes than there are hash tags, plus a list of queries equal to the database (thus making the protocol itself cause and exploit a hash collision).

Limiting the Adversary

- Consequently, we
  - limit the number of distinct oracle calls that the adversary may make in its initial call, and
  - limit the sizes of the database and lists of queries that the adversary may return,

  in such a way that
  - the initial adversary call has a negligible chance of forcing a collision, and
  - there is a negligible probability of a collision occurring during the execution of the protocol itself.
- When the adversary doesn't play by the rules, it loses the game.
- But we must prove that the protocol itself keeps within its budget of distinct oracle calls.

EasyCrypt Proof Overview

- It will take us fifteen transitions to get from the real game (`GReal`) to the ideal game (`GIdeal`).
- We'll start by giving some necessary definitions.
- Then, we'll consider `GReal` and `GIdeal` in detail.
- Finally, we'll look at highlights of the intermediate games and transition proofs.
- During the following lab session, you'll explore the case study proof in more detail.

Attributes and Tags

```
type attr.

type attrs_counts = (attr, int)map.

op tagLen : int.

axiom TagLen : tagLen >= 0.

clone Word as Tag with op length = tagLen.

type tag = Tag.word.

op tagDistr : tag distr = Tag.Dword.dword.
```

Oracles

```
type hash_map = (attr, tag)map.

op budget : int.

axiom Budget : 1 <= budget <= 2 ^ tagLen.

op collBound : real = (budget * (budget - 1))%r / (2 ^ tagLen)%r.

module type OR = {
  fun init() : unit
  fun bhash(attr : attr) : tag
  fun hash(attr : attr) : tag
  fun suff(n : int) : bool
}.
```

Standard Oracle

```
module Or : OR = {
  var mp : hash_map
  var ctr : int
  var over : bool

  fun init() : unit = {
    mp = Map.empty;
    ctr = 0;
    over = false;
  }
```

Standard Oracle

```
fun bhash(attr : attr) : tag = {
  if (!(in_dom attr mp)) {
    mp.[attr] = $tagDistr;
    if (ctr < budget) {
      ctr = ctr + 1;
    }
    else {
      over = true;
    }
  }
  return proj mp.[attr];
}
```

Standard Oracle

```
    fun hash(attr : attr) : tag = {
      if (!(in_dom attr mp)) {
        mp.[attr] = $tagDistr;
      }
      return proj mp.[attr];
    }

    fun suff(n : int) : bool = {
      return !over /\ n <= budget - ctr;
    }
  }
```

Types, Adversaries and Games

```
type db = attr list.

type qrys = attr list.

type hdb = tag list.

type view = (attr * tag * int)list.

module type ADV(O : OR) = {
  fun pre() : db * qrys {* O.bhash}
  fun post(view : view) : bool {O.hash}
}.

module type GAME = {
  fun main() : bool
}.
```

Real Game

```
module GReal(Adv : ADV) : GAME = {
  fun serverInit(db : db) : hdb = {
    var i : int;
    var attr : attr;
    var tag : tag;
    var hdb : tag list;
    hdb = [];
    i = 0;
    while (i < length db) {
      attr = proj(nth db i);
      tag = Or.hash(attr);
      hdb = hdb ++ [tag];
      i = i + 1;
    }
    return hdb;
  }
```

Real Game

```
fun tpQueryResponse(tag : tag, hdb : hdb) : int = {
  var i, n : int;
  n = 0;
  i = 0;
  while (i < length hdb) {
    if (proj(nth hdb i) = tag) {
      n = n + 1;
    }
    i = i + 1;
  }
  return n;
}
```

Real Game

```
fun clientQueryLoop(qrys : qrys, hdb : hdb) : view = {
  var view : view;
  var i, n : int;
  var tag : tag;
  var qry : attr;
  view = [];
  i = 0;
  while (i < length qrys) {
    qry = proj(nth qrys i);
    tag = Or.hash(qry);
    n = tpQueryResponse(tag, hdb);
    view = view ++ [(qry, tag, n)];
    i = i + 1;
  }
  return view;
}
```

Real Game

```
fun view(qrys : qrys, db : db) : view = {
  var view : view;
  var hdb : hdb;
  hdb = serverInit(db);
  view = clientQueryLoop(qrys, hdb);
  return view;
}

module A = Adv(Or)

fun main() : bool = {
  var db : db;
  var qrys : qrys;
  var view : view;
  var b, suff : bool;
```

Real Game

```
      Or.init();
      (db, qrys) = A.pre();
      suff = Or.suff(length db  + length qrys);
      if (suff) {
        view = view(qrys, db);
      }
      else {
        view = [];
      }
      b = A.post(view);
      return b;
    }
  }.
```

Ideal Game

```
module GIdeal(Adv : ADV) : GAME = {
  fun computeAnswers(qrys : qrys, db : db) : attrs_counts = {
    var qrysCounts : attrs_counts;
    var i : int;
    var attr : attr;
```

Ideal Game

```
  qrysCounts = Map.empty;
  i = 0;
  while (i < length db) {
    attr = proj(nth db i);
    if (mem attr qrys) {
      if (in_dom attr qrysCounts) {
        qrysCounts.[attr] = proj qrysCounts.[attr] + 1;
      }
      else {
        qrysCounts.[attr] = 1;
      }
    }
    i = i + 1;
  }
  return qrysCounts;
}
```

Ideal Game

```
fun simulator(qrys : qrys, qrysCounts : attrs_counts) : view = {
   var view : view;
   var i, n : int;
   var tag : tag;
   var qry : attr;
```

Ideal Game

```
  view = [];
  i = 0;
  while (i < length qrys) {
    qry = proj(nth qrys i);
    tag = Or.hash(qry);
    if (in_dom qry qrysCounts) {
      n = proj qrysCounts.[qry];
    }
    else {
      n = 0;
    }
    view = view ++ [(qry, tag, n)];
    i = i + 1;
  }
  return view;
}
```

Ideal Game

```
fun view(qrys : qrys, db : db) : view = {
  var view : view;
  var qrysCounts : attrs_counts;
  qrysCounts = computeAnswers(qrys, db);
  view = simulator(qrys, qrysCounts);
  return view;
}

module A = Adv(Or)

fun main() : bool = {
  var db : db;
  var qrys : qrys;
  var view : view;
  var b, suff : bool;
```

Ideal Game

```
    Or.init();
    (db, qrys) = A.pre();
    suff = Or.suff(length db  + length qrys);
    if (suff) {
      view = view(qrys, db);
    }
    else {
      view = [];
    }
    b = A.post(view);
    return b;
  }
}.
```

Proof Goal

- We will prove that distance between the real and ideal games is negligible, i.e., that the absolute value of the difference of the probabilities of the real and idea games returning `true` is negligible.
- Since the view constructed by the simulator in the ideal game only depends on the query counts, and not on any other information derived from the database, this will tell us that the Client view of the protocol is secure.

Injective Oracle

- To transition from the real game—with false positives stemming from hash collisions—to the ideal game—in which attribute counts are accurate, we transition to an intermediate game in which:
  - the functions of the game do budgeted hashing; and
  - the oracle's map stays injective (collision free), as long the game stays within budget.
- Subseqently, we must transition back to using the standard oracle, and to using unbudgeted hashing in the games.
- Each application of our version of the Switching Lemma results in a distance between games bounded by `collBound`:

  `(budget * (budget - 1))%r / (2 ^ tagLen)%r`.
- We assume `collBound` has been tuned to be negligible.

Proof Body

```
section.

declare module Adv : ADV{Or}.

...

lemma GReal_GIdeal &m :
  (forall (O <: OR{Adv}),
   islossless O.bhash => islossless Adv(O).pre) =>
  (forall (O <: OR{Adv}),
   islossless O.hash => islossless Adv(O).post) =>
  `|Pr[GReal(Adv).main() @ &m : res] -
    Pr[GIdeal(Adv).main() @ &m : res]| <= 2%r * collBound.
proof.
...
qed.
```

Proof Body

```
end section.

print GReal_GIdeal.

lemma GReal_GIdeal :
  forall (Adv <: ADV{Or}) &m,
    (forall (O <: OR{Adv}),
     islossless O.bhash => islossless Adv(O).pre) =>
    (forall (O <: OR{Adv}),
     islossless O.hash => islossless Adv(O).post) =>
  `|Pr[GReal(Adv).main() @ &m : res] -
    Pr[GIdeal(Adv).main() @ &m : res]| <= 2%r * collBound.
```

- **Inside the section, our intermediate games aren't parameterized by** `Adv`**—they simply refer to it.**

GReal ➔ G1

- In the transition from GReal to G1, we switch to using budgeted hashing in serverInit and clientQueryLoop, and we inline tpQueryResponse in clientQueryLoop :

```
fun clientQueryLoop(qrys : qrys, hdb : hdb) : view = {
 var view : view;
 var i, j, n : int;
 var tag : tag;
 var qry : attr;
 view = [];
 i = 0;
```

GReal ➔ G1

```
  while (i < length qrys) {
    qry = proj(nth qrys i);
    tag = Or.bhash(qry);
    j = 0;
    n = 0;
    while (j < length hdb) {
      if (proj(nth hdb j) = tag) {
        n = n + 1;
      }
      j = j + 1;
    }
    view = view ++ [(qry, tag, n)];
    i = i + 1;
  }
  return view;
}
```

G1 → G2 → G3 → G4

- **In the next three transitions, we leave our game the same, but change our oracle, moving to an oracle, `OrInj`, whose map will be injective as long as only `bhash` is called and the budget is respected:**
  - `G2`—`OrInst`;
  - `G3`—`OrInj'`;
  - `G4`—`OrInj`.

OrInst

```
local module OrInst : OR = {
  var mp : hash_map
  var ctr : int
  var over : bool
  var range : tag set
  var coll : bool

  fun init() : unit = {
    mp = Map.empty;
    ctr = 0;
    over = false;
    range = FSet.empty;
    coll = false;
  }
```

OrInst

```
fun bhash(attr : attr) : tag = {
   var tag : tag;
   if (!(in_dom attr mp)) {
     if (ctr < budget) {
       tag = $tagDistr;
       ...
       mp.[attr] = tag;
       range = add tag range;
       ctr = ctr + 1;
     }
     else {
       mp.[attr] = $tagDistr;
       over = true;
     }
   }
   return proj mp.[attr];
 }
```

OrInst

```
  fun hash(attr : attr) : tag = {
    if (!(in_dom attr mp)) {
      mp.[attr] = $tagDistr;
    }
    return proj mp.[attr];
  }

  fun suff(n : int) : bool = {
    return !over /\ n <= budget - ctr;
  }
}.
```

OrInst ➔ OrInj’ ➔ OrInj

OrInst

```
if (mem tag range /\ card range <= ctr) {
  coll = true;
}
```

OrInj'

```
if (mem tag range /\ card range <= ctr) {
  coll = true;
  tag = $tagDistr \ range;
}
```

OrInj

```
if (mem tag range) {
  coll = true;
  tag = $tagDistr \ range;
}
```

Collision Lemma

```
module type ADV'(O : OR) = {
  fun main() : bool {* O.bhash O.hash O.suff}
}.

local module Coll(Adv' : ADV') = {
  module A' = Adv'(OrInst)
  fun main() : bool = {
    var b : bool;
    OrInst.init();
    b = A'.main();
    return b;
  }
}.

local lemma Collision (Adv' <: ADV'{OrInst}) &m :
  Pr[Coll(Adv').main() @ &m : OrInst.coll] <= collBound.
```

Switching Lemma

```
local module Switch1(Adv' : ADV') = {
  module A' = Adv'(OrInst)
  fun main() : bool = {
    var b : bool;
    OrInst.init(); b = A'.main(); return b;
  }
}.

local module Switch2(Adv' : ADV') = {
  module A' = Adv'(OrInj')
  fun main() : bool = {
    var b : bool;
    OrInj'.init(); b = A'.main(); return b;
  }
}.
```

Switching Lemma

```
local lemma Switch_main(Adv' <: ADV'{OrInst, OrInj'}) :
  (forall (O <: OR{Adv'}),
   islossless O.bhash => islossless O.hash => islossless O.suff =>
   islossless Adv'(O).main) =>
  equiv
  [Switch1(Adv').main ~ Switch2(Adv').main :
   true ==>
   OrInst.coll{1} = OrInj'.coll{2} /\
   (!OrInst.coll{1} => ={res})].

local lemma Switch (Adv' <: ADV'{OrInst, OrInj'}) &m :
  (forall (O <: OR{Adv'}),
   islossless O.bhash => islossless O.hash => islossless O.suff =>
   islossless Adv'(O).main) =>
  `|Pr[Switch1(Adv').main() @ &m : res] -
    Pr[Switch2(Adv').main() @ &m : res]| <=
    Pr[Switch1(Adv').main() @ &m : OrInst.coll].
```

Instantiation of `Collision` and `Switch`

- **We use instantiation of the Collision and Switching Lemmas, getting us to:**

```
local lemma GReal_G4 &m :
  (forall (O <: OR{Adv}),
   islossless O.bhash => islossless Adv(O).pre) =>
  (forall (O <: OR{Adv}),
   islossless O.hash => islossless Adv(O).post) =>
  `|Pr[GReal(Adv).main() @ &m : res] - Pr[G4.main() @ &m : res]| <=
   collBound.
```

G4 → G5

- **In the next transition, we make use of the fact that—if the game keeps within budget—the oracle will remain injective, i.e., collision free.**

G4 ➔ G5

```
local module G5 = {
  fun serverInit(db : db) : hdb * attrs_counts = {
    var i : int;
    var attr : attr;
    var tag : tag;
    var hdb : tag list;
    var attrsCounts : attrs_counts;
    attrsCounts = Map.empty;
    i = 0;
    hdb = [];
```

G4 ➔ G5

```
  while (i < length db) {
    attr = proj(nth db i);
    tag = OrInj.bhash(attr);
    hdb = hdb ++ [tag];
    if (in_dom attr attrsCounts) {
      attrsCounts.[attr] = proj attrsCounts.[attr] + 1;
    }
    else {
      attrsCounts.[attr] = 1;
    }
    i = i + 1;
  }
  return (hdb, attrsCounts);
}
```

G4 ➔ G5

```
fun clientQueryLoop
    (hdb : hdb, qrys : qrys, attrsCounts : attrs_counts) : view = {
  var view : view;
  var i, n : int;
  var tag : tag;
  var qry : attr;
  view = [];
  i = 0;
```

G4 ➔ G5

```
  while (i < length qrys) {
    qry = proj(nth qrys i);
    tag = OrInj.bhash(qry);
    if (in_dom qry attrsCounts) {
      n = proj attrsCounts.[qry];
    }
    else {
      n = 0;
    }
    view = view ++ [(qry, tag, n)];
    i = i + 1;
  }
  return view;
}
```

G4 → G5

```
fun view(qrys : qrys, db : db) : view = {
  var view : view;
  var hdb : hdb;
  var attrsCounts : attrs_counts;
  (hdb, attrsCounts) = serverInit(db);
  view = clientQueryLoop(hdb, qrys, attrsCounts);
  return view;
}

...
}.
```

G4 ➔ G5

- To prove this transition, we make use of a relational invariant:
  - the parts of `db` and `qrys` yet to be processed, `glob OrInj` and `hdb` are equal in the two games;
  - `OrInj.mp`, and `OrInj.range` are consistent and injective;
  - `card OrInj.range = OrInj.ctr`;
  - `!OrInj.over`;
  - the sum of the parts of `db` and `qrys` yet to be processed is no more than `budget – OrInj.ctr`;
  - `hdb`, `OrInj.mp` and `attrsCounts` (only in `G5`) are consistent.

G4 → G5

```
pred Inj (mp : ('a, 'b)map) (ran : 'b set) =
  (forall (y : 'b), in_rng y mp <=> mem y ran) /\
  (forall (x1 : 'a, x2 : 'a),
   in_dom x1 mp => in_dom x2 mp =>
   proj mp.[x1] = proj mp.[x2] =>
   x1 = x2).
```

- `numOccsUpTo` *`x`* *`ys`* *`i`* is the number of occurrences of *`x`* in the first *`i`* elements of *`ys`*; it's `0`, if *`i`* `< 0` or *`i`* `> length` *`ys`*:

```
op numOccsUpTo : 'a -> 'a list -> int -> int.
```

G4 ➔ G5

```
pred AttrsCounts
     (hdb : hdb) (mp : hash_map) (attrsCounts : attrs_counts) =
  (forall (tag : tag),
   mem tag hdb => in_rng tag mp) /\
  (forall (attr : attr),
   !in_dom attr attrsCounts =>
   !in_dom attr mp \/
   numOccsUpTo (proj mp.[attr]) hdb (length hdb) = 0) /\
  (forall (attr : attr),
   in_dom attr attrsCounts =>
   in_dom attr mp /\
   numOccsUpTo (proj mp.[attr]) hdb (length hdb) =
   proj attrsCounts.[attr]).
```

G4 ➔ G5

- **For example:**

```
local lemma G4_G5_serverInit :
  forall (qrys : qrys),
  equiv
  [G4.serverInit ~ G5.serverInit :
   ={db, glob OrInj} /\ OrInj.ctr{1} = card OrInj.range{1} /\
   Inj OrInj.mp{1} OrInj.range{1} /\ !OrInj.over{1} /\
   length db{1} + length qrys <= budget - OrInj.ctr{1} ==>
   ={glob OrInj} /\ res{1} = fst res{2} /\
   OrInj.ctr{1} = card OrInj.range{1} /\
   Inj OrInj.mp{1} OrInj.range{1} /\ !OrInj.over{1} /\
   AttrsCounts (fst res{2}) OrInj.mp{1} (snd res{2}) /\
   length qrys <= budget - OrInj.ctr{1}].
```

G5 → G6 → G7 → G8

- Having made use of injectivity, we now move back to using our standard oracle, Or—this takes us to G8.
- In these steps we use instantiation of the Collision and Switching Lemmas.
- We also use the Triangular Inequality.
- This take us to:

```
local lemma GReal_G8 &m :
  (forall (O <: OR{Adv}),
   islossless O.bhash => islossless Adv(O).pre) =>
  (forall (O <: OR{Adv}),
   islossless O.hash => islossless Adv(O).post) =>
  `|Pr[GReal(Adv).main() @ &m : res] - Pr[G8.main() @ &m : res]| <=
   2%r * collBound.
```

`G8` → `G9`

- `G9` is like `G8` except:
  - `serverInit` is inlined into `view`;
  - only queries (not all attributes) are counted in the database;
  - `clientQueryLoop` is renamed to `simulator` and no longer takes the hashed database as an argument;
  - apart from in the adversary's `pre` function, all hashing is done using `hash` (not `bhash`).

G8 ➔ G9

```
local module G9 = {
  fun simulator(qrys : qrys, qrysCounts : attrs_counts) : view = {
    var view : view;
    var i, n : int;
    var tag : tag;
    var qry : attr;
    view = [];
    i = 0;
```

G8 ➔ G9

```
  while (i < length qrys ) {
    qry = proj(nth qrys i);
    tag = Or.hash(qry);
    if (in_dom qry qrysCounts) {
      n = proj qrysCounts.[qry];
    }
    else {
      n = 0;
    }
    view = view ++ [(qry, tag, n)];
    i = i + 1;
  }
  return view;
}
```

G8 ➔ G9

```
fun view(qrys : qrys, db : db) : view = {
  var view : view;
  var qrysCounts : attrs_counts;
  var i : int;
  var attr : attr;
  var tag : tag;
  qrysCounts = Map.empty;
  i = 0;
```

G8 → G9

```
  while (i < length db) {
    attr = proj(nth db i);
    tag = Or.hash(attr);
    if (mem attr qrys) {
      if (in_dom attr qrysCounts) {
        qrysCounts.[attr] = proj qrysCounts.[attr] + 1;
      }
      else {
        qrysCounts.[attr] = 1;
      }
    }
    i = i + 1;
  }
  view = simulator(qrys, qrysCounts);
  return view;
}
```

G8 ➔ G9

```
    module A = Adv(Or)

    fun main() : bool = {
      ...
    }
  }.
```

G9 → G10 → G11

- In the next two transitions, we prepare the while loop of `view` for loop fission, and then perform loop fission.
- The preparation makes two parts—database hashing and query counting—of the while loop independent.

G9 ➔ G10

```
qrysCounts = Map.empty;
i = 0;
while (i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  attr' = proj(nth db i);
  if (mem attr' qrys) {
    if (in_dom attr' qrysCounts) {
      qrysCounts.[attr'] = proj qrysCounts.[attr'] + 1;
    }
    else {
      qrysCounts.[attr'] = 1;
    }
  }
  i = i + 1;
}
view = simulator(qrys, qrysCounts);
```

G10 ➔ G11

```
qrysCounts = Map.empty;
i = 0;
while (i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
```

G10 ➔ G11

```
while (j < length db) {
  attr' = proj(nth db j);
  if (mem attr' qrys) {
    if (in_dom attr' qrysCounts) {
      qrysCounts.[attr'] = proj qrysCounts.[attr'] + 1;
    }
    else {
      qrysCounts.[attr'] = 1;
    }
  }
  j = j + 1;
}
view = simulator(qrys, qrysCounts);
```

G11 ➔ G12

- **In the next transition, we move the query counting into its own function, `computeAnswers`.**
- **We also switch the order in which the query counting and database hashing are done.**

```
fun computeAnswers(qrys : qrys, db : db) : attrs_counts = {
  var qrysCounts : attrs_counts;
  var i : int;
  var attr : attr;
  qrysCounts = Map.empty;
  i = 0;
```

G11 ➔ G12

```
  while (i < length db) {
    attr = proj(nth db i);
    if (mem attr qrys) {
      if (in_dom attr qrysCounts) {
        qrysCounts.[attr] = proj qrysCounts.[attr] + 1;
      }
      else {
        qrysCounts.[attr] = 1;
      }
    }
    i = i + 1;
  }
  return qrysCounts;
}
```

G11 ➔ G12

```
fun view(qrys : qrys, db : db) : view = {
  var view : view;
  var qrysCounts : attrs_counts;
  var i : int;
  var tag : tag;
  var attr : attr;
  qrysCounts = computeAnswers(qrys, db);
  i = 0;
  while (i < length db) {
    attr = proj(nth db i);
    tag = Or.hash(attr);
    i = i + 1;
  }
  view = simulator(qrys, qrysCounts);
  return view;
}
```

Redundant Hashing

- What remains is to move the hashing

```
while (i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
```

of the database past the calls to `simulator` and `A.post`, to a point where it will be seen to be redundant.

- We will use lazy sampling to do this, once this is added to the new EasyCrypt.
- First we need to put the relevant code in a new function.

G12 ➔ G13

```
module A = Adv(Or)

fun viewPost(qrys : qrys, db : db) : bool = {
  var view : view;
  var qrysCounts : attrs_counts;
  var i : int;
  var tag : tag;
  var attr : attr;
  var b : bool;
```

G12 ➔ G13

```
    qrysCounts = computeAnswers(qrys, db);
    i = 0;
    while (i < length db) {
      attr = proj(nth db i);
      tag = Or.hash(attr);
      i = i + 1;
    }
    view = simulator(qrys, qrysCounts);
    b = A.post(view);
    return b;
  }
```

G12 ➔ G13

```
fun main() : bool = {
  var db : db;
  var qrys : qrys;
  var view : view;
  var b, suff : bool;
  Or.init();
  (db, qrys) = A.pre();
  suff = Or.suff(length db  + length qrys);
  if (suff) {
    b = viewPost(qrys, db);
  }
  else {
    b = A.post([]);
  }
  return b;
}
```

G13 → G14

- Now we want to use lazy sampling within the function `viewA2`.
- This is not yet implemented in the new EasyCrypt, but here is how it will go.

G13 ➔ G14

Eager

```
qrysCounts = computeAnswers(qrys, db);
i = 0;
while (i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
view = simulator(qrys, qrysCounts);
b = A.post(view);
return b;
```

G13 ➔ G14

Lazy

```
qrysCounts = computeAnswers(qrys, db);
view = simulator(qrys, qrysCounts);
b = A.post(view);
i = 0;
while (i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
return b;
```

G13 ➔ G14

- We must show that a single call of `Or.hash` commutes with the database hashing.
- Precondition:

```
pre = ={db, Or.mp, _attr}
```

- Postcondition:

```
pre = ={db, Or.mp, _attr}
```

G13 ➔ G14

- **Statement 1 (Eager):**

```
i = 0;
while (i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
if (!in_dom _attr Or.mp) {
  Or.mp.[_attr] = $tagDistr;
}
```

- **Statement 2 (Lazy):**

```
if (!in_dom _attr Or.mp) {
  Or.mp.[_attr] = $tagDistr;
}
i = 0;
while (i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
```

G13 ➔ G14

- **Cases Analysis:**
  - `in_dom _attr Or.mp`;
  - `!in_dom _attr Or.mp`;
    - `!mem _attr db`;
    - `mem _attr db.`
- **In the case `!in_dom _attr Or.mp /\ mem _attr db`, we use `splitwhile`, `unroll` and `inline` to uncover the point in the second game where `Or.mp.[_attr]` is chosen.**
- **We then use `derandomize`, so we can match the values chosen in the two games.**

G13 → G14

Eager

```
i = 0;
while (i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
if (!(in_dom _attr Or.mp)) Or.mp.[_attr] = $tagDistr;
```

Lazy

```
if (!(in_dom _attr Or.mp)) Or.mp.[_attr] = $tagDistr;
i = 0;
while (i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
```

G13 ➔ G14

```
i = 0;
while (proj(nth db i) <> _attr /\ i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
while (i < length db) {
  attr = proj(nth db i);
  if (!in_dom attr mp) Or.mp.[attr] = $tagDistr;
  i = i + 1;
}
if (!(in_dom _attr Or.mp)) Or.mp.[_attr] = $tagDistr;
```

G13 ➔ G14

```
if (!in_dom _attr Or.mp) Or.mp.[_attr] = $tagDistr;
i = 0;
while (proj(nth db i) <> _attr /\ i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
while (i < length db) {
  attr = proj(nth db i);
  if (!in_dom attr Or.mp) Or.mp.[attr] = $tagDistr;
  i = i + 1;
}
```

G13 ➔ G14

```
i = 0;
while (proj(nth db i) <> _attr /\ i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
if (i < length db) {
  attr = proj(nth db i);
  if (!in_dom attr mp) Or.mp.[attr] = $tagDistr;
  i = i + 1;
}
while (i < length db) {
  attr = proj(nth db i);
  if (!in_dom attr mp) Or.mp.[attr] = $tagDistr;
  i = i + 1;
}
if (!(in_dom _attr Or.mp)) Or.mp.[_attr] = $tagDistr;
```

G13 ➔ G14

```
if (!in_dom _attr Or.mp) Or.mp.[_attr] = $tagDistr;
i = 0;
while (proj(nth db i) <> _attr /\ i < length db) {
  attr = proj(nth db i);
  tag = Or.hash(attr);
  i = i + 1;
}
if (i < length db) {
  attr = proj(nth db i);
  if (!in_dom attr Or.mp) Or.mp.[attr] = $tagDistr;
  i = i + 1;
}
while (i < length db) {
  attr = proj(nth db i);
  if (!in_dom attr Or.mp) Or.mp.[attr] = $tagDistr;
  i = i + 1;
}
```

G13 ➔ G14

```
_tag = $tagDistr;
i = 0;
while (proj(nth db i) <> _attr /\ i < length db) {
  ...
}
if (i < length db) {
  attr = proj(nth db i);
  if (!in_dom attr mp) Or.mp.[attr] = _tag;
  i = i + 1;
}
while (i < length db) {
  ...
}
if (!(in_dom _attr Or.mp)) Or.mp.[_attr] = $tagDistr;
```

G13 → G14

```
_tag = $tagDistr;
if (!in_dom _attr Or.mp) Or.mp.[_attr] = _tag;
i = 0;
while (proj(nth db i) <> _attr /\ i < length db) {
  ...
}
if (i < length db) {
  attr = proj(nth db i);
  if (!in_dom attr Or.mp) Or.mp.[attr] = $tagDistr;
  i = i + 1;
}
while (i < length db) {
  ...
}
```

G14 → GIdeal

- **Because the postcondition of the needed equivalence for Main**

```
local lemma G14_GIdeal_main :
equiv[G14.main ~ GIdeal(Adv).main : true ==> ={res}].
```

we can remove the hashing of the database:

G14 $\rightarrow$ GIdeal

```
fun view(qrys : qrys, db : db) : view = {
  var view : view;
  var qrysCounts : attrs_counts;
  qrysCounts = computeAnswers(qrys, db);
  view = simulator(qrys, qrysCounts);
  return view;
}

module A = Adv(Or)
```

G14 ➔ GIdeal

```
fun main() : bool = {
  var db : db;
  var qrys : qrys;
  var view : view;
  var b, suff : bool;
  Or.init();
  (db, qrys) = A.pre();
  suff = Or.suff(length db  + length qrys);
  if (suff) {
    view = view(qrys, db);
  }
  else {
    view = [];
  }
  b = A.post(view);
  return b;
}
```

Overall Conclusion

```
lemma GReal_GIdeal :
  forall (Adv <: ADV{Or}) &m,
    (forall (O <: OR{Adv}),
     islossless O.bhash => islossless Adv(O).pre) =>
    (forall (O <: OR{Adv}),
     islossless O.hash => islossless Adv(O).post) =>
    `|Pr[GReal(Adv).main() @ &m : res] -
      Pr[GIdeal(Adv).main() @ &m : res]| <=
    2%r * collBound.
```

Conclusions

- Limiting the adversary’s use of the random oracle was subtle, since the protocol itself depends upon the oracle working.
- The Collision Lemma—proved using the Failure Event Lemma—gave us an upper bound on the probability of hash collisions occurring. We instantiated it twice.
- We proved the Switching Lemma—moving between oracles with and without collisions—using reasoning up to failure, and instantiated it twice.
- When moving to the game with perfect counting, we worked with an injective—collision free—oracle, made use of a complex relational invariant, and did careful budgeting.
- Lazy random sampling played an essential role in the transition to the ideal game.
- Even such a relatively simple protocol was fairly hard work to prove secure.

Future Work

- Prove the security of the Third Party (TP) view of Mini-APP.
- Finish applying the techniques learned/developed to more realistic protocols, e.g., the full UCI APP protocol and SPAR protocols.
- Discover and implement ways of reducing the difficulty of carrying out such proofs:
  - new proof techniques;
  - new theories;
  - new abstraction mechanisms;
  - new tactics.