data_treatment_actual

June 22, 2021

[ ]:
```
import sys# make sure all relevant libraries are installed
!{sys.executable} -m pip install pandas pyserial pyyaml
```

[5]:
```
# import all functions used for processing data
import pandas as pd
from SerialLogger import write_logs
from Analyzelogs import load_log, categorize_logfiles, calculate_all, \
 calculate_cost, extract_avg
```

```
Reading content in progress, this may take some time..
Writing log xor_256 for device Arduino Nano 33 IoT
Writing log rng_256 for device Arduino Nano 33 IoT
Done!
```

[ ]:
```
# run this cell each time you log primitive(s) to write the .log files
# from serial communication. store in test_results folder.
write_logs("test_results/")
```

[5]:
```
# Categorizing all the expermimental results on cryptographic primitives
#  for ESP32 into pandas dataframes.  stored in dicts.

root_dir = "./test_results"
esp32_dir =  "Adafruit_Feather"
esp8266_dir = "Adafruit_Huzzah"
nano_dir =  "Arduino_Nano_33_IoT"


# put all logfiles into dicts by each device, each logfile a pandas dataframe
#  with columnssample and time [microseconds]
esp32 = categorize_logfiles(f"{root_dir}/{esp32_dir}")
esp8266 = categorize_logfiles(f"{root_dir}/{esp8266_dir}")
nano_iot = categorize_logfiles(f"{root_dir}/{nano_dir}")

time_col = "time [microseconds]"
esp32_xor = esp32["xor_256"][time_col]
```

```python
print(round(esp32_xor.mean()))
```

3

[6]:

```python
# Constructing the measured primitive table using the mean execution time in␣
↪microseconds
index = ["XOR", "RNG", "BLAKE2s", "SHA256", "SHA3-256","AES256 Enc.",
 "AES256 Dec.", "AES-256-GCM", "NIST P-256 ECC+DH"]

primitives = ['xor_256', 'rng_256','blake2s','sha2_256', 'sha3_256'
,'aes256_enc','aes256_dec','aes256_gcm', 'secp256r1_ecc_dh']

# for each primitive in the list, calculate the avg execution time and put that␣
↪in a list.
# returns a column of average execution time for each device
esp32_avgs = extract_avg(esp32, primitives)
esp8266_avgs = extract_avg(esp8266, primitives)
nano_avgs = extract_avg(nano_iot, primitives)

# display values in dataframe table setting to 2 significant digit
pd.set_option('display.float_format', '{:.2E}'.format)

# create primitive execution time table, columns for each device average␣
↪execution time.
primitives_avg = pd.DataFrame({"ESP32" : esp32_avgs,
"ESP8266" : esp8266_avgs, "Nano IoT" : nano_avgs}, index=index)
print(primitives_avg)

# store in file ./tables/primitives_avg.tex
primitives_avg.to_latex('./tables/primitives_avg.tex')
```

```
                     ESP32  ESP8266  Nano IoT
XOR               3.00E-06 7.00E-06  1.10E-05
RNG               3.10E-05 9.20E-05  1.70E-04
BLAKE2s           1.90E-05 8.30E-05  1.44E-04
SHA256            2.80E-05 1.07E-04  2.61E-04
SHA3-256          1.90E-04 5.50E-04  1.34E-03
AES256 Enc.       1.50E-05 1.79E-04  3.24E-04
AES256 Dec.       9.00E-06 2.44E-04  5.25E-04
AES-256-GCM       1.53E-04 1.03E-03  2.57E-03
NIST P-256 ECC+DH 1.70E-01 9.78E-01  7.81E-01
```

[7]:

```python
#  implemented scheme average execution time

# index for dataframe/table implemented scheme average excution time
implemented_schemes_index = ["HashXOR SHA3", "NewHope", "ECIES" ]
# keys to use in dictionary to get right dataframe
```

```
implemented_schemes = ["hashxor", "newhope_client", "ecies"]

# create 3 lists containing avg execution time of lists for each device
#  in form [hashxor_avg, newhope_avg, ecies_avg]
esp32_avg_schemes = extract_avg(esp32, implemented_schemes)
esp8266_avg_schemes = extract_avg(esp8266, implemented_schemes)
nano_avg_schemes = extract_avg(nano_iot, implemented_schemes)

# assemble dataframe where primitive is index and columns are avg execution␣
 ↪time of schemes by device.
implemented_schemes_avg = pd.DataFrame({ "ESP32" : esp32_avg_schemes,
"ESP8266" : esp8266_avg_schemes, "Nano IoT" : nano_avg_schemes},
 index=implemented_schemes_index)

print(implemented_schemes_avg)

 # store processed data in file
implemented_schemes_avg.to_latex('./tables/implemented_schemes_avg.tex')
```

```
                 ESP32   ESP8266   Nano IoT
HashXOR SHA3 3.11E-04 8.56E-04  2.18E-03
NewHope      1.12E-02      NaN  7.73E-02
ECIES        1.70E-01 9.79E-01  7.84E-01
```

[13]:
```
# theoretical scheme average execution time
# khan: rand, 3 xor, 4 hash, 1 sc.dc
# braeken: 7 hash
# SEL-AKA: rand, 3 hash, sc.en
# ecies: 1 hash, 1 ecc+dh, 1 aes256/gcm
# hashxor: 2 rng, 3 xor, 2 distinct hashing functions

# index to use in assembled dataframe of theoretical scheme avg execution time
scheme_index = ['Khan', "Braeken", "SEL-AKA",  "HashXOR BlAKE2s",
 "HashXOR SHA3", "ECIES"]

# primitives and amounts they're used for each scheme, used to calculate scheme
#  cost by table lookup in avg
# primitive execution time table
braeken_prims = [["SHA256", 7]]
sel_aka_prims = [["RNG", 1], ["SHA256",3], ["AES256 Enc.", 1]]
khan_prims = [["RNG", 1], ["XOR", 3], ["AES256 Dec.", 1], ["SHA256", 4]]
hashxor_2 = [["RNG", 2], ["XOR", 3], ["SHA256", 1], ["BLAKE2s", 1]]
ecies_prims = [["SHA256", 1], ["NIST P-256 ECC+DH", 1], ["AES-256-GCM", 1]]
hashxor_prims = [["RNG", 2], ["XOR",3], ["SHA256", 1], ["SHA3-256", 1]]



# scheme costs put in a list so they can be iterated through,
```

```
# same order as index so they're matching
schemes = [khan_prims, braeken_prims, sel_aka_prims,
  hashxor_2, hashxor_prims, ecies_prims ]

# use scheme costs and goes through all of them, returns
#  list of avg theoretical execution time for the schemes in the same order as␣
 ↪index.
esp32_schemes = calculate_all(primitives_avg, schemes, "ESP32")
esp8266_schemes = calculate_all(primitives_avg, schemes, "ESP8266")
nano_schemes = calculate_all(primitives_avg, schemes, "Nano IoT")


# assemble dataframe/table by using columns as costs
#  for each device, and scheme names as index.
schemes_theoretical = pd.DataFrame({"ESP32" : esp32_schemes,
 "ESP8266" : esp8266_schemes, "Nano IoT" : nano_schemes}, index=scheme_index)
schemes_theoretical.to_latex("./tables/theoretical_schemes_avg.tex")
print(schemes_theoretical)
```

```
                  ESP32  ESP8266  Nano IoT
Khan           1.61E-04 7.85E-04  1.77E-03
Braeken        1.96E-04 7.49E-04  1.83E-03
SEL-AKA        1.30E-04 5.92E-04  1.28E-03
HashXOR BlAKE2s 1.18E-04 3.95E-04  7.78E-04
HashXOR SHA3   2.89E-04 8.62E-04  1.98E-03
ECIES          1.70E-01 9.79E-01  7.84E-01
```

[73]:
```
# calculate percentage error between theoretical schemes and implemented schemes

def percentage_difference(a,b):
    delta = abs(a - b);
    return round((delta/(a+b))*100,2)

def percentage_error(a, b):
    delta = abs(a - b)
    return round((delta/abs(b))*100,2)

def percentage_change(a,b):
    delta = a -b
    return round((delta/abs(b))*100, 2)

# look at error differences between scheme theoretical and implemented



theoretical = schemes_theoretical.at["ECIES", "ESP32"]*1e6
implemented = implemented_schemes_avg.at["ECIES", "ESP32"]*1e6
```

```python
print("theoretical: ",theoretical)
print("implemented: ",implemented)
error = percentage_error(theoretical, implemented)
difference = percentage_difference(theoretical, implemented);
change = percentage_change(implemented, theoretical)
print(f"error: {error}%\ndifference: {difference}%\nchange: {change}%")
```

```
theoretical:  170006.0
implemented:  170049.0
error: 0.03%
difference: 0.01%
change: 0.03%
```

[ ]:

```python

```