Towards Information-theoretic Security Proofs in EasyCrypt for Multi-party Protocols

Alley Stoughton

IACR School on Computer-aided Cryptography
University of Maryland
June 1–4, 2015

SPAR Formal Methods Project

- As part of the IARPA SPAR Program, the SPAR Formal Methods Team—at MIT Lincoln Laboratory and the Naval Research Laboratory—undertook the verification in EasyCrypt of a Protocol developed by the University of California, Irvine, as part of IARPA's APP (Automatic Privacy Protection) Program
- The UCI PPSSI (Privacy-preserving Sharing of Sensitive Information) Protocol is a three party Private Information Retrieval (PIR) protocol involving a relational database
- Database queries are handled in such a way that:
  - The Client only learns query results
  - The Server—database holder—doesn't learn which queries are made
  - An untrusted Third Party is needed to make this work

Genesis of PCR Protocol

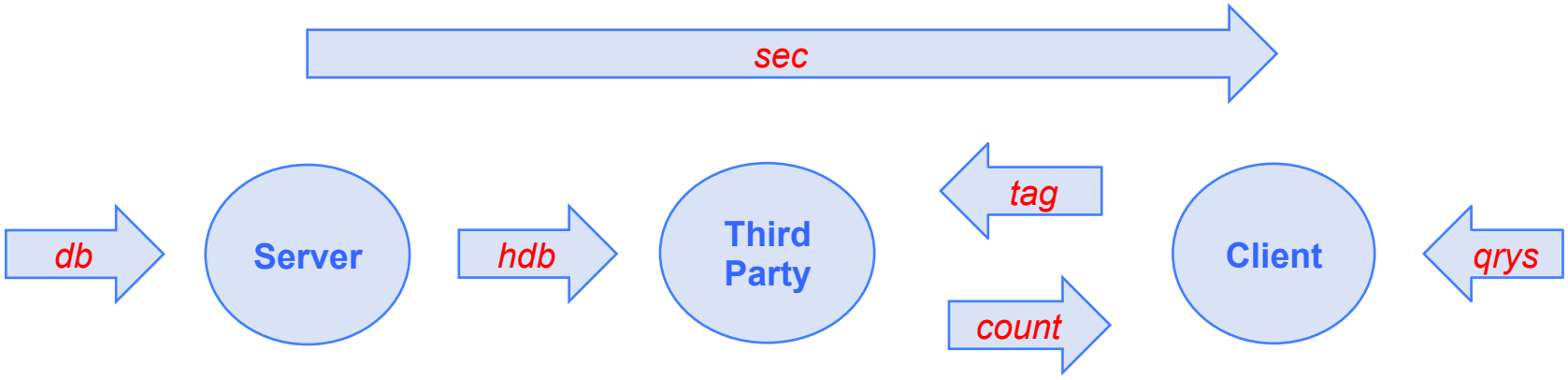
- The verification of the UCI PPSSI Protocol required our learning about/developing various sophisticated techniques
- Because of the complexity of the protocol, it was sub-optimal to be learning/developing these techniques in the context of the full UCI PPSSI proofs
- Consequently, I began proving the security of a simpler protocol in parallel with my work on the UCI PPSSI Third Party (Isolated Box) proof
- The protocol and proof evolved over time; eventually, I named the protocol PCR, for Private Count Retrieval

PCR Protocol

- In the PCR Protocol, a *database* is one-dimensional: it consists of a list of *elements*
- Each *query* is also an element—a request for the *count* of the number of times it occurs in the database
- The Client should only learn the counts for its queries
  - E.g., it shouldn't learn the order of the database, or counts of elements it doesn't ask about
- The Server shouldn't learn what queries the Client makes
- To make this work, the protocol uses an untrusted Third Party, which should learn nothing about the database and queries other than *patterns*

PCR Protocol

- The Server randomly generates a secret, *sec*, and shares it with the Client
- The Server randomly shuffles its database, *db*, turning the result into a hashed database, *hdb*, and sending *hdb* to the Third Party (TP)
  - Each element, *elem*, of *db* is turned into the hash of (*elem*, *sec*)
- For each query, *qry*, the Client hashes (*qry*, *sec*), and asks the TP for the number of occurrences of this hash tag in *hdb*

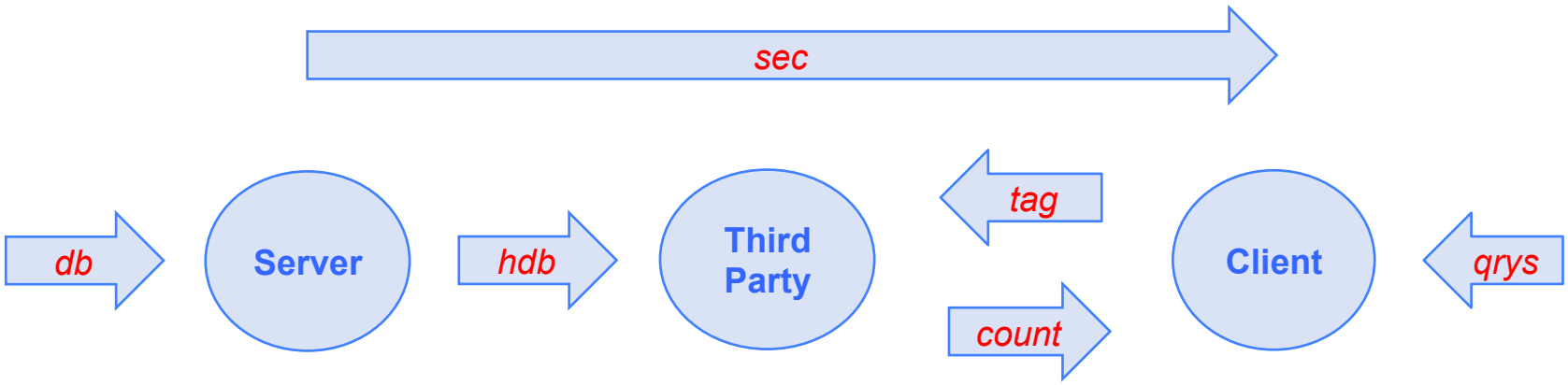


Secrets and Random Oracle

- A secret is a bit string of length *secLen*
- A hash tag is a bit string of length *tagLen*
- Hashing is done using a random oracle, consisting of a map to which new elements are added, dynamically
  - element/secret pairs are mapped to hash tags
- Normally, there will be many fewer hash tags than elements
- If a hash collision occurs, the Client's results may be inflated
  - E.g., if the database consists of elements *x* and *y*, but (*x*, *sec*) and (*y*, *sec*) hash to the same hash tag, then the count for query *x* will be 2 not 1
  - But hash collisions will be very unlikely—assuming we choose *tagLen* to be big

Protocol Security

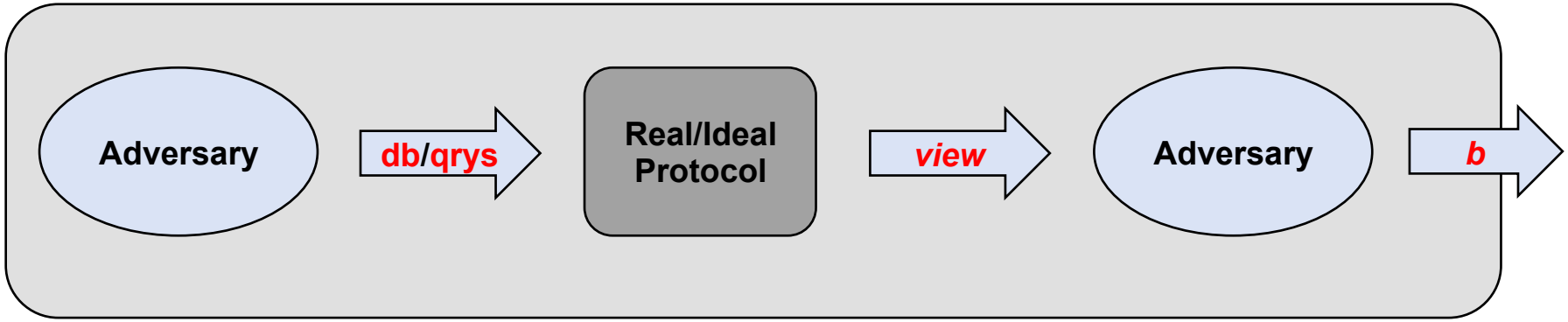
- **Informally, from an honest but curious perspective:**
  - **The protocol is secure against the Server, as the Server doesn't learn anything about *qrys***
  - **The protocol is secure against the Client, because the Client only learns the counts of the queries it makes**
  - **The protocol is secure against the TP, because hashing isn't efficiently invertible, TP very unlikely to guess *sec,* and *db* shuffled before hashing—so the TP will only learn element *patterns***



Real and Ideal Games

- We formalize security of the protocol using pairs of cryptographic games—one pair for each protocol party (Server, Client, Third Party (TP))
- The "real" games are based on the protocol as described above
  - Everything the party sees is recorded in its *view*
- The "ideal" game for a given party is based on a variant of the protocol in which it is obvious the party doesn't learn anything it shouldn't
  - The party's view must be constructed from the available information

Adversary Model



- Our games are parameterized by an Adversary with access to the random oracle
- Adversary allowed to maintain state between its calls
- The protocol is said to be *secure against* the given party 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

Information-theoretic Proofs

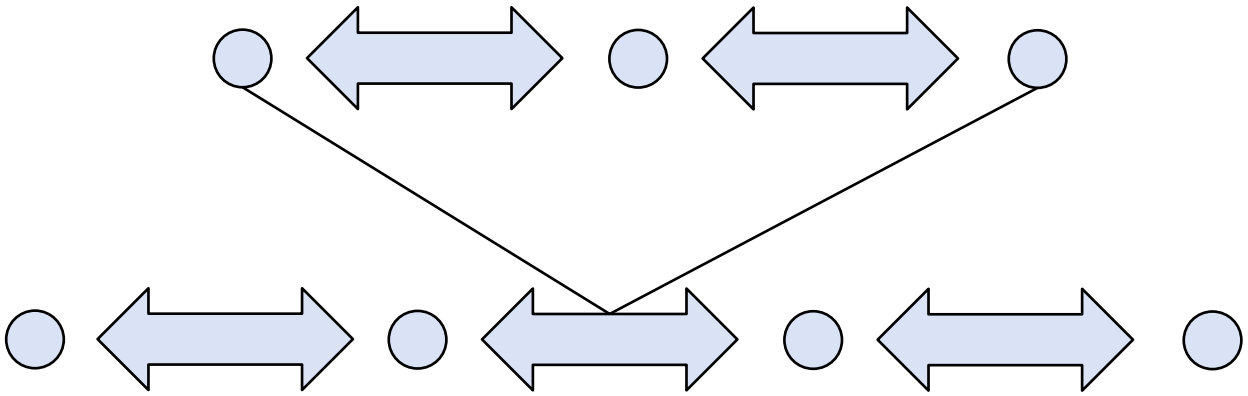
- We use a random oracle so as to be able to do information-theoretic proofs
- Random oracle has encapsulated state based on random choices not known to Adversary, and we can limit the Adversary's use of random oracle
- No need to manually verify efficiency of constructed adversaries
- Able to obtain concrete upper-bounds on game differences, involving parameters limiting Adversary

EasyCrypt Proof

- About 5,600 lines of proof script
- Proofs make heavy use of abstraction—reusable lemmas
- Proofs are both *horizontal* and *vertical*
  - *Horizontal*: sequences of games, connecting real and ideal games
    - In each step, we upper-bound the distance between the games

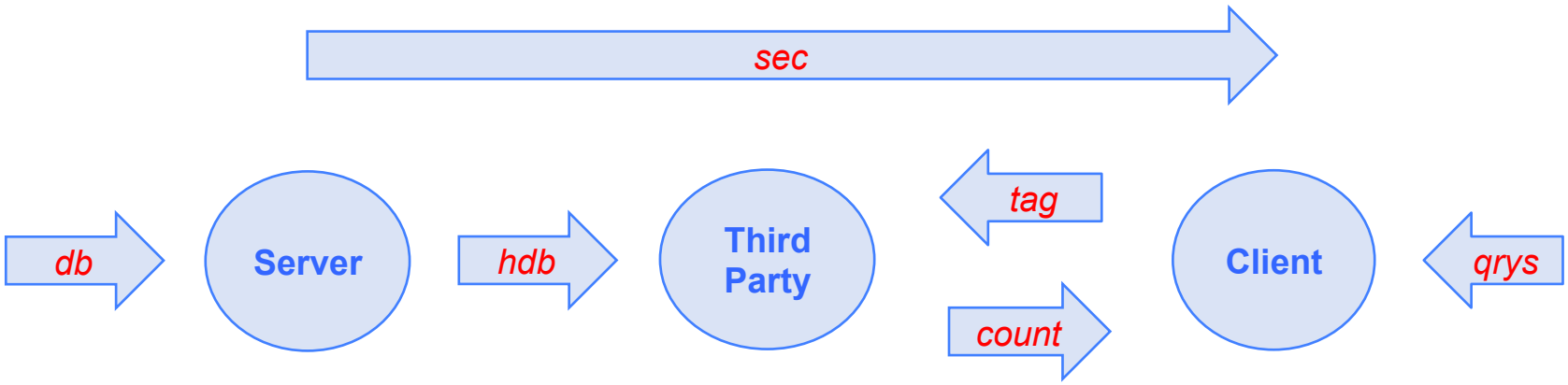
EasyCrypt Proof

- About 5,300 lines of proof script
- Proofs make use of abstraction—reusable lemmas/theories
- Proofs are both *horizontal* and *vertical*
  - *Horizontal*: sequences of games, connecting real and ideal games
    - In each step, we upper-bound the distance between the games
  - *Vertical*: reductions



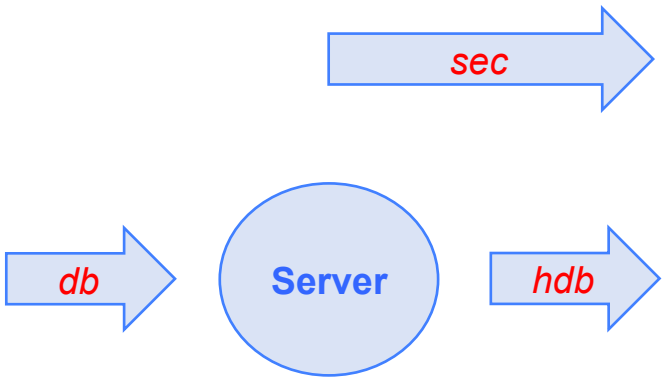
Security Against Server

- In the Server's ideal game, the Client and TP are absent
  - Nothing is done with *qrys*—in particular, the queries aren't hashed



Security Against Server

- In the Server's ideal game, the Client and TP are absent
  - Nothing is done with *qrys*—in particular, the queries aren't hashed
- Because the Server generates *sec*, its choice of *db* may be influenced by *sec*
- Consequently, the initial call to the Adversary is given *sec*, so its choice of *db* and *qrys* may be a function of *sec*—giving us a strong security guarantee

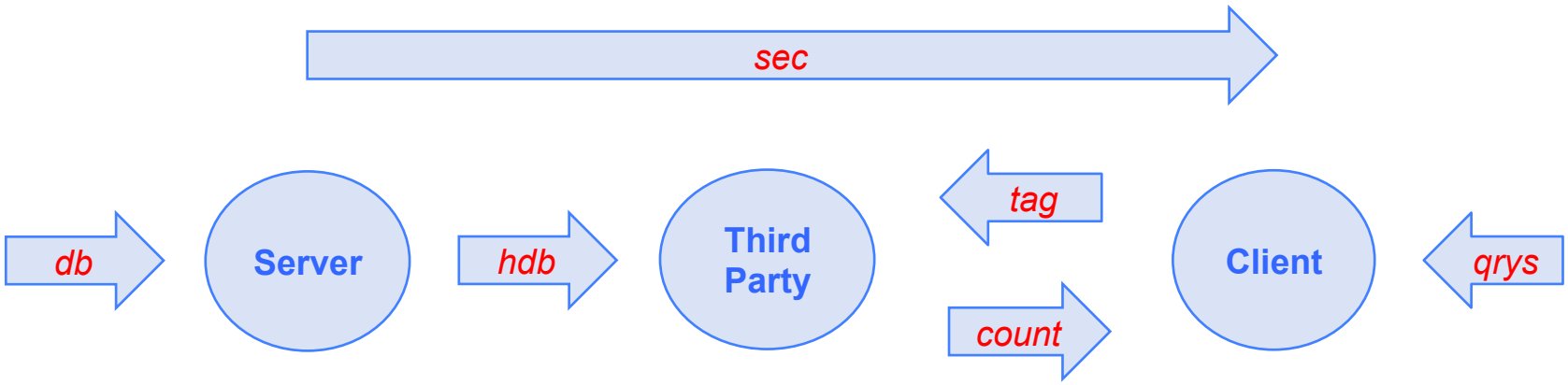


Security Against Server

- We prove that the absolute value of the difference between the probabilities that the Adversary returns true in the real and ideal games is 0—i.e., the Adversary is equally likely to return true in the two games
- In the sequence of games connecting the real and ideal games, we must get rid of the hashing done by the Client
  - With random oracle model, this takes some work—must show that even though oracle's map changes, this makes no difference
  - Proved reusable lemma for removing redundant hashing—using EasyCrypt's eager sampling tactics in a novel way

Security Against Third Party

- In the TP's ideal game, Server and Client do their element hashing using a private random oracle—for hashing elements, not element/secret pairs
  - So TP will only learn element *patterns* from view
  - Won't learn anything about database order



Security Against Third Party

- The Adversary can't be given *sec*, since otherwise it could differentiate real and ideal games
  - See if *sec* paired with elements of *db* hashes to permutation of hashed database part of view—very unlikely to happen in ideal game
- The Adversary's choice of *db*/*qrys* can't be allowed to depend on *sec*—even if state after generation of *db*/*qrys* doesn't carry over to final call of Adversary
  - *db* could encode the secret: use *n* copies of a single element *elem*, where *n* encodes the *secLen* bits of the secret using unique prime factorization
  - For $1 \leq i \leq \text{secLen}$, include *i*-th prime in product iff *i*-th bit is 1
  - Then secret will be recoverable from the *length* of hashed database, letting Adversary differentiate games

Security Against Third Party

- The Adversary can search for value of *sec* that correlates with *db*, *qrys*, the tags of the view, and the (non-private) random oracle
  - Will succeed in real game, but unlikely to succeed in ideal game
- Consequently, we must limit the number of distinct calls the Adversary may make to the random oracle to some limit, *limit*
  - Subsequently, a dummy value is returned, but previously-hashed element/secret pairs hash as before

Security Against Third Party

- **We prove that the absolute value of the difference between the probabilities that the Adversary returns true in the real and ideal games is upper-bounded by *limit* / 2^{secLen}**

Security Against Third Party

- We use a reduction to a lemma upper-bounding the distance between games in which an adversary is given access to two versions of a secrecy random oracle:
  - A procedure lhash for hashing up to *limit* distinct element/secret pairs—excess inputs yield dummy value
  - A procedure hash for hashing elements
- The secrecy random oracle's initialization init procedure is given *sec*, but adversary isn't given *sec*
- In first oracle implementation, hashing an element *elem* using hash hashes (*elem*, *sec*) in map used by lhash
- In second oracle implementation, hash's map is independent from lhash's map

Security Against Third Party

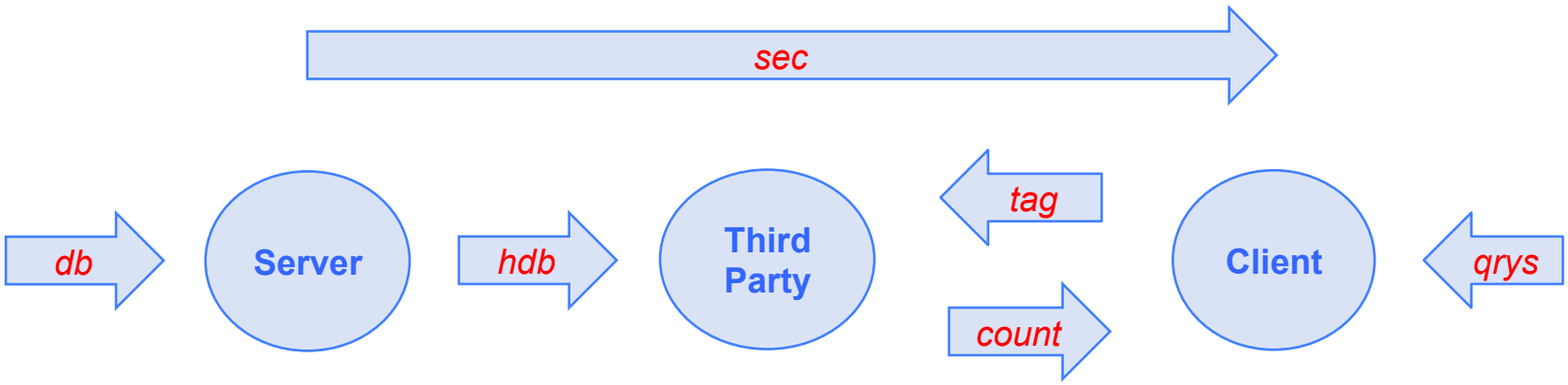
- If Adversary never calls lhash with (*elem*, *sec*), for some *elem*, then it won't be able to tell the games apart
- Consequently, we are able to upper-bound the distance between the games by $limit / 2^{secLen}$

Security Against Third Party

- Lemma bounding distance between games involving secrecy random oracles is reduced to lemma about secret guessing game
- Guessing oracle's initialization procedure init takes in *sec*
- Guessing oracle's guessing procedure guess lets adversary try to guess secret—but it doesn't learn if it succeeded
- Adversary allowed *limit* calls to guess—after which it's a no-op
- We are able to upper-bound probability that adversary guesses secret by $limit / 2^{secLen}$
  - Uses EasyCrypt's Probabilistic Hoare Logic

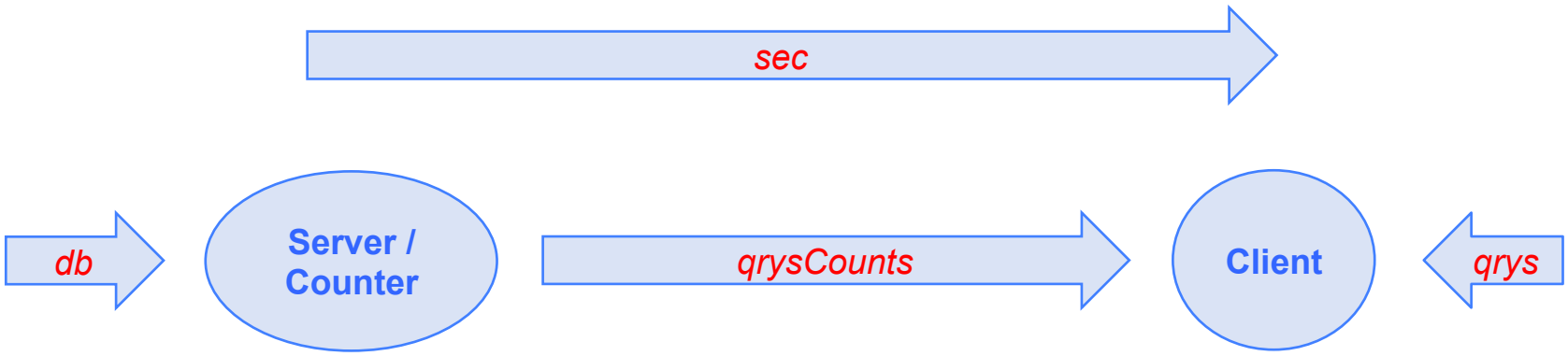
Security Against Client

- In the Client's ideal game, the TP is absent, and the Server only generates *sec* and gives it to the Client
- The Client is given map containing counts of (only) its queries



Security Against Client

- In the Client's ideal game, the TP is absent, and the Server only generates *sec* and gives it to the Client
- The Client is given map containing counts of (only) its queries
  - It consults this map instead of interacting with TP—and thus its counts are never inflated
  - But it still hashes query/*sec* pairs, as the resulting hash tags go in its view



Security Against Client

- Because Client receives *sec*, it could influence its choice of *qrys*
- Server generates *sec*, and could let *sec* influence its choice of *db*
- So for strong security guarantee, we supply *sec* to initial call to Adversary
- If Adversary can do unlimited hashing, it can find and exploit collision
  - It can find $elem_1$, $elem_2$ such that hashing ($elem_1$, *sec*) and ($elem_2$, *sec*) give same tag
  - Then *db* = [$elem_1$], *qrys* = [$elem_2$] will allow it to distinguish real and ideal games

Security Against Client

- If Adversary can pick *db* and *qrys* of arbitrary size, it can let *db* = *qrys* = list of distinct elements of length > number of hash tags
  - Then all query counts in ideal game will be 1, but at least one count in real game will be more than 1
- Consequently, we give Adversary a hashing budget, *budget*, that is ≤ number of hash tags
- If |*db*| + |*qrys*| + hashes done by Adversary in initial call isn't ≤ *budget*, Adversary loses game (is given empty view)
- We prove that the absolute value of the difference between the probabilities that the Adversary returns true in the real and ideal games is upper-bounded by $(budget \times (budget - 1)) / 2^{tagLen}$

Security Against Client

- In the sequence of games connecting the real and ideal games, we transition in and out of games in which the random oracles stay collision-free as long as no more than *budget* distinct hashes are done
  - The Adversary's final call uses collision-possible hashing—at that point, it can't learn anything through a collision happening
  - Our Switching Lemma is proved using intermediate games, and used twice
  - Distance between games of Switching Lemma upper-bounded by $(budget \times (budget - 1)) / 2^{(tagLen + 1)}$

Security Against Client

- The transition from relying on the TP for counting occurrences of a query's tag to looking the query's count up in a supplied map involves using a complex relational invariant
  - Must show that counting hash tags in hashed database is equivalent to looking them up in map
  - Must prove that hashing budget is respected by Server/Client, because collisions could break this equivalence
- Before reaching ideal game, must get rid of hashing done by Server—we reuse lemma for removing redundant hashing
- We must also prove that randomly shuffling the database and then computing the query counts map is equivalent to simply computing the query counts map without first shuffling—because the later is what the ideal game does

Future Work

- **Adapt PCR proof to work with adaptive query choice by the adversary**
- **Prove the security of more complex private information retrieval protocols—e.g., ones like the UCI PPSSI protocol—in EasyCrypt**
  - **My plan is to continue doing information-theoretic proofs**
  - **Model encryption and other constructions using oracles with encapsulated state, to which adversary's access may be limited**

Acknowledgements

- **SPAR Formal Methods Team**

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

- **MIT Lincoln Laboratory SPAR Team**

  Robert Cunningham, Emily Shen, Mayank Varia (formally MIT Lincoln Laboratory), Arkady Yerukhimovich

- **EasyCrypt Team**

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