DPA Contest Tutorial

1 Introduction

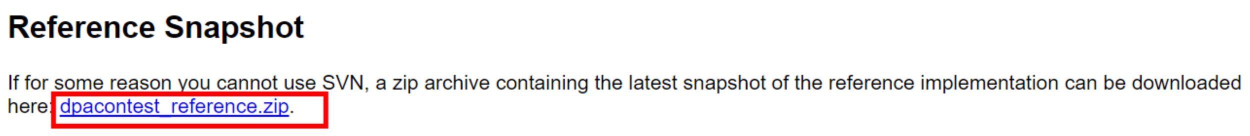
In this lab we implement a differential power analysis (DPA) attack based on the reference model provided on the DPA contest website and then use this tool to analyze the power traces used for the v1 of DPA contest. Below, you will find the step by step guide to perform the reference attack and some information regarding the improvements you could make to the code. In this document we assume that you have the following resources available on your machine. Of course you could perform this attack on any operating system or any version of Python, but you would have to find the appropriate substitute os-specific codes yourself and you might need to make adjustments to the python codes as the syntax of python changes often.

- Microsoft Windows 7 or newer
- Python 2.7 and default packages
- 7GB of available disk space or more
- 1GB of available memory or more

2 Reference Attack

First, we need to download the attack and power traces archives and extract them on our local disk.

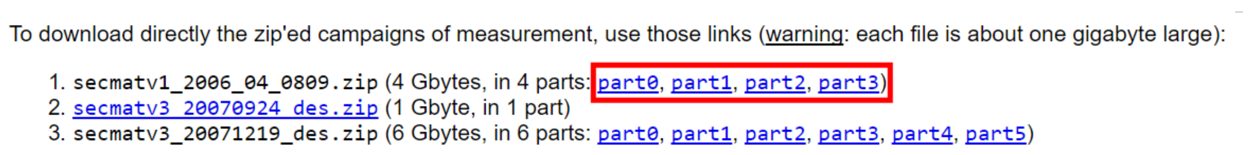
1. In your `C:` directory create a folder called `DPAcontest`. This is will be our working directory.
2. Download the reference attack from the bottom of this page and extract its contents into the `C:\DPAcontest\reference\` directory.



Reference Snapshot

If for some reason you cannot use SVN, a zip archive containing the latest snapshot of the reference implementation can be downloaded here: dpacontest_reference.zip.

3. Download the power traces labeled `secmatv1_2006_04_0809.zip` from the bottom of this page. This archive is split into four pieces; you need to download all of them and copy them to `C:\DPAcontest\`.



To download directly the zip'ed campaigns of measurement, use those links (warning: each file is about one gigabyte large):

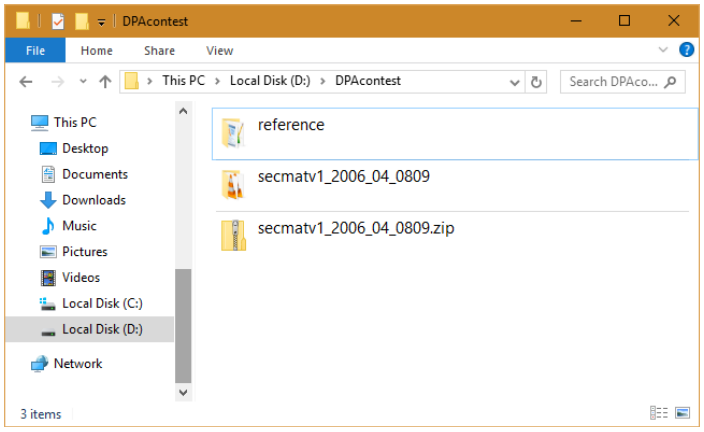
1. secmatv1_2006_04_0809.zip (4 Gbytes, in 4 parts: part0, part1, part2, part3)
2. secmatv3_20070924_des.zip (1 Gbyte, in 1 part)
3. secmatv3_20071219_des.zip (6 Gbytes, in 6 parts: part0, part1, part2, part3, part4, part5)

4. You now need to concatenate the four parts of the `secmatv1_2006_04_0809.zip` to create the complete zip file. Open Command Prompt from the start menu and type the following commands

   - `cd C:\DPAcontest\`
   - `copy /b secmatv1_2006_04_0809.zip.part* merged.zip`

5. Now unzip `merged.zip` in the `C:\DPAcontest\` directory. This should create a folder called `secmatv1_2006_04_0809` which contains all the binary trace files.

At this point we have the reference attack and the raw trace files that we need ready. Your folder contents should look like this:



Now if you try to run the reference attack, you will notice that it exits with a message indicating that you need the "PostgreSQL" package for python to be able to connect to the online database and retrieve the trace files. However, this online database has been shut down and we need to modify the code to use the file system instead. So instead, we change the code to load the trace files from the disk.

To minimize our modifications to the code, we need to keep the interface of the new module, which will replace `traces_database.py`, similar to the existing one. Let's take the following steps to prepare for replacing `traces_database.py`.

6. Create a file in `C:\DPAcontest\reference\` and name it `trace_loader.py`
7. Now copy and paste the contents of `traces_database.py` into `trace_loader.py`. We will use this as our starting code and will modify it.
8. Delete the contents of `__init__` function and implement an algorithm that does the following.
   - Take the folder name of where the traces are as argument
   - Save this folder name in a class variable for future use.
9. Delete the contents of `get_trace` function and implement an algorithm that does the following.
   - List all files in the folder (not the directories)
   - For all the files in that directory:
     - Extract message and cryptogram from the file name. We won't need the key.
     - Open the file as **binary** and read its content in
     - Parse the content using `prase_binary` function.
     - Close the file
     - Yield the next (message, cryptogram, parsed file content)
   - Note that Yielding turns the function to a generator, so we need to update the codes which use it. More on python generators here.
10. Delete the contents of test function and implement an algorithm that tests your implementation.

You will find the solution implementation in Appendix I (trace_loader.py for section 2) along with helpful comments. To try it out, simply copy the code from Appendix I (trace_loader.py for section 2) into `trace_loader.py`.

Now that we have implemented a new module to load the traces from our local disk, we need to modify the codes that used the old `traces_database` module to use our module instead. One thing to notice is that the original `get_trace()` was a simple function call, while our new `get_trace()` is a generator, so we need to make the calls to the object with `.next()` like this:

```
traces = tdb.get_trace()
msg, crypt, trace= traces.next()
```

Although there are many approaches to this, we use a generator approach so that `get_trace()` remembers its internal states and creates the list of the files in the folder, which is a slow process, only once.

Affected files to be updated are `key_estimator.py`, `sbox_breaker.py` and `main.py`. Let's update them.

11. Open `key_estimator.py` in your favorite text editor.
12. Replace `import traces_database` with `from trace_loader import traces_database`.
13. Update the `test()` function to look like this

```
def test():
   ke= key_estimator(0, 56) # 56 is the correct key for the sbox #0
   tdb= traces_database.traces_database(__TABLE__)
   for i in range(10):
      msg, crypt, trace= tdb.get_trace()
      ke.process(msg, trace)
      print "processed trace:", i, "- mark is:", ke.get_mark()

   fd= open("output.csv", "w")
   for f in ke.get_differential():
      fd.write(str(f)+"\n")
   fd.close()
```

14. Open `sbox_breaker.py` in your favorite text editor.
15. Replace `import traces_database` with `from trace_loader import traces_database`.
16. Update the `test()` function to look like this

```
def test():
   sb= sbox_breaker( 1 )
   tdb= traces_database(__TABLE__)
   traces = tdb.get_trace()
   for i in range(10):
      msg, crypt, trace= traces.next()
      sb.process(msg, trace)
      best_key= sb.get_key()
      print "processed trace:", i, "- best key is:", best_key
```

17. Finally, open `main.py` in your favorite text editor
18. Replace `import traces_database` with `from trace_loader import traces_database`.
19. Replace the lines 49 and 50 (below)

```
while fullkey == None:
   msg, crypt, trace= tdb.get_trace()
```

with these three lines:

```
traces = tdb.get_trace()
while fullkey == None:
   msg, crypt, trace = traces.next()
```

Now we are done with updating the code. The reference code is ready to read the files from the local disk and interpret them correctly, and the rest of the code is updated to use our new module instead. To run the attack, follow these steps:

20. Open command prompt from the start menu.
21. Type `cd C:\DPAcontest\reference\`
22. Type `python main.py`

This will compile and execute the python code. Notice that depending on your machine, the reference code might take a few thousand iterations and from hours to a day to complete. At the end you should get a prompt that looks like this and shows the that you obtained the correct code. (the "`L`" at the end means long number in python and is not a part of the key.)

```
# Key: 0x6a64786a64786a64L
```

3 Possible Improvements

This is where you can get creative. In the "Hall of Fame" section you can find the fully working implementation of an optimized code based on the reference attack that extracts the key in a few minutes (as opposed to many hours).

Here are a few suggestions of the general areas that could be optimized and a few suggestions.

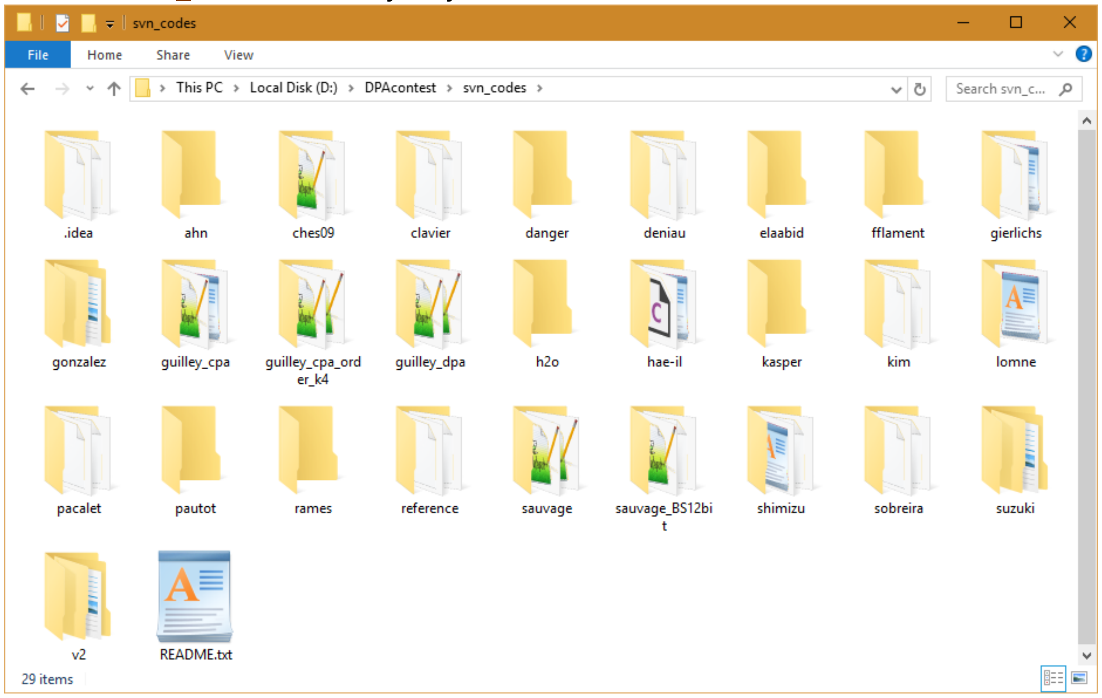
- Trace Loader
  - Trace loader can preprocess the traces and load the most different ones, based on hamming distance, or load the files at random. Both of these will improve efficiency.
- Analysis algorithm
  - There are lots of documentations on the DPA contest webpage regarding different attack algorithms which improved performance significantly. For instance, you may choose to use `kurtosis` or `RMS` digital signal processing (DSP) libraries to improve performance. You can also implement your own algorithm and test its performance.
- Trace windows size
  - You may choose to analyze only the power consumption of the first round of the DES and ignore the rest of it. This will make your code run faster. To be exact, you can look at indices [5700, 5900] for the first round power consumption. (adjusted for the `secmatv1_2006_04_0809.zip` dataset.
- Using optimized math modules
  - `numpy` is the most famous math library in python which has its core code implemented in `C`. This library can perform many times faster than native python and increase your performance. You can use this, or other libraries.

Remember that there are many other improvements that could be made and you are not limited to the ones mentioned here. These are just a few suggestions to get you started. You can also see the implement ion of the ones mentioned above at the Hall of Fame.

4 Hall of Fame

On the Hall of Fame section of the DPA contest website, there are mentions of the working submitted algorithms and their performance. In this section we go over how to download them from the SVN repository and specifically how to use the modify and use the latest version of the reference code that was described in the CHES09 conference.

1. Download any SVN client. We suggest using a portable client because we will use it only once to download the content of the repository. We used SmartSVN.
2. Use the client to clone the repository at https://svn.comelec.enst.fr/dpacontest/code/ at `C:\DPAcontest\svn_codes\` directory on your machine. The svn_codes should look like this:



3. Make sure the traces are at the `C:\DPAcontest\secmatv1_2006_04_0809` on your machine.
4. Navigate to `C:\DPAcontest\svn_codes\ches09\` directory on your machine
5. Open `constants.py` do the following (final version at "Appendix II (constants.py for section 4)"):
   a. Replace `DIRECTORY = "/traces/secmatv1_2006_04_0809"` with `DIRECTORY = "../../secmatv1_2006_04_0809"`
   b. Uncomment the line `BEGIN_N, END_N = None, None # All files` and comment out the next line
   c. Uncomment the line `N = 1` and comment out the next line
6. Open `traces_database.py` and comment out line 41 that says `sys.exit(1)`

We are now ready to run the code.

7. Open commend prompt from start menu.
8. Type `cd c:\DPAcontest\svn_codes\ches09`
9. Type `python main.py`

If everything is done correctly, within a few minutes you should get the result and correct key. That is, you should see this:

```
# Key: 0x6a64786a64786a64L
```

You can look at this code for some optimizations and comments on it that work correctly and are efficiently implemented.

5 APPENDIX I (TRACE_LOADER.PY FOR SECTION 2)

```
# Function allowing conversion from C binary data
from struct import unpack
import os


def parse_binary( raw_data ):
   """
   Takes a raw binary string containing data from our oscilloscope.
   Returns the corresponding float vector.
   """
   ins =  4   # Size of int stored if the raw binary string
   cur =  0   # Cursor walking in the string and getting data
   cur += 12  # Skipping the global raw binary string header
   whs =  unpack("i", raw_data[cur:cur+ins])[0] # Storing size of the waveform header
   cur += whs # Skipping the waveform header
   dhs =  unpack("i", raw_data[cur:cur+ins])[0] # Storing size of the data header
   cur += dhs # Skipping the data header
   bfs =  unpack("i", raw_data[cur-ins:cur])[0] # Storing the data size
   sc  =  bfs/ins # Samples Count - How much samples compose the wave
   dat =  unpack("f"*sc, raw_data[cur:cur+bfs])
   return dat

class traces_database:
   """ Class providing database IOs """
   __folder_name = None

   def __init__(self, folder_name):
      """ No arguments needed """
      self.__folder_name = folder_name;

   def get_trace(self):
      """
      Do not take any argument.
      Returns the next triplet (message, cipher, trace), where:
       - message is an ascii string containing a 64 bits clear message in hexadecimal,
       - cipher is an ascii string containing a 64 bits ciphered message in hexadecimal,
       - trace is a float vector containing a trace during the cipher operation.
      """
      # List all files in the folder containing traces
      files = [f for f in os.listdir('../' + self.__folder_name) if not os.path.isdir(f)]

      for f in files:
         name, info = f.split('__') # split the file name into name, and info
         key, msg, crypt = info.split('_') # split the file name at the "_" mark
         key = key[2:] # remove "k=" from the key portion of the name
         msg = msg[2:] # remove "m=" from the message portion of the name
         crypt = crypt[2:-4] # remove "c=" and ".bin" from the crypt portion of the name
         raw_data = None

         full_path = '../' + self.__folder_name + '/' + f
         with open(full_path, 'rb') as file_content: # the "rb" flag is crucial (read binary)
            raw_data = file_content.read()  # read contents of the binary file

         # This creates a generator from our function so that we perform the file listing step only once.
         # Calling the get_trace() only executes one loop of the for loop every time.
         yield msg, crypt, parse_binary(str(raw_data))

def test():
   tdb = traces_database("secmatv1_2006_04_0809")

   traces = tdb.get_trace()
   for i in range(10):
      msg, crypt, data = traces.next()
      print ("msg=%s c=%s") % (msg, crypt)


if __name__ == "__main__":
   test()
```

6 Appendix II (constants.py for section 4)

```
# constants.py customizes the side-channel analysis python application.
# Copyright (C) 2008 Florent Flament (florent.flament@telecom-paristech.fr)
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, version 3 of the License.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

# ----------------
# Files management
# ----------------

# Traces access
#DB_or_FS  = True  # From the DataBase
DB_or_FS  = False # From the filesystem (False)

# Traces on which we'll lead our DPA
TABLE     = "secmatv1_2006_04_0809" # For traces from the DataBase
#DIRECTORY = "/home/traces/secmatv1_2006_04_0809" # For traces from the filesystem on attack
DIRECTORY = "/traces/secmatv1_2006_04_0809" # For traces from the filesystem on attack2
DIRECTORY = "../../secmatv1_2006_04_0809" # For traces from the filesystem on attack2
# DIRECTORY = "../../secmatv3_20070924_des" # For traces from the filesystem on attack2

# Files order
#ORDER = "none"
ORDER = "randomized"

# Number of traces to consider
BEGIN_N, END_N = None, None # All files
#BEGIN_N, END_N = 1000, 2000 # 1000 traces, from the 1000th to the 2000th (after sort)

# ---------------------
# Traces pre-processing
# ---------------------

# Trace Zoom in
# BEGIN_T, END_T = None, None # Full trce

BEGIN_T, END_T = 5700, 5900 # Zoom in on power consumption of first round
                            # @note Entire round1 clokc cycle is in [5500:6200]

# DSP
#DSP="none"
#DSP="kurtosis"
DSP="RMS"

# --------------------
# Algorithm parameters
# --------------------

#Type of analysis
PPA="dpa"
#PPA="cpa"

# List of SBoxes to break
SB_LST = [0,1,2,3,4,5,6,7] # All SBoxes
#SB_LST = [0]

# The DES substitution box output targeted bit
```

```
#BITS = [0] # Monobit with bit 0 of each SBox
BITS = [0,1,2,3] # All bits of each SBox
#BITS = [0,2] # You can test some strange combinaisons

# ----------------
# Results analysis
# ----------------

# Number of analyses to perform:
#    - =1: single analysis
#    - >1: multiple analyses (representative order)
N = 1
# N = 1000

# The number of iterations during which the key must have been stable
STABILITY_THRESHOLD = 100

# The number of traces we have to compute before doing any test
ITERATION_THRESHOLD = 1

# Subkeys of round1 when master key is "keykeyke"
SK_R1 = (56,11,59,38,0,13,25,55)

# Output directory where to store results files
def outdir():
   OUTDIR= PPA
   if len(SB_LST) == 1:
      OUTDIR += "_sbox" + str( SB_LST[0] )
   if len(BITS) == 1:
      OUTDIR += "_bit" + str( BITS[0] )
   if DSP != "none":
      OUTDIR += "_" + DSP
   if BEGIN_T:
      OUTDIR += "_" + str(BEGIN_T) + '-' + str(END_T)
   if ORDER != "none":
      OUTDIR += "_" + ORDER
   return OUTDIR
```