Source Code for Experimental Evaluation:

This source code has been adopted from https://github.com/google/rappor.

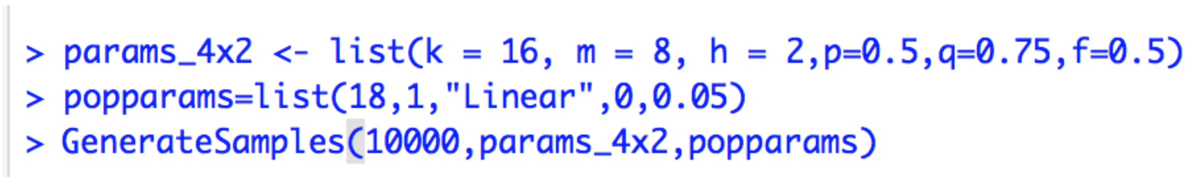
For Simulation there are specific files and functions which are used.

In the folder R/analysis, there are functions for encoding and decoding a particular client values.

Experimental Evaluation Parameters:

params_4x2 <- list(k = 16, m = 8, h = 2,p=0.5,q=0.75,f=0.5)
k -> Number of bits
m -> Number of cohorts
h -> Number of Hash Functions
p,q -> Probability values
f -> Frequency

Inputs:
Sample Size: 10000 user survey responses are generated.



```
> params_4x2 <- list(k = 16, m = 8, h = 2,p=0.5,q=0.75,f=0.5)
> popparams=list(18,1,"Linear",0,0.05)
> GenerateSamples(10000,params_4x2,popparams)
```

Fig. 1 Console inputs

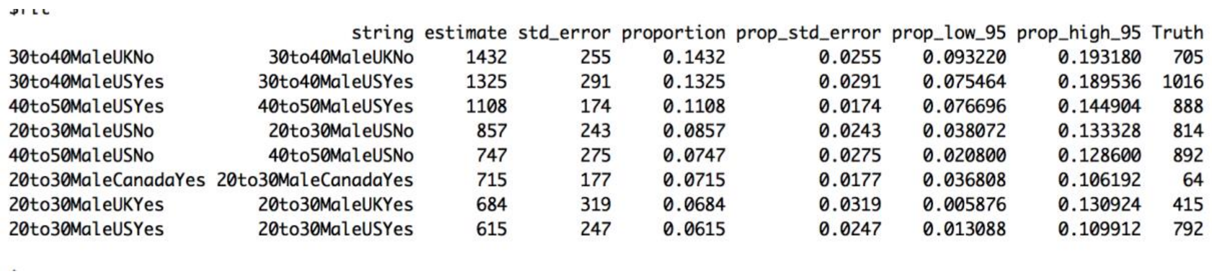
Output summary:

```
$summary
                 parameters       values
1          Candidate strings      18.0000
2           Detected strings       8.0000
3            Sample size (N)   10000.0000
4  Discovered Prop (out of N)       0.7483
5         Explained Variance       0.5040
6           Missing Variance -952625.2870
7             Noise Variance  952625.7830
8 Theoretical Noise Std. Dev.     140.3120

$privacy
```

```
          parameters     values
1         Effective p  0.5625000
2         Effective q  0.6875000
3            exp(e_1)  2.9279012
4                 e_1  1.0742859
5          exp(e_inf) 81.0000000
6               e_inf  4.3944492
7 Detection frequency  0.1100469
```

Output:



```
                                   string estimate std_error proportion prop_std_error prop_low_95 prop_high_95 Truth
30to40MaleUKNo             30to40MaleUKNo     1432       255     0.1432         0.0255    0.093220     0.193180   705
30to40MaleUSYes           30to40MaleUSYes     1325       291     0.1325         0.0291    0.075464     0.189536  1016
40to50MaleUSYes           40to50MaleUSYes     1108       174     0.1108         0.0174    0.076696     0.144904   888
20to30MaleUSNo             20to30MaleUSNo      857       243     0.0857         0.0243    0.038072     0.133328   814
40to50MaleUSNo             40to50MaleUSNo      747       275     0.0747         0.0275    0.020800     0.128600   892
20to30MaleCanadaYes 20to30MaleCanadaYes      715       177     0.0715         0.0177    0.036808     0.106192    64
20to30MaleUKYes           20to30MaleUKYes      684       319     0.0684         0.0319    0.005876     0.130924   415
20to30MaleUSYes           20to30MaleUSYes      615       247     0.0615         0.0247    0.013088     0.109912   792
```

Fig. 2 Sample size '10000'

```
$metrics
$metrics$sample_size
[1] 10000

$metrics$allocated_mass
[1] 0.7483

$metrics$num_detected
[1] 8

$metrics$explained_var
[1] 0.504

$metrics$missing_var
[1] -952625.3
```

Results Explanation:

- Input String Explanation - 30to40-> Represents the age of the client, Male/Female Represents the gender of the client, UK represents country and Yes/No represent if that user has health issues or not.
- Out of all the data sent to user, truth column says how many values of that particular string are encrypted and sent to the server.

- Estimate values are values decoded from the encrypted values that were sent to the server during encoding.
- String “30to40MaleUKNo”
    - Total number of responses sent to server -> 705 (truth values)
    - Total number of responses found -> 1432 (decoded from noise)
    - This is higher than the actual number of responses.
    - Proportion error=0.0255.
    - Comparing the total number of truth values and the estimates, it can be seen that estimates are greater than that of actual number of values.
    - The additional number of response (noise) in estimates are added to maintain privacy.

- From the above results, it can be seen that most of the values are decoded with significant accuracy. Also, privacy is preserved because actual bits are not sent to the server.
- When the sample size is 0 -100, server does not return any response.
- When the sample size is increased, server can decode the inputs and detect the survey responses sent. **($metrics$num_detected: 8)** (Fig. 2)

Source code: Simulation.r

Description:

This file has all the utility functions needed for running the simulation. We have edited the params, values and input data to run the simulation according to our dataset.

```
# Copyright 2014 Google Inc. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

#
# RAPPOR simulation library. Contains code for encoding simulated data and
#     creating the map used to encode and decode reports.

library(glmnet)
library(parallel)  # mclapply
```

#MODIFIED

```
# Line 1
params_4x2 <- list(k = 16, m = 8, h = 2,p=0.5,q=0.75,f=0.5)
```

#MODIFIED

```
# Line 2
popparams=list(18,1,"Linear",0,0.05)
```

#MODIFIED

```
# Here the input is modified to the set of Output options selected by user or
Experiment Purpose
SetOfStrings <- function(num_strings = 100) {
  # Generates a set of strings for simulation purposes.
   #strs <- paste0("V_", as.character(1:num_strings))
    strs <-
c("30to40MaleUSNo","30to40MaleUSYes","40to50MaleUSYes","40to50MaleUSNo
","20to30MaleUSNo","20to30MaleUSYes","30to40MaleUKNo","30to40MaleUKYes
","40to50MaleUKYes","40to50MaleUKNo","20to30MaleUKNo","20to30MaleUKYes
",
"30to40MaleCanadaNo","30to40MaleCanadaYes","40to50MaleCanadaYes","40to
50CanadaUKNo","20to30MaleCanadaNo","20to30MaleCanadaYes")
  strs
}

#Get sample probability (source code)
GetSampleProbs <- function(params) {
  # Generate different underlying distributions for simulations purposes.
  # Args:
  #    - params: a list describing the shape of the true distribution:
  #              c(num_strings, prop_nonzero_strings, decay_type,
  #                rate_exponetial).
  nstrs <- params[[1]]
  nonzero <- params[[2]]
  decay <- params[[3]]
  expo <- params[[4]]
  background <- params[[5]]

  probs <- rep(0, nstrs)
  ind <- floor(nstrs * nonzero)
  if (decay == "Linear") {
    probs[1:ind] <- (ind:1) / sum(1:ind)
  } else if (decay == "Constant") {
    probs[1:ind] <- 1 / ind
```

```r
 } else if (decay == "Exponential") {
    temp <- seq(0, nonzero, length.out = ind)
    temp <- exp(-temp * expo)
    temp <- temp + background
    temp <- temp / sum(temp)
    probs[1:ind] <- temp
  } else {
    stop('params[[4]] must be in c("Linear", "Exponenential",
"Constant")')
  }
  probs
}

EncodeAll <- function(x, cohorts, map, params, num_cores = 1) {
  # Encodes the ground truth into RAPPOR reports.
  #
  # Args:
  #   x: Observed strings for each report, Nx1 vector
  #   cohort: Cohort assignment for each report, Nx1 vector
  #   map: list of matrices encoding locations of hashes for each
  #       string, for each cohort
  #   params: System parameters
  #
  # Returns:
  #   RAPPOR reports for each piece of data.


  print('X value is ')
  print(x)
  p <- params$p
  q <- params$q
  f <- params$f
  k <- params$k

  qstar <- (1 - f / 2) * q + (f / 2) * p
  pstar <- (1 - f / 2) * p + (f / 2) * q

  candidates <- colnames(map[[1]])
  if (!all(x %in% candidates)) {
    stop("Some strings are not in the map. set(X) - set(candidates):
",
         paste(setdiff(unique(x), candidates), collapse=" "), "\n")
  }
  bfs <- mapply(function(x, y) y[, x], x, map[cohorts], SIMPLIFY =
FALSE,
                USE.NAMES = FALSE)
  reports <- mclapply(bfs, function(x) {
    noise <- sample(0:1, k, replace = TRUE, prob = c(1 - pstar,
pstar))
    ind <- which(x)
    noise[ind] <- sample(0:1, length(ind), replace = TRUE,
                         prob = c(1 - qstar, qstar))
    noise
  }, mc.cores = num_cores)

  reports
}

CreateMap <- function(strs, params, generate_pos = TRUE, basic =
FALSE) {
  # Creates a list of 0/1 matrices corresponding to mapping between the strs
and
  # Bloom filters for each instance of the RAPPOR.
  # Ex. for 3 strings, 2 instances, 1 hash function and Bloom filter of size
4,
  # the result could look this:
  # [[1]]
  #   1 0 0 0
  #   0 1 0 0
  #   0 0 0 1
  # [[2]]
  #   0 1 0 0
  #   0 0 0 1
  #   0 0 1 0
  #
  # Args:
  #    strs: a vector of strings
  #    params: a list of parameters in the following format:
  #         (k, h, m, p, q, f).
  #    generate_pos: Tells whether to generate an object storing the
  #        positions of the nonzeros in the matrix
  #    basic: Tells whether to use basic RAPPOR (only works if h=1).

 M <- length(strs)
  map_by_cohort <- list()
  k <- params$k
  h <- params$h
  m <- params$m

  for (i in 1:m) {
    if (basic && (h == 1) && (k == M)) {
      ones <- 1:M
    } else {
      ones <- sample(1:k, M * h, replace = TRUE)
    }
    cols <- rep(1:M, each = h)
    map_by_cohort[[i]] <- sparseMatrix(ones, cols, dims = c(k, M))
    colnames(map_by_cohort[[i]]) <- strs
  }

  all_cohorts_map <- do.call("rBind", map_by_cohort)
  if (generate_pos) {
    map_pos <- t(apply(all_cohorts_map, 2, function(x) {
      ind <- which(x == 1)
      n <- length(ind)
      if (n < h * m) {
        ind <- c(ind, rep(NA, h * m - n))
      }
      ind
    }))
  } else {
    map_pos <- NULL
  }

  list(map_by_cohort = map_by_cohort, all_cohorts_map =
all_cohorts_map,
       map_pos = map_pos)
}

GetSample <- function(N, strs, probs) {
  # Sample for the strs population with distribution probs.
  sample(strs, N, replace = TRUE, prob = probs)
}

GetTrueBits <- function(samp, map, params) {
  # Convert sample generated by GetSample() to Bloom filters where mapping
  # is defined in map.
  # Output:
  #    - reports: a matrix of size [num_instances x size] where each row
  #               represents the number of times each bit in the Bloom filter
  #               was set for a particular instance.
  # Note: reports[, 1] contains the same size for each instance.

 N <- length(samp)
  k <- params$k
  m <- params$m
  strs <- colnames(map[[1]])
  reports <- matrix(0, m, k + 1)
  inst <- sample(1:m, N, replace = TRUE)
  for (i in 1:m) {
    tab <- table(samp[inst == i])
    tab2 <- rep(0, length(strs))
    tab2[match(names(tab), strs)] <- tab
    counts <- apply(map[[i]], 1, function(x) x * tab2)
    # cat(length(tab2), dim(map[[i]]), dim(counts), "\n")
    reports[i, ] <- c(sum(tab2), apply(counts, 2, sum))
  }
  reports
}

GetNoisyBits <- function(truth, params) {
  # Applies RAPPOR to the Bloom filters.
  # Args:
  #     - truth: a matrix generated by GetTrueBits().

  k <- params$k
  p <- params$p
  q <- params$q
  f <- params$f
```

```
 rappors <- apply(truth, 1, function(x) {
    # The following samples considering 4 cases:
    # 1. Signal and we lie on the bit.
    # 2. Signal and we tell the truth.
    # 3. Noise and we lie.
    # 4. Noise and we tell the truth.

    # Lies when signal sampled from the binomial distribution.
    lied_signal <- rbinom(k, x[-1], f)

    # Remaining must be the non-lying bits when signal. Sampled with q.
    truth_signal <- x[-1] - lied_signal

    # Lies when there is no signal which happens x[1] - x[-1] times.
    lied_nosignal <- rbinom(k, x[1] - x[-1], f)

    # Trtuh when there's no signal. These are sampled with p.
    truth_nosignal <- x[1] - x[-1] - lied_nosignal

    # Total lies and sampling lies with 50/50 for either p or q.
    lied <- lied_signal + lied_nosignal
    lied_p <- rbinom(k, lied, .5)
    lied_q <- lied - lied_p

    # Generating the report where sampling of either p or q occurs.
    rbinom(k, lied_q + truth_signal, q) + rbinom(k, lied_p +
truth_nosignal, p)
  })

  cbind(truth[, 1], t(rappors))
}

GenerateSamples <- function(N = 10^5, params, pop_params, alpha = .05,
                            prop_missing = 0,
                            correction = "Bonferroni") {
 # Simulate N reports with pop_params describing the population and
  # params describing the RAPPOR configuration.
  num_strings = pop_params[[1]]

  strs <- SetOfStrings(num_strings)    # list("V1","V2","V3")
  probs <- GetSampleProbs(pop_params) # list(3,3,"Linear",0,0.05)
  samp <- GetSample(N, strs, probs)
  map <- CreateMap(strs, params)

  print('current map')
  print(map)

  truth <- GetTrueBits(samp, map$map_by_cohort, params)
  rappors <- GetNoisyBits(truth, params)
  print ('rappors is')
  print (rappors)
  strs_apprx <- strs
  map_apprx <- map$all_cohorts_map
  # Remove % of strings to simulate missing variables.
  if (prop_missing > 0) {
    ind <- which(probs > 0)
    removed <- sample(ind, ceiling(prop_missing * length(ind)))
    map_apprx <- map$all_cohorts_map[, -removed]
    strs_apprx <- strs[-removed]
  }



  # Randomize the columns.
  ind <- sample(1:length(strs_apprx), length(strs_apprx))
  map_apprx <- map_apprx[, ind]
  strs_apprx <- strs_apprx[ind]

  print('approximate map is')
  print(map)
  fit <- Decode(rappors, map_apprx, params, alpha = alpha,
                correction = correction)

  # Add truth column.
  fit$fit$Truth <- table(samp)[fit$fit$string]
  fit$fit$Truth[is.na(fit$fit$Truth)] <- 0

  fit$map <- map$map_by_cohort
  fit$truth <- truth
  fit$strs <- strs
  fit$probs <- probs

  fit
}
```

Significant functions:

```
Decode <- function(counts, map, params, alpha = 0.05,
                   correction = c("Bonferroni"), quiet = FALSE, ...) {

  print('Decode Counts is')
  print(counts)
  error_msg <- CheckDecodeInputs(counts, map, params)
  if (!is.null(error_msg)) {
    stop(error_msg)
  }

  k <- params$k
  p <- params$p
  q <- params$q
  f <- params$f
 h <- params$h
  m <- params$m

  S <- ncol(map)  # total number of candidates

  N <- sum(counts[, 1])
  if (k == 1) {
    return(.DecodeBoolean(counts, params, N))
  }

  filter_cohorts <- which(counts[, 1] != 0)  # exclude cohorts with 
zero reports

 # stretch cohorts to bits
 filter_bits <- as.vector(matrix(1:nrow(map), ncol = 
m)[,filter_cohorts, drop = FALSE])

  map_filtered <- map[filter_bits, , drop = FALSE]

  es <- EstimateBloomCounts(params, counts)

  estimates_stds_filtered <-
    list(estimates = es$estimates[filter_cohorts, , drop = FALSE],
         stds = es$stds[filter_cohorts, , drop = FALSE])

  coefs_all <- vector()

  # Run the fitting procedure several times (5 seems to be sufficient and not
  # too many) to estimate standard deviation of the output.
 for(r in 1:5) {
    if(r > 1)
      e <- Resample(estimates_stds_filtered)
    else
      e <- estimates_stds_filtered

    coefs_all <- rbind(coefs_all,
                       FitDistribution(e, map_filtered,  quiet))
  }

  coefs_ssd <- N * apply(coefs_all, 2, sd)  # compute sample standard 
deviations
  coefs_ave <- N * apply(coefs_all, 2, mean)

  # Only select coefficients more than two standard deviations from 0. May
  # inflate empirical SD of the estimates.
  reported <- which(coefs_ave > 1E-6 + 2 * coefs_ssd)

  mod <- list(coefs = coefs_ave[reported], stds = coefs_ssd[reported])

  coefs_ave_zeroed <- coefs_ave
  coefs_ave_zeroed[-reported] <- 0
  residual <- as.vector(t(estimates_stds_filtered$estimates)) -
    map_filtered %*% coefs_ave_zeroed / N

  if (correction == "Bonferroni") {
    alpha <- alpha / S
  }

  inf <- PerformInference(map_filtered[, reported, drop = FALSE],

as.vector(t(estimates_stds_filtered$estimates)),
                          N, mod, params, alpha,
                          correction)
  fit <- inf$fit
  # If this is a basic RAPPOR instance, just use the counts for the estimate
  #     (Check if the map is diagonal to tell if this is basic RAPPOR.)
 if (sum(map) == sum(diag(map))) {
    fit$Estimate <- colSums(counts)[-1]
  }

  # Estimates from the model are per instance so must be multipled by h.
  # Standard errors are also adjusted.
 fit$estimate <- floor(fit$Estimate)
  fit$proportion <- fit$estimate / N

  fit$std_error <- floor(fit$SD)
  fit$prop_std_error <- fit$std_error / N

  # 1.96 standard deviations gives 95% confidence interval.
  low_95 <- fit$proportion - 1.96 * fit$prop_std_error
  high_95 <- fit$proportion + 1.96 * fit$prop_std_error
  # Clamp estimated proportion.  pmin/max: vectorized min and max
  fit$prop_low_95 <- pmax(low_95, 0.0)
  fit$prop_high_95 <- pmin(high_95, 1.0)

  fit <- fit[, c("string", "estimate", "std_error", "proportion",
                 "prop_std_error", "prop_low_95", "prop_high_95")]

  allocated_mass <- sum(fit$proportion)
  num_detected <- nrow(fit)

  ss <- round(inf$SS, digits = 3)
  explained_var <- ss[[1]]
  missing_var <- ss[[2]]
  noise_var <- ss[[3]]

  noise_std_dev <- round(inf$resid_sigma, digits = 3)
  # Compute summary of the fit.
 parameters <-
      c("Candidate strings", "Detected strings",
        "Sample size (N)", "Discovered Prop (out of N)",
        "Explained Variance", "Missing Variance", "Noise Variance",
        "Theoretical Noise Std. Dev.")
  values <- c(S, num_detected, N, allocated_mass,
              explained_var, missing_var, noise_var, noise_std_dev)

  res_summary <- data.frame(parameters = parameters, values = values)

  privacy <- ComputePrivacyGuarantees(params, alpha, N)
  params <- data.frame(parameters =
                       c("k", "h", "m", "p", "q", "f", "N", "alpha"),
                       values = c(k, h, m, p, q, f, N, alpha))


  # This is a list of decode stats in a better format than 'summary'.
 metrics <- list(sample_size = N,
                 allocated_mass = allocated_mass,
                 num_detected = num_detected,
                 explained_var = explained_var,
                 missing_var = missing_var)

  list(fit = fit, summary = res_summary, privacy = privacy, params =
params,
       lasso = NULL, residual = as.vector(residual),
       counts = counts[, -1], resid = NULL, metrics = metrics,
       ests = es$estimates  # ests needed by Shiny rappor-sim app
  )
}


Encode <- function(value, map, strs, params, N, id = NULL,cohort =
NULL, B = NULL, BP = NULL) {
  # Encode value to RAPPOR and return a report.
  #
  # Input:
  #    value: value to be encoded
  #    map: a mapping matrix describing where each element of strs map in
  #         each cohort
  #    strs: a vector of possible values with value being one of them
  #    params: a list of RAPPOR parameters described in decode.R
  #    N: sample size
  # Optional parameters:
  #    id: user ID (smaller than N)
  #    cohort: specifies cohort number (smaller than m)
  #    B: input Bloom filter itself, in which case value is ignored
  #    BP: input Permanent Randomized Response (memoized for multiple
colections
  #        from the same user

 k <- params$k
  p <- params$p
  q <- params$q
  f <- params$f
  h <- params$h
  m <- params$m

  print('params is')
  print(params)

  print('Value is ')
  print(value)
  print('STRS IS ')
  print(strs)


  print('Map is')
  print(map)

  if (is.null(cohort)) {
    cohort <- sample(1:m, 1)
  }

  if (is.null(id)) {
    id <- sample(N, 1)
  }

  ind <- which(value == strs)
  print('Indices is ')
  print(ind)

  print('cohort is')
  print(cohort)


  if (is.null(B)) {
    B <- as.numeric(map[[cohort]][, ind])
  }
  print('B is')
  print(B)
  if (is.null(BP)) {
    BP <- sapply(B, function(x) sample(c(0, 1, x), 1,
                                       prob = c(0.5 * f, 0.5 * f, 1 -
f)))
    print('BP is')
    print(BP)
  }
  rappor <- sapply(BP, function(x) rbinom(1, 1, ifelse(x == 1, q, p)))
  print('RAPPOR VALUE IS ')
  print(rappor)

  list(value = value, rappor = rappor, B = B, BP = BP, cohort =
cohort, id = id)
}
```