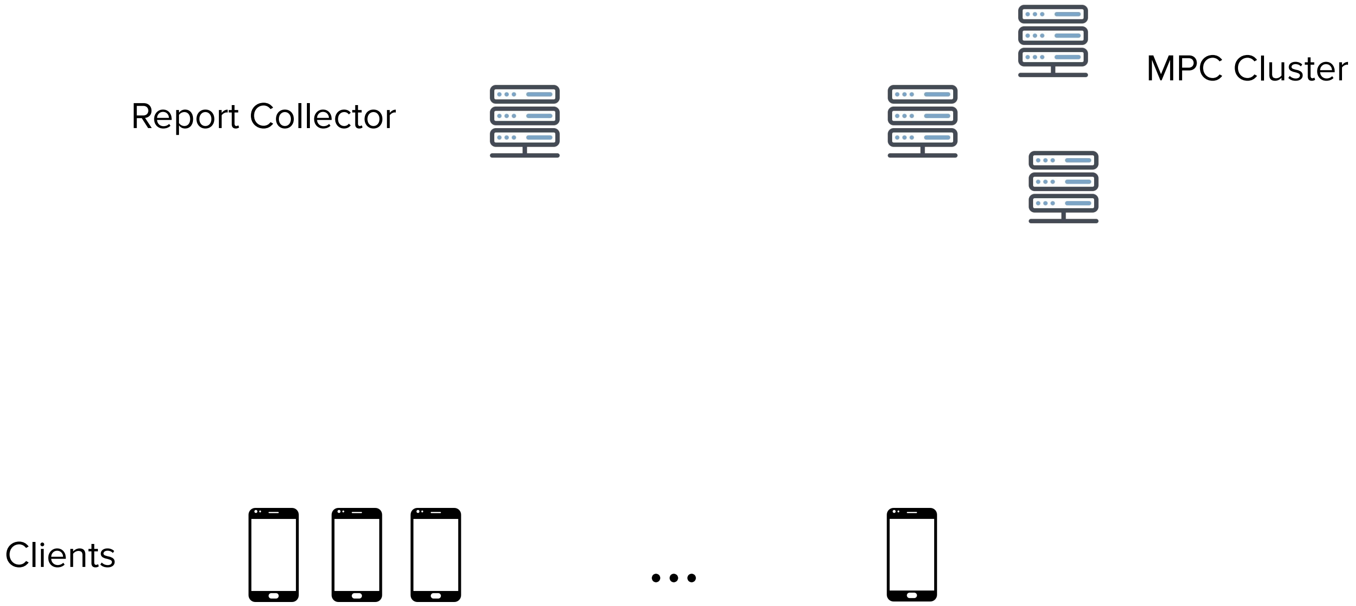
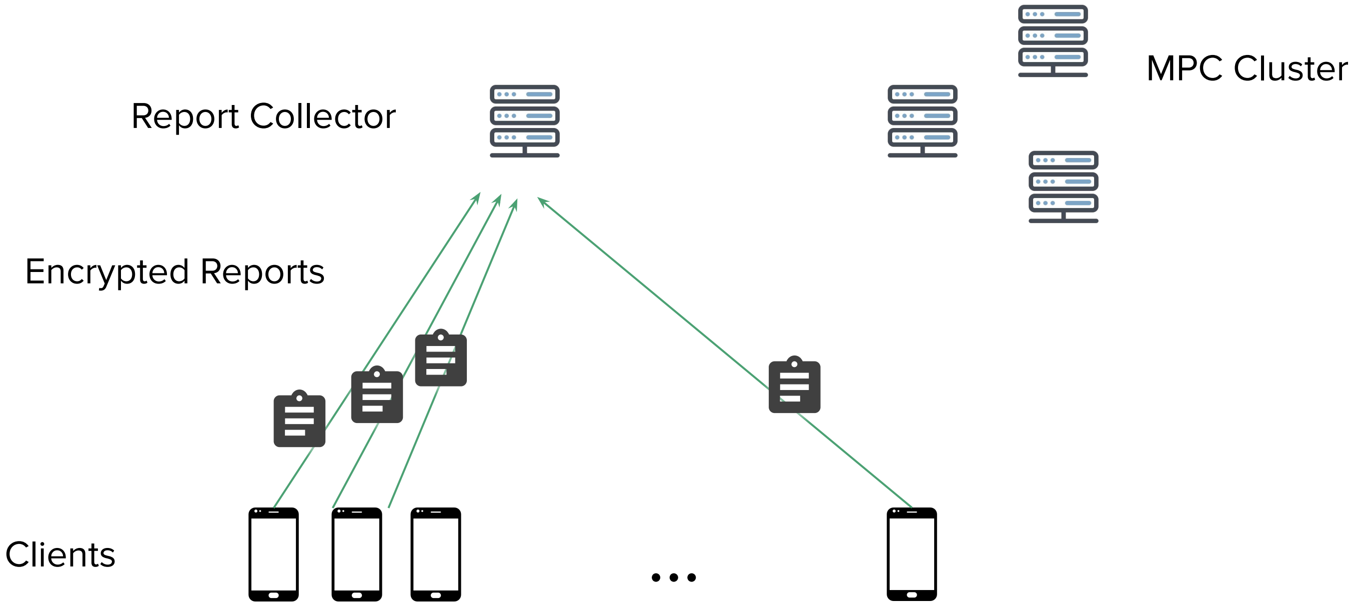
Maliciously Secure Partitioning

Adria Gascon, Mariana Raykova, **Phillipp Schoppmann**, Karn Seth

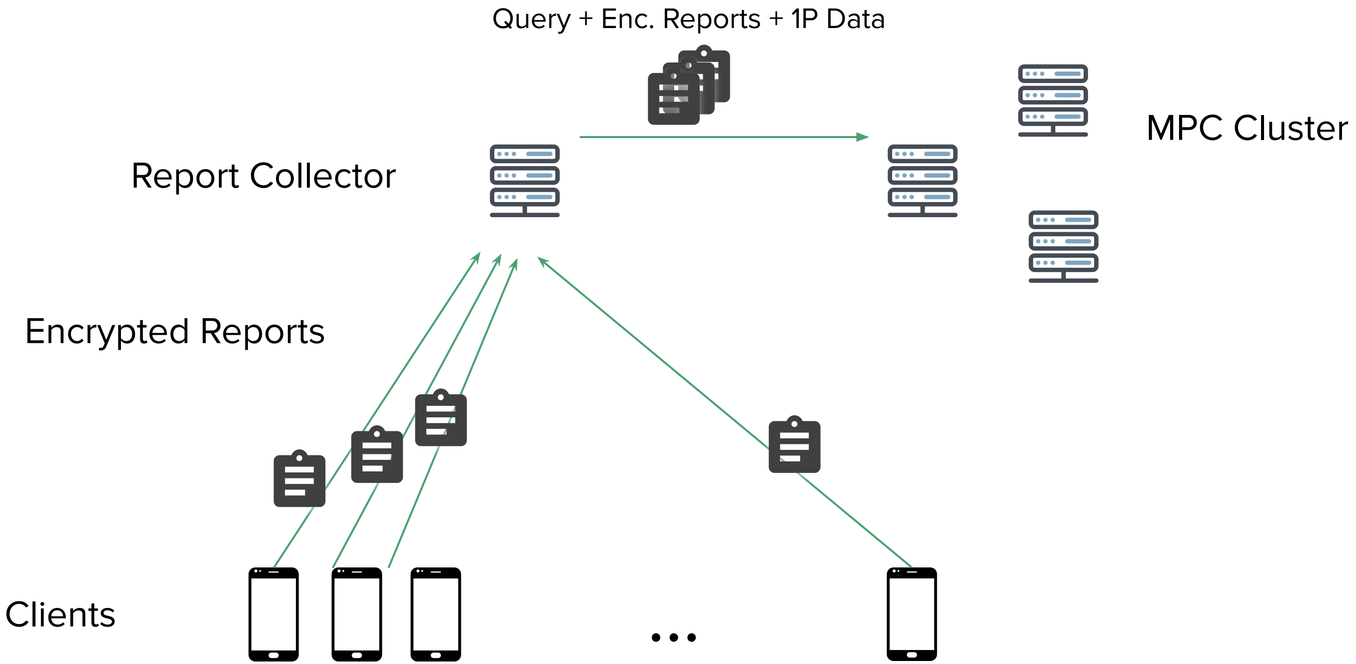
Background: Interoperable Private Attribution (IPA)



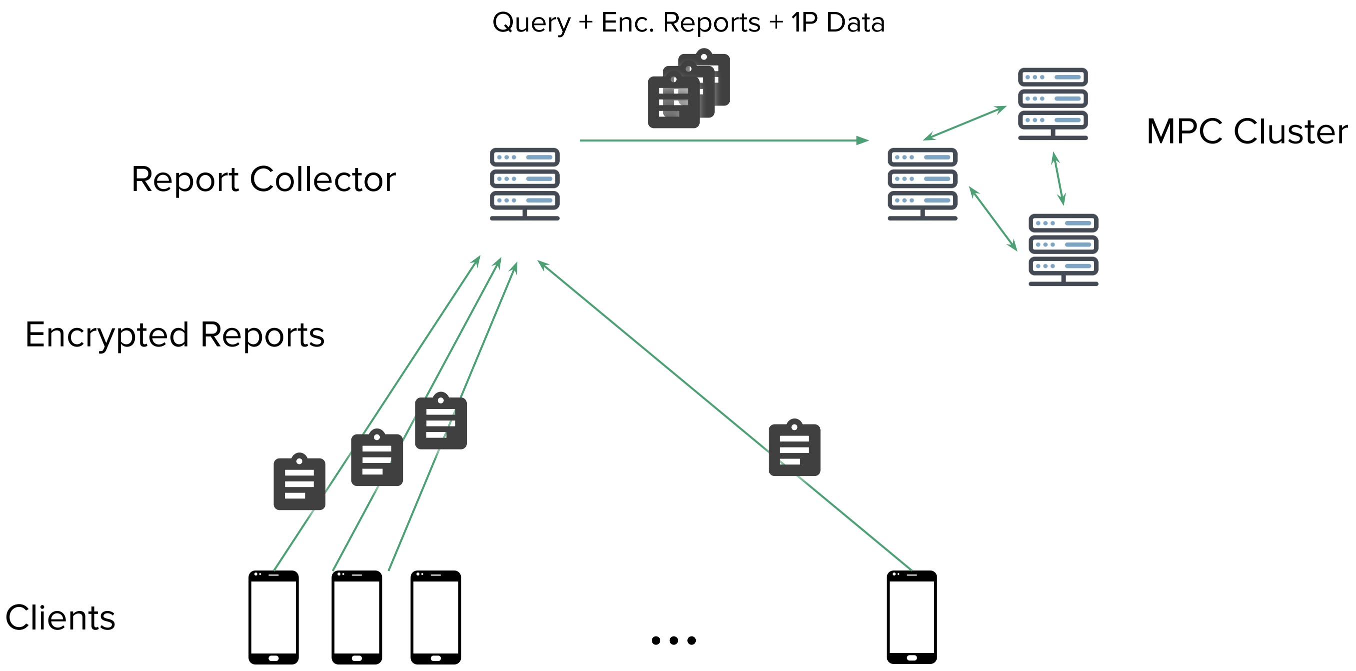
Background: Interoperable Private Attribution (IPA)



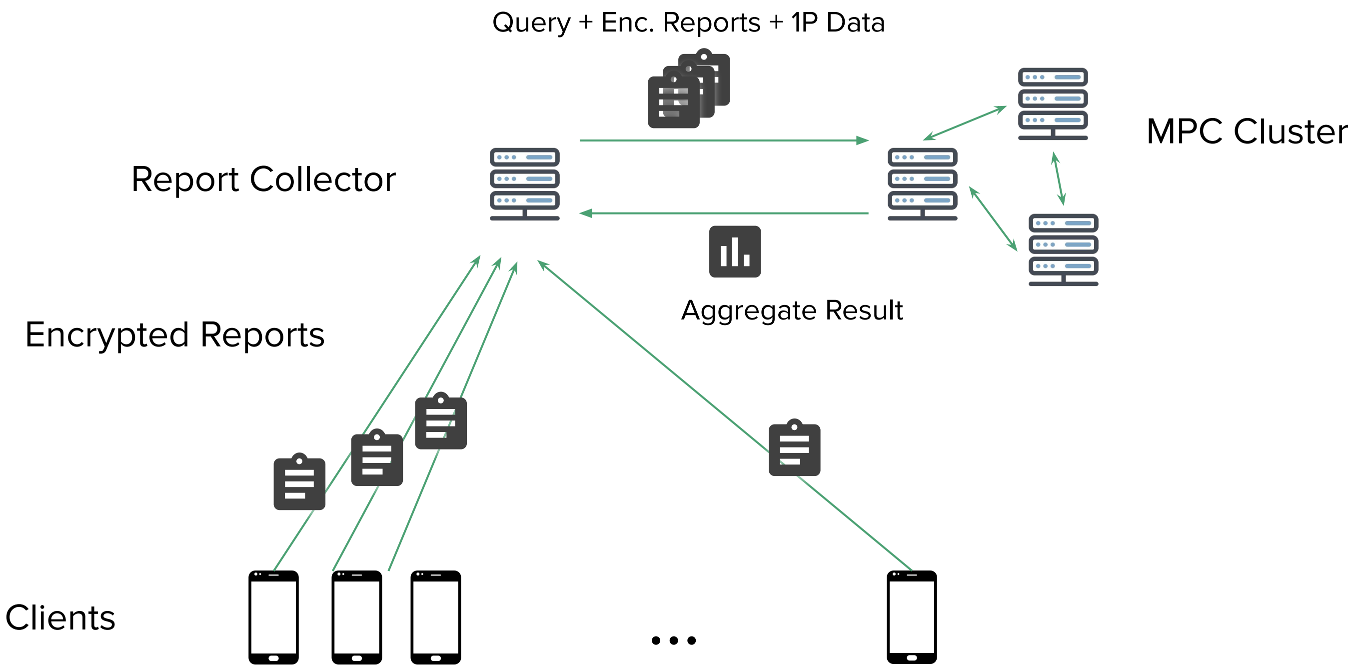
Background: Interoperable Private Attribution (IPA)



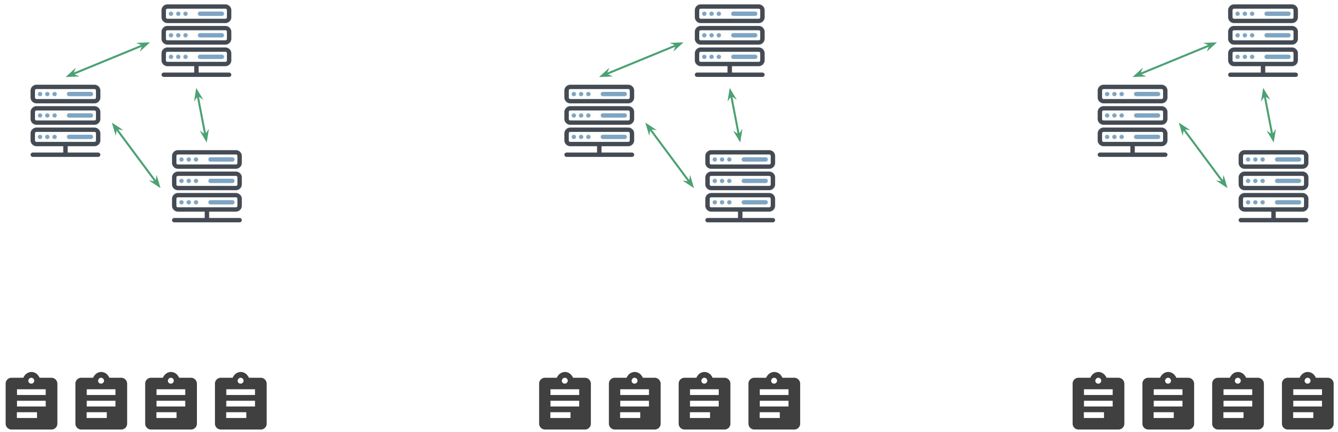
Background: Interoperable Private Attribution (IPA)



Background: Interoperable Private Attribution (IPA)



Sharding MPC Clusters



Challenge: How to partition reports across shards, s.t. all reports of the same client end up in the same shard?

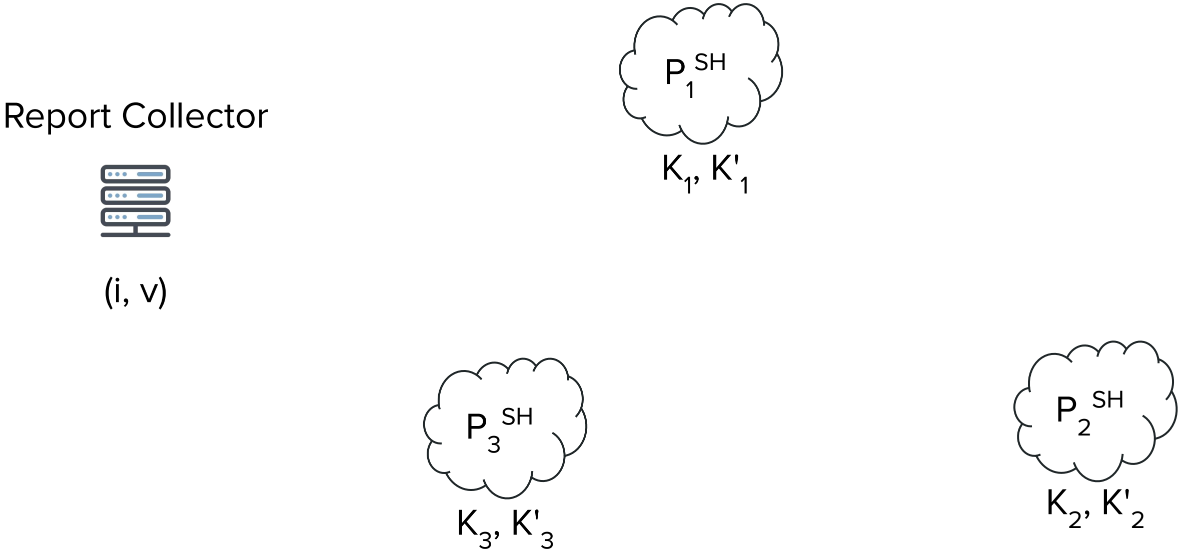
Goals

- Inputs from the same client end up in the same shard
- Low communication overhead and round complexity
- Partitioning must not affect correctness / utility of downstream computation
- Security against a single malicious party out of three

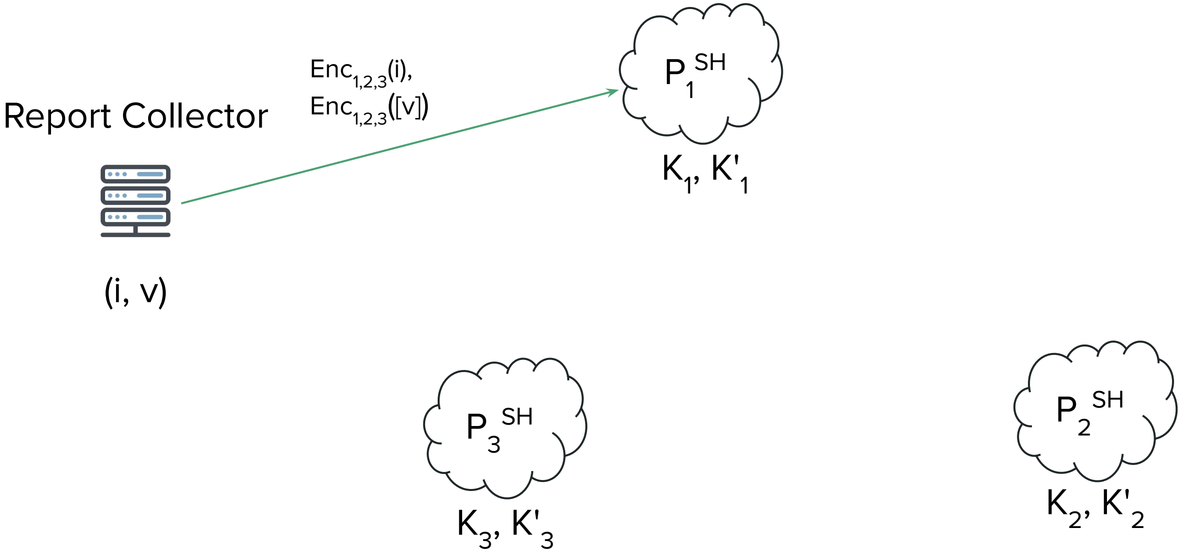
Assumptions

- Bound M on the number of contributions per client
- Lots of clients (billions), few shards (thousands)

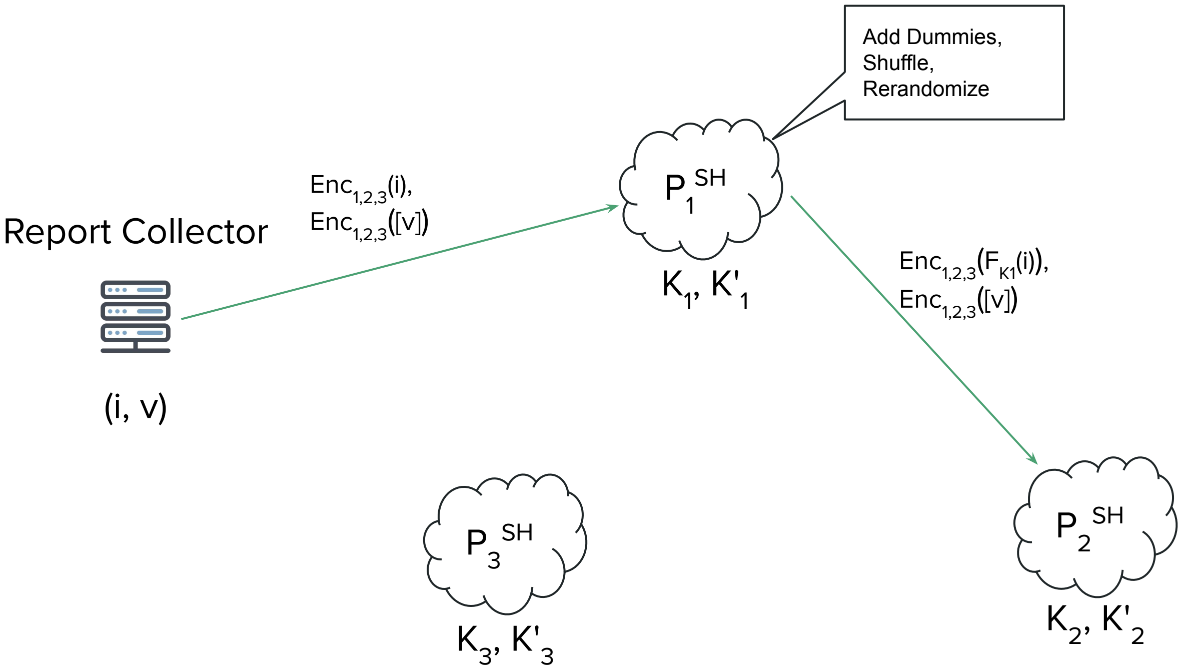
First Step: Semi-Honest Protocol with Dishonest Majority



First Step: Semi-Honest Protocol with Dishonest Majority

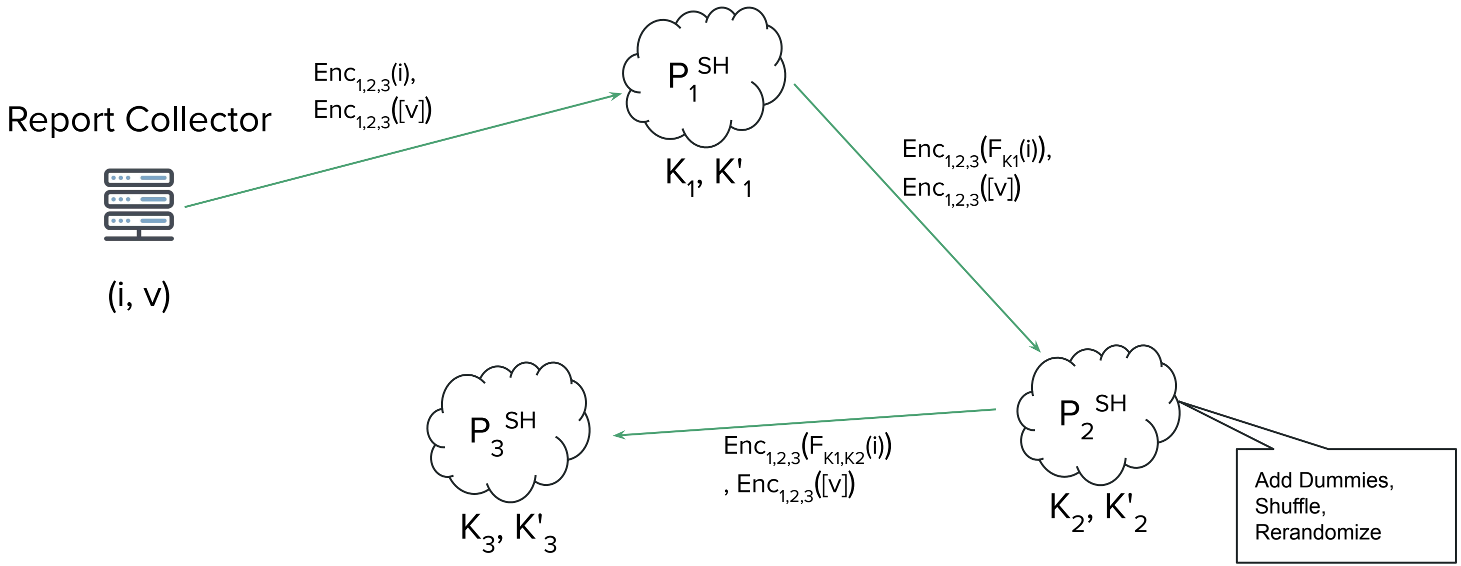


First Step: Semi-Honest Protocol with Dishonest Majority



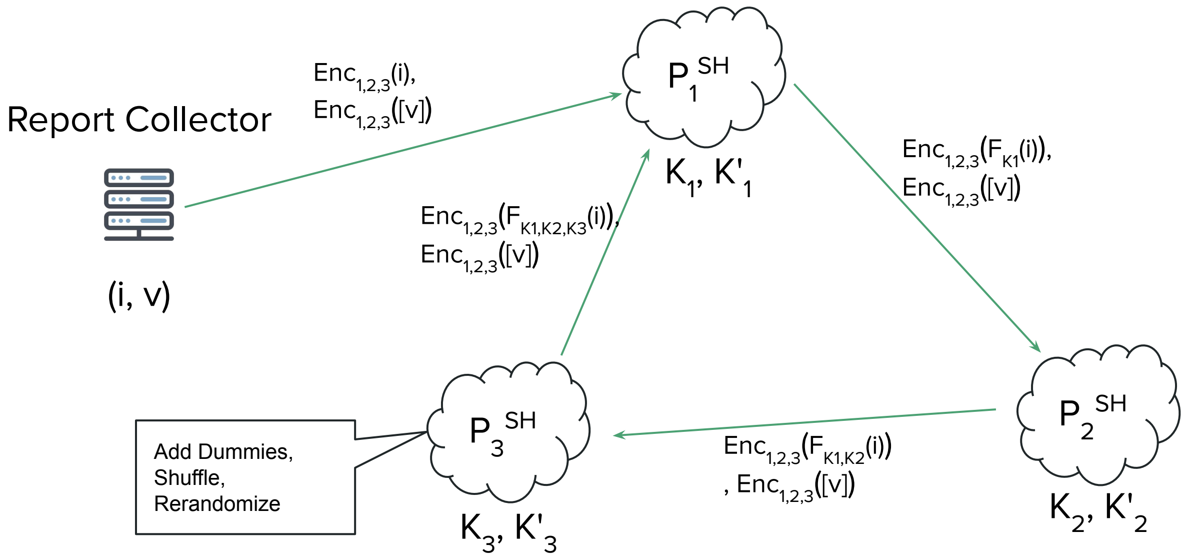
F_K: Pseudorandom Function with key K

First Step: Semi-Honest Protocol with Dishonest Majority



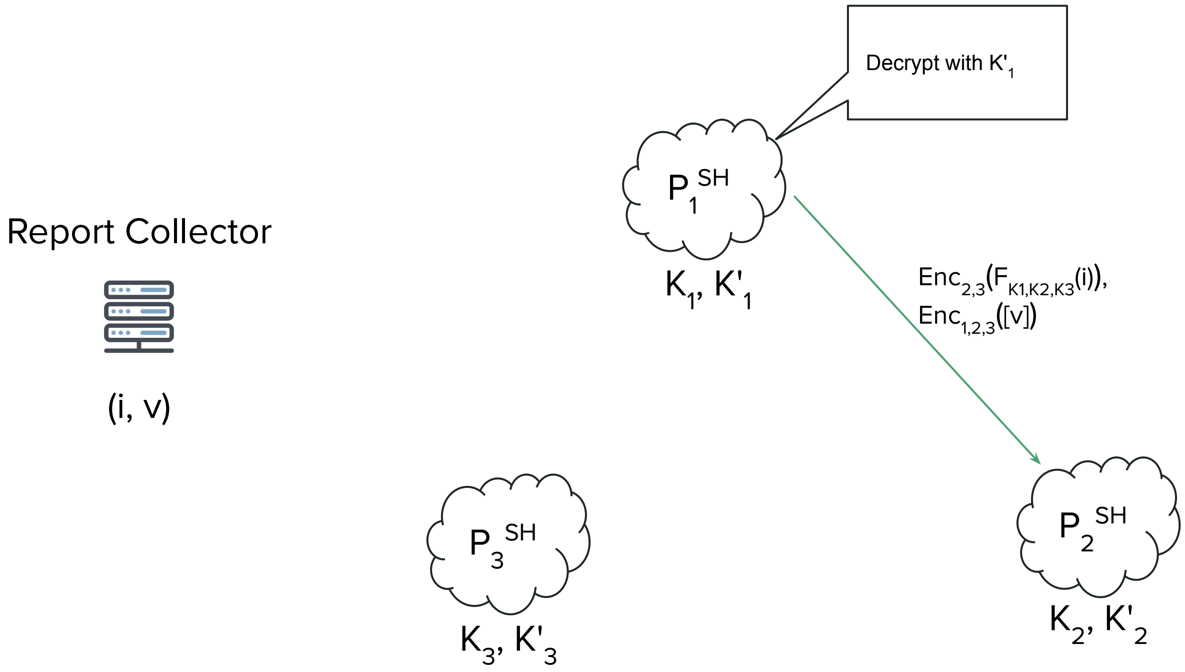
F_K: Pseudorandom Function with key K

First Step: Semi-Honest Protocol with Dishonest Majority

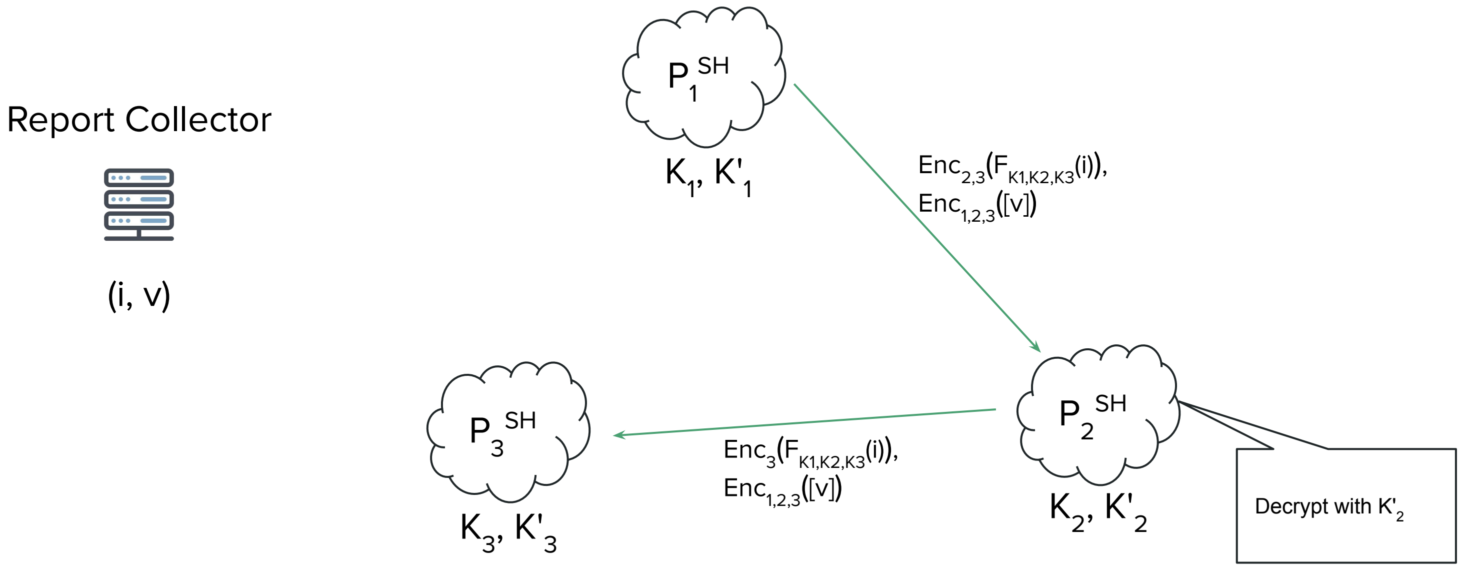


F_K: Pseudorandom Function with key K

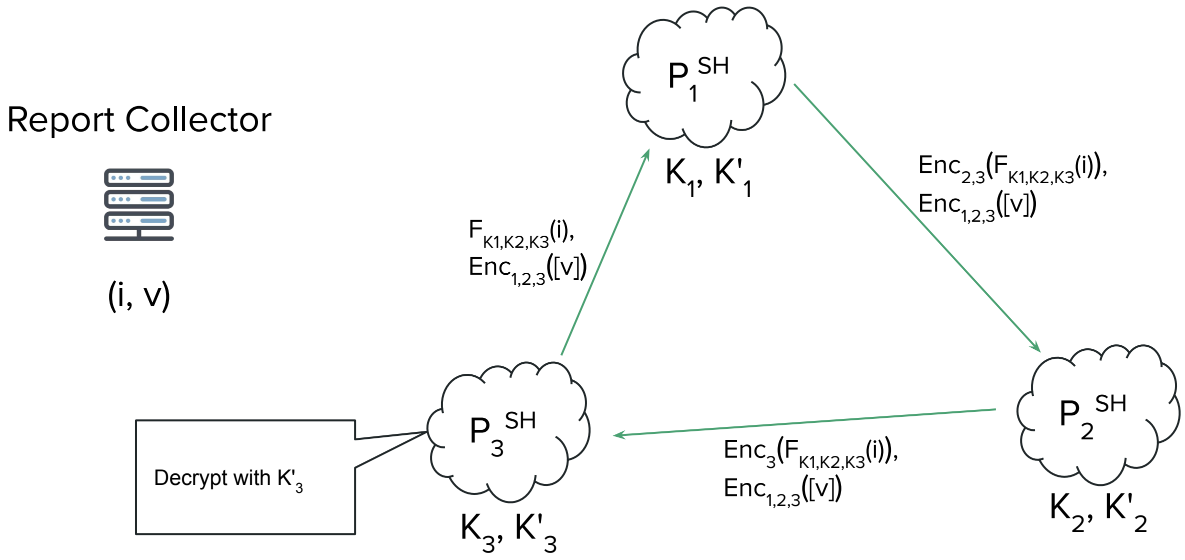
First Step: Semi-Honest Protocol with Dishonest Majority



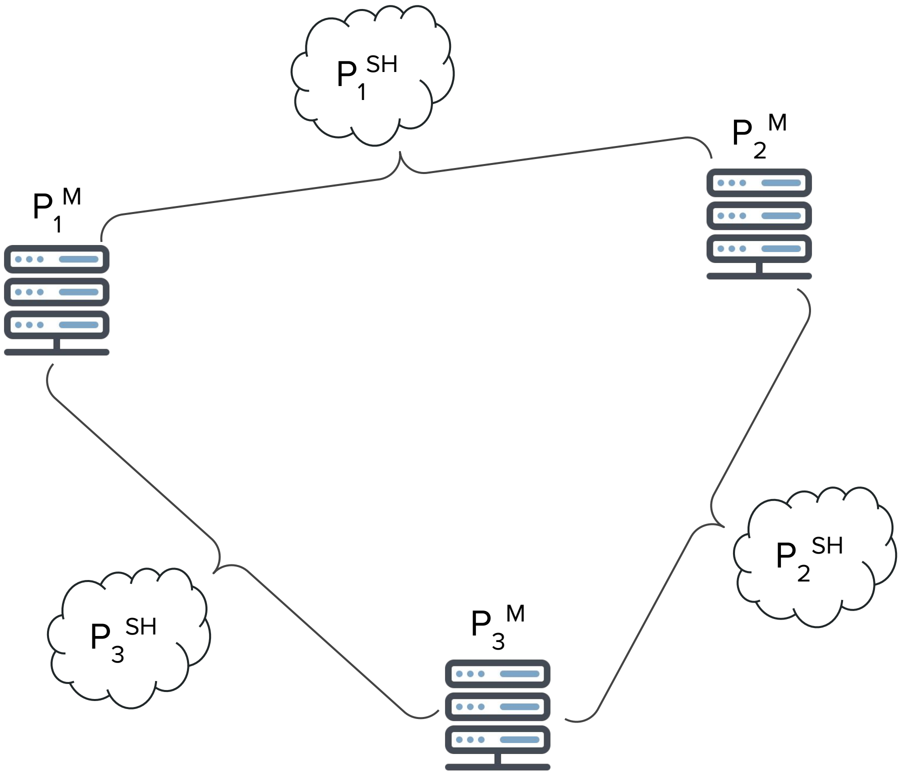
First Step: Semi-Honest Protocol with Dishonest Majority



First Step: Semi-Honest Protocol with Dishonest Majority



Compiling to Malicious Security / Honest Majority



Compiling to Malicious Security / Honest Majority

1. Both real (malicious) servers implementing a virtual (semi-honest) party:
   a. Run a malicious coin flipping protocol to sample joint randomness.
   b. Generate all messages sent by the virtual party using the joint randomness.
   c. Send messages to the real parties implementing the virtual recipient.
      Note: One of these will be one of the sender parties
2. The receiver parties each check that the messages received from the two senders match, and abort if they don't.

Implementing the Semi-Honest Protocol

Building blocks:

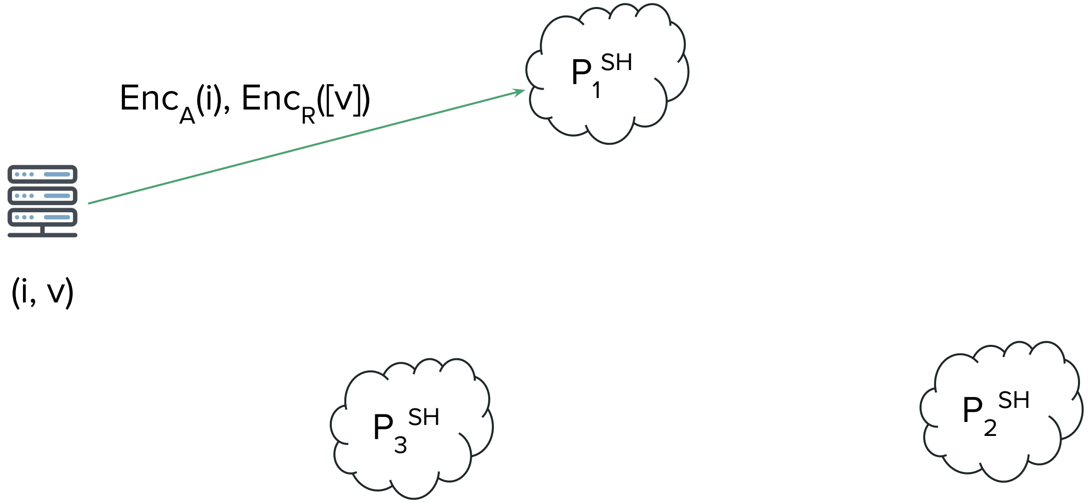
- Multiplicatively homomorphic encryption scheme Enc_M
- Additively homomorphic encryption scheme Enc_A
- Rerandomizable encryption scheme Enc_R

Secret keys for all of these are assumed secret-shared between all parties.

OPRF public key: $PK_{PRF} = Enc_A(K_{PRF})$, where K_{PRF} is again secret-shared.

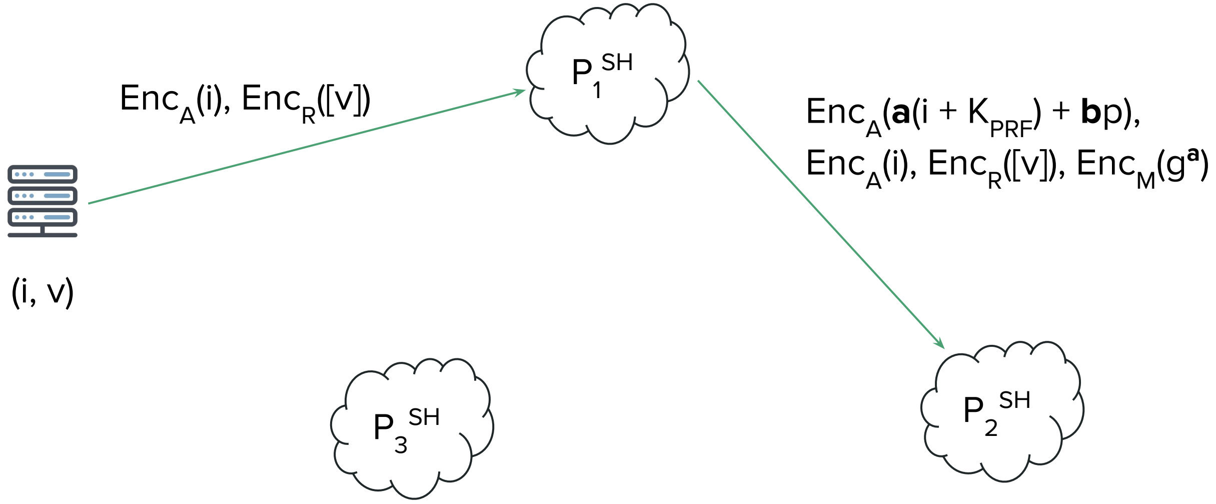
Dodis-Yampolskiy PRF: $F(i) = g^{1/(i+K_{PRF})}$

Implementing the Semi-Honest Protocol



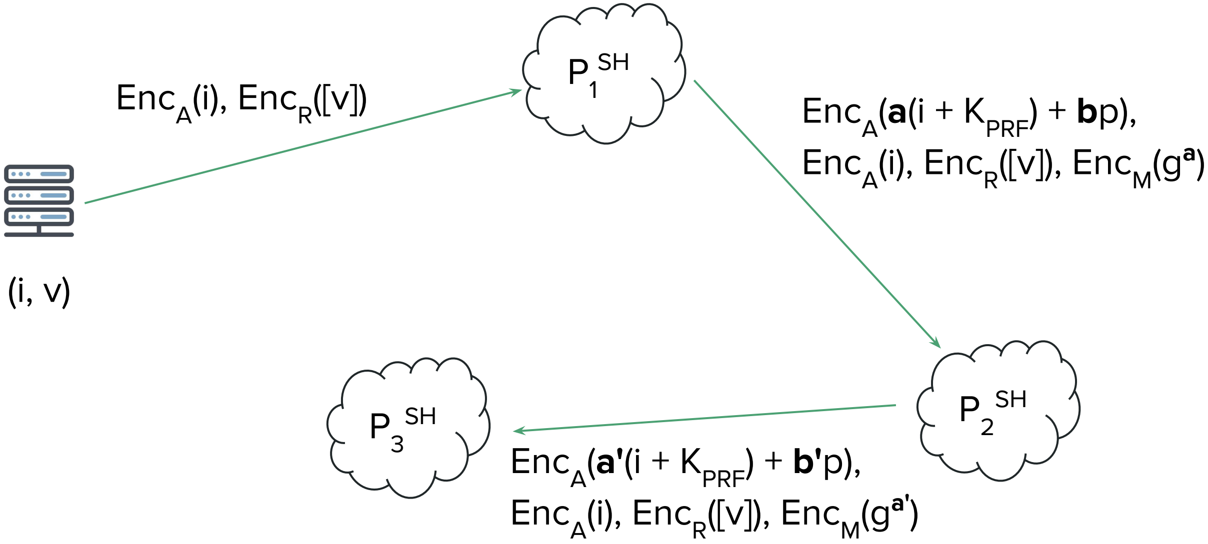
K_{PRF}: OPRF private key
p: OPRF output group size
g: Generator of OPRF output group

Implementing the Semi-Honest Protocol



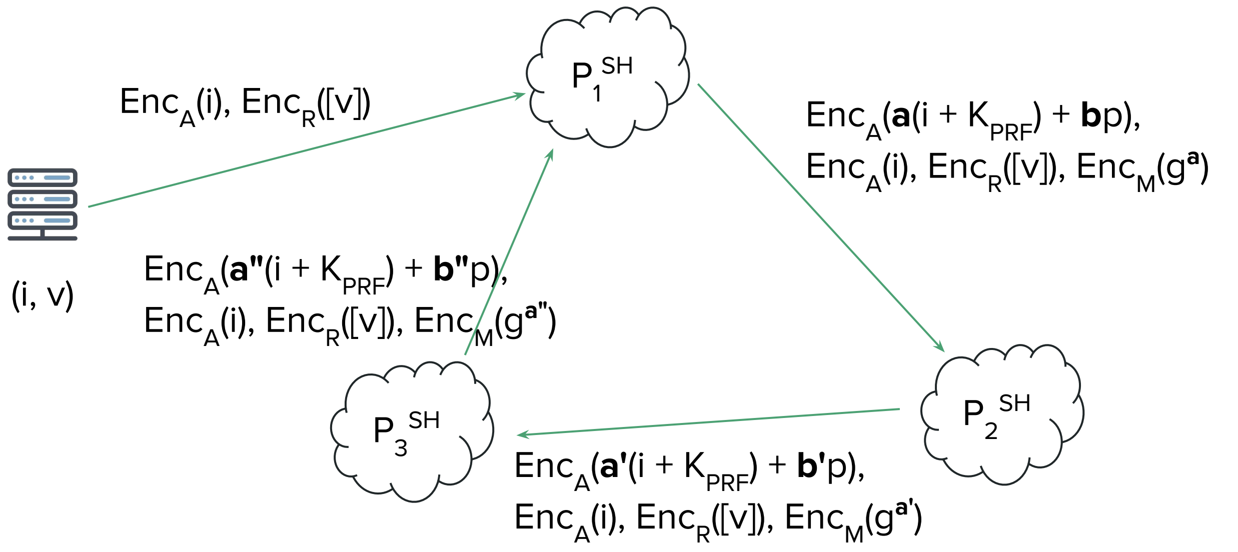
K_{PRF}: OPRF private key
p: OPRF output group size
g: Generator of OPRF output group

Implementing the Semi-Honest Protocol



K_{PRF}: OPRF private key
p: OPRF output group size
g: Generator of OPRF output group

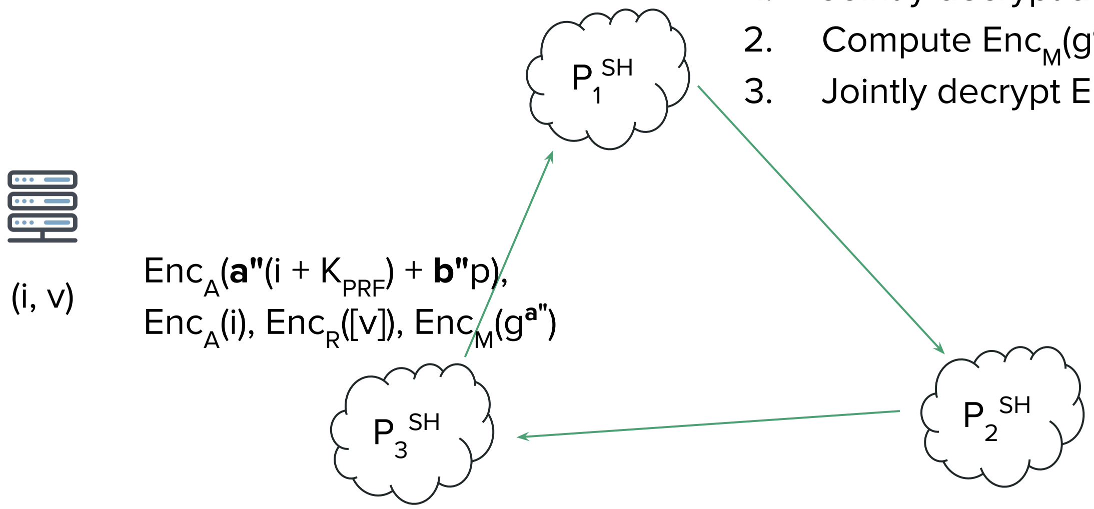
Implementing the Semi-Honest Protocol



K_{PRF}: OPRF private key
p: OPRF output group size
g: Generator of OPRF output group

Implementing the Semi-Honest Protocol

1. Jointly decrypt **a"**(i + K_{PRF}) + **b"**p
2. Compute $Enc_M(g^{\mathbf{a''} / \mathbf{a''}(i + K_{PRF}) + \mathbf{b''}p}) = Enc_M(F(i))$
3. Jointly decrypt $Enc_M(F(i))$ to obtain F(i)

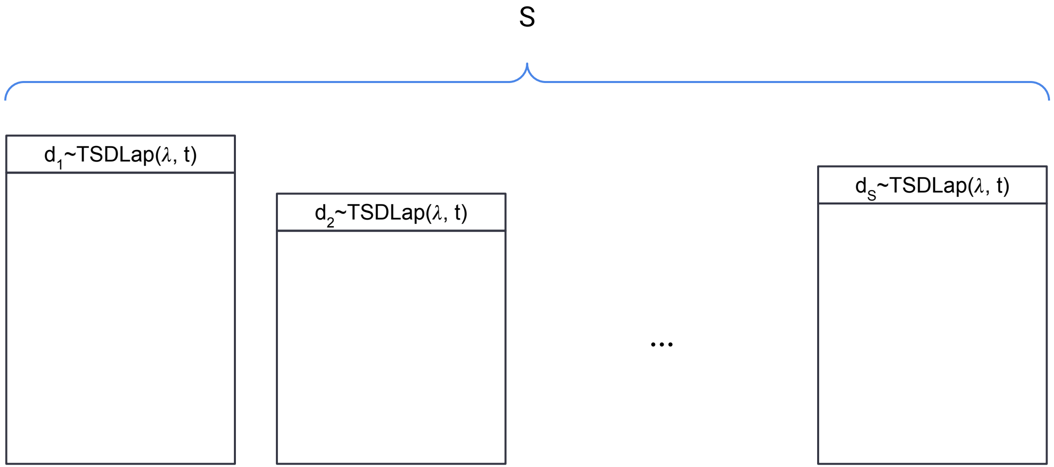


K_{PRF}: OPRF private key
p: OPRF output group size
g: Generator of OPRF output group

Dense Partitioning: Adding Dummies

M: Upper bound on the number of ciphertexts with the same index / from the same client
S: Number of shards
TSDLap(λ, t): Truncated, shifted, discrete Laplace distribution with mean t and scale λ

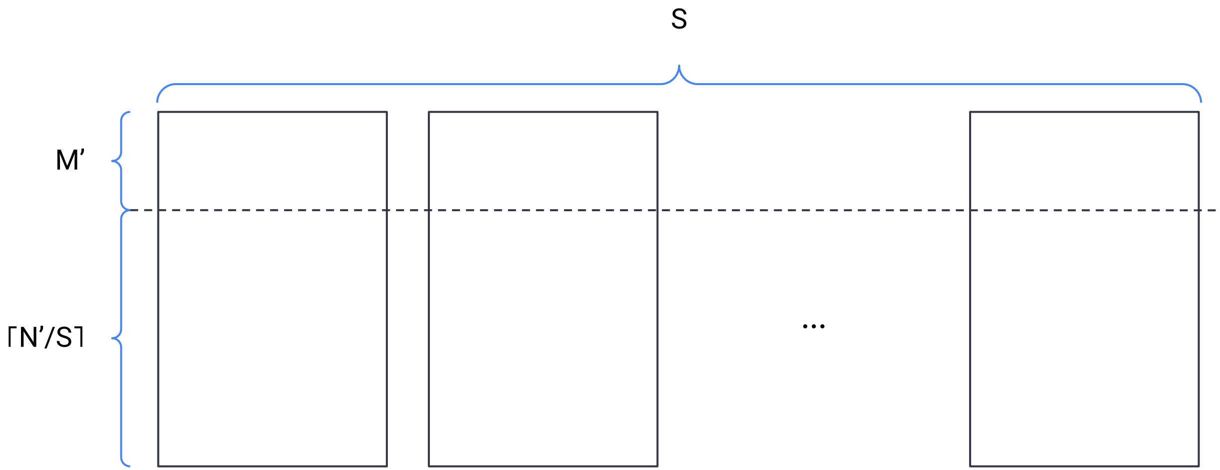
Expected #dummies per bucket for $\varepsilon = 0.5$ and $\delta = 10^{-11}$: 49 * M per server



Sparse Partitioning: Assigning Ciphertexts to Shards

N, N': Number of ciphertexts before / after adding dummies
M, M': Upper bound on the number of ciphertexts with the same index
S: Number of shards

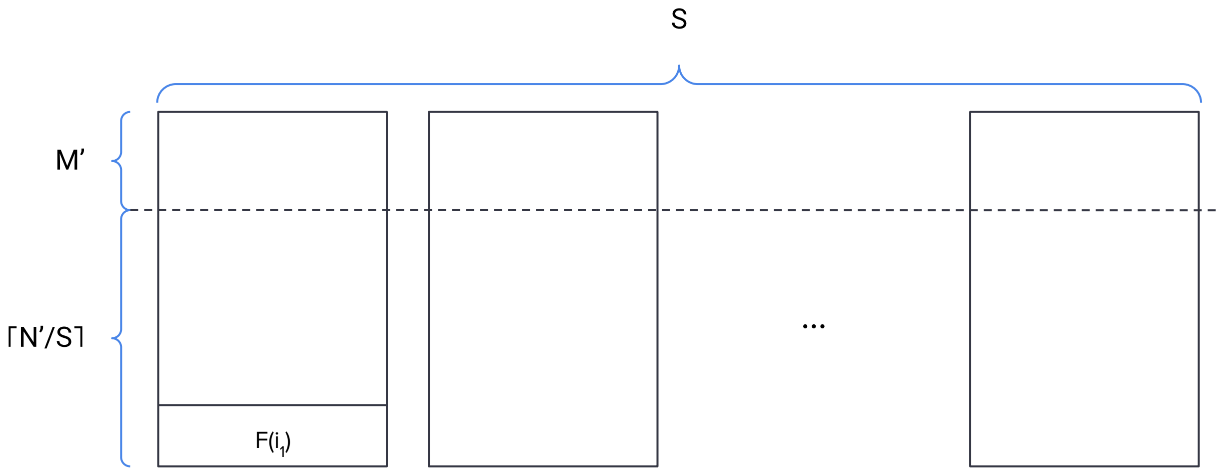
Observation: As long as $M' \ll \lceil N'/S \rceil$, the overhead will be small in practice.
But: N' might still be significantly larger than N. For $\varepsilon = 0.5$ and $\delta = 10^{-11}$, $N'/N = 1.1$



Sparse Partitioning: Assigning Ciphertexts to Shards

N, N’: Number of ciphertexts before / after adding dummies
M, M’: Upper bound on the number of ciphertexts with the same index
S: Number of shards

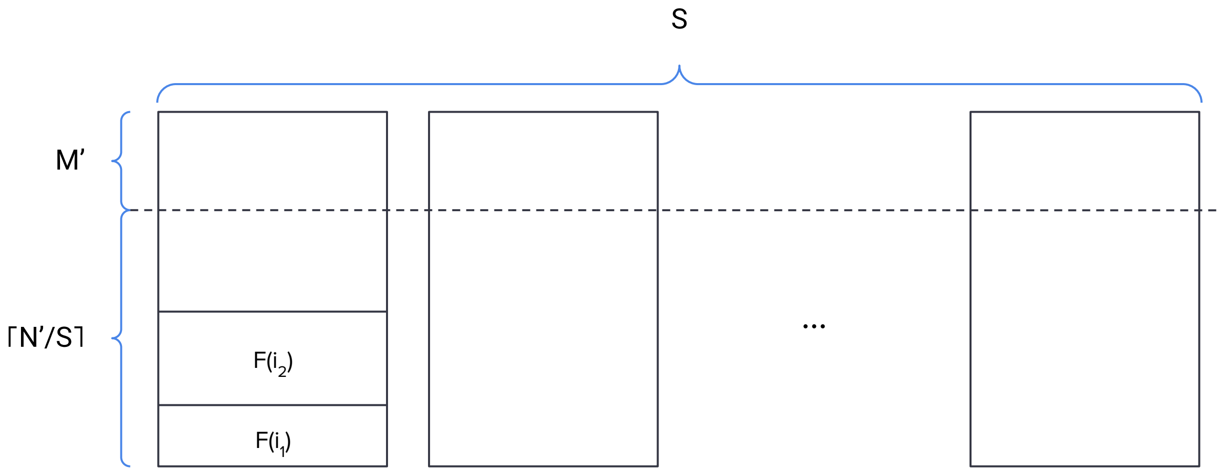
Observation: As long as $M' \ll \lceil N'/S \rceil$, the overhead will be small in practice.
But: N’ might still be significantly larger than N. For $\varepsilon = 0.5$ and $\delta = 10^{-11}$, $N'/N = 1.1$



Sparse Partitioning: Assigning Ciphertexts to Shards

N, N': Number of ciphertexts before / after adding dummies
M, M': Upper bound on the number of ciphertexts with the same index
S: Number of shards

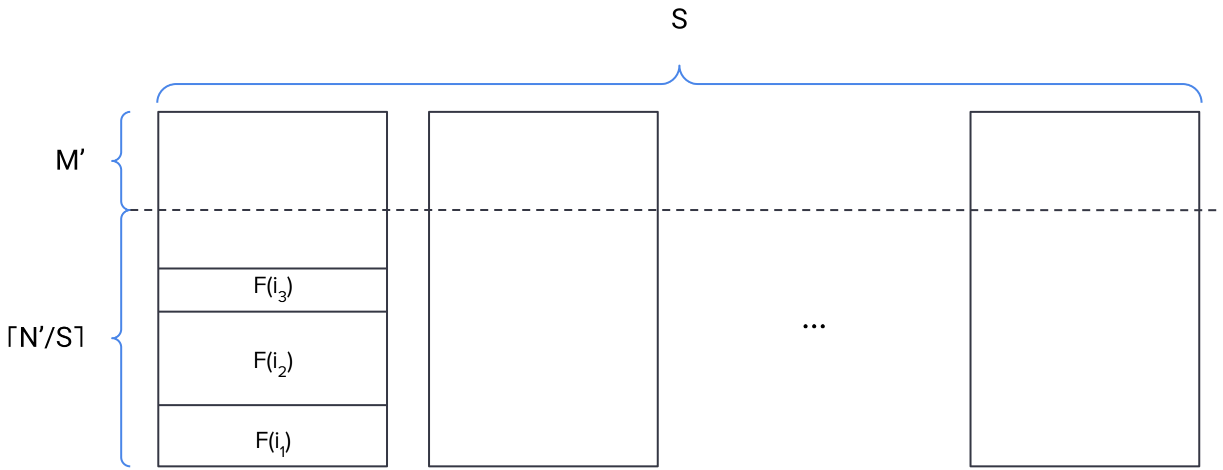
Observation: As long as $M' \ll \lceil N'/S \rceil$, the overhead will be small in practice.
But: N' might still be significantly larger than N. For $\varepsilon = 0.5$ and $\delta = 10^{-11}$, $N'/N = 1.1$



Sparse Partitioning: Assigning Ciphertexts to Shards

N, N’: Number of ciphertexts before / after adding dummies
M, M’: Upper bound on the number of ciphertexts with the same index
S: Number of shards

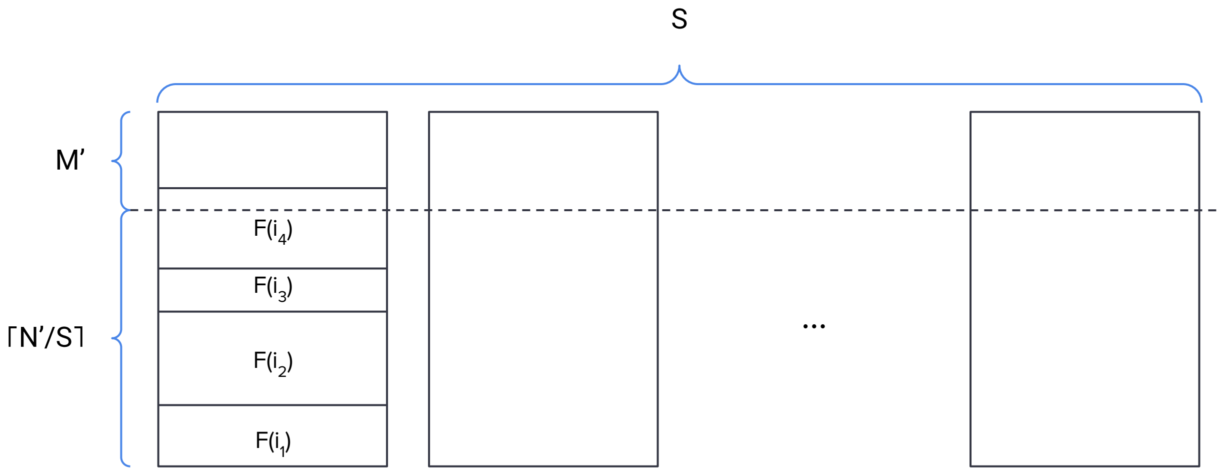
Observation: As long as $M' \ll \lceil N'/S \rceil$, the overhead will be small in practice.
But: N’ might still be significantly larger than N. For $\varepsilon = 0.5$ and $\delta = 10^{-11}$, $N'/N = 1.1$



Sparse Partitioning: Assigning Ciphertexts to Shards

N, N’: Number of ciphertexts before / after adding dummies
M, M’: Upper bound on the number of ciphertexts with the same index
S: Number of shards

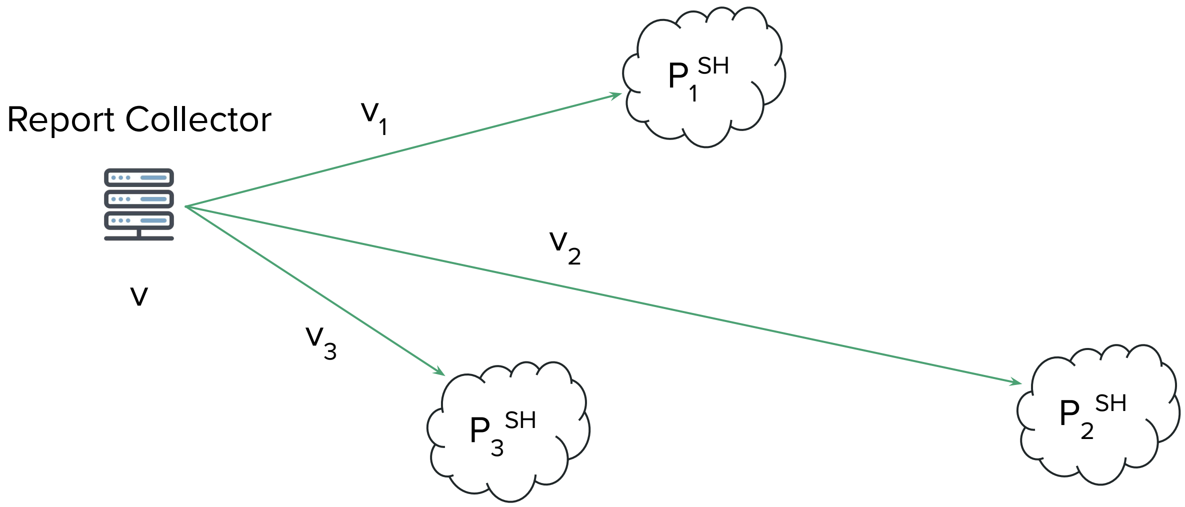
Observation: As long as $M' \ll \lceil N'/S \rceil$, the overhead will be small in practice.
But: N’ might still be significantly larger than N. For $\varepsilon = 0.5$ and $\delta = 10^{-11}$, $N'/N = 1.1$



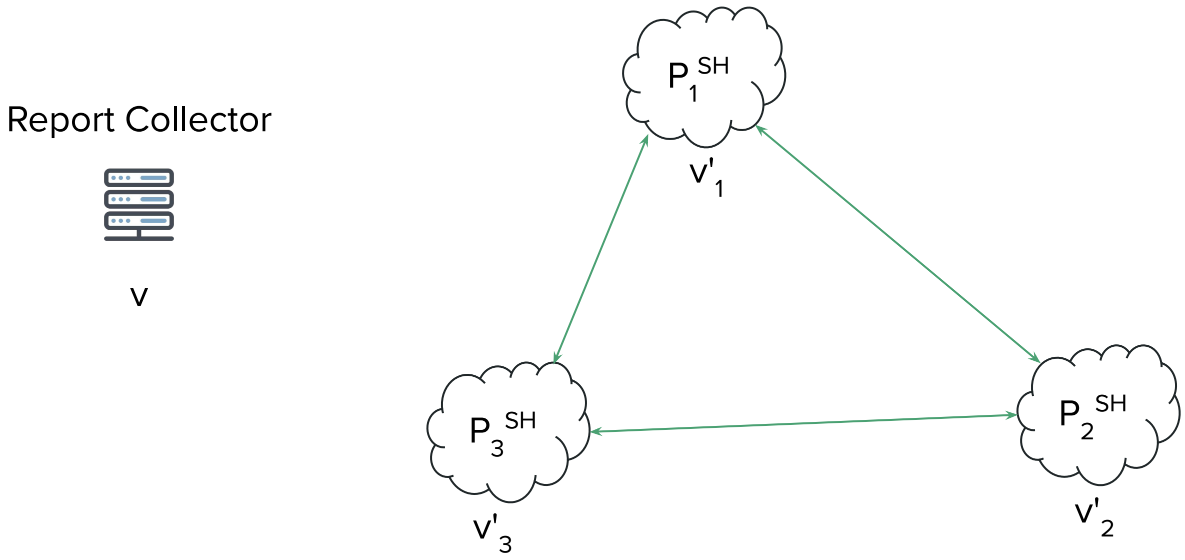
Open Questions

- Most efficient instantiations of Enc_A, Enc_M, Enc_R?
  - Possible choice: Enc_A = Carmenish-Shoup, Enc_M = ElGamal, Enc_R = ElGamal
- More efficient protocol using oblivious transfer instead of AHE?
  - Recent paper to explore: https://eprint.iacr.org/2023/602
- Time spent on partitioning vs. time spent in MPC in each partition
  - Sparse OPRF removes the need to sort by match keys in MPC
  - But may require sorting by timestamps instead (assuming these are considered private)

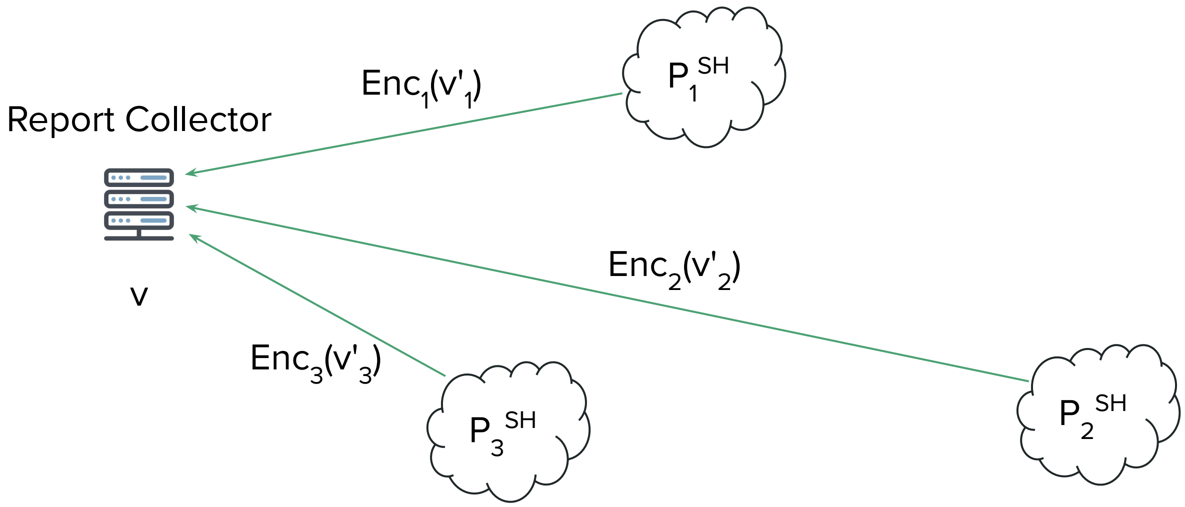
Secret-Sharing Payloads



Secret-Sharing Payloads



Secret-Sharing Payloads



How to Ensure Encrypted Matchkeys are Genuine

Client:

- $e = Enc_{AHE}(i)$: Encrypted matchkey i with randomness r
- p: Zero-knowledge proof of knowledge of i and r that encrypt to e.

P_1^{SH}:

- Verify p w.r.t. e before starting the protocol.