Hardware Security

Implementation of Differential Power Attack (DPA)

Fereshteh Baradaran

24th Azar, 1402

Abstract

This report details the implementation of a Differential Power Attack (DPA) using Python. DPA is a form of side-channel attack that exploits variations in power consumption during cryptographic operations to extract secret keys. Unlike direct attacks, DPA leverages physical leakages from devices, making them more efficient against certain cryptographic systems.

Introduction

Differential Power Analysis (DPA) is a sophisticated method in cryptanalysis that relies on analyzing power consumption patterns of cryptographic devices to deduce secret keys. This report describes an implementation of DPA targeting an AES encryption system, focusing on the power consumption during the Substitution Box (S-Box) operation.

Implementation

Software and Tools Used:

- **Python** programming language
- **NumPy** library for numerical computations

The complete source code of this assignment is available in a GitHub repository that can be accessed at the following link.

- calculateSboxOutput.py:
  - S-Box Array:

    Purpose: The S-Box (Substitution box) is a core component in many symmetric key algorithms like AES (Advanced Encryption Standard), used for introducing non-linearity in cryptographic operations.

    Implementation: Contains a predefined array representing the S-Box. This array is utilized to simulate the substitution step of the cryptographic algorithm under analysis.

- calculateSboxOutput(plainText, key):

  Purpose: Compute the output of the cryptographic S-Box given a specific input.

  Implementation: Returns sbox(plainText[i] ⊕ key[i])

- getInput.py:
  - loadTrace(fname, trlen, start, length, n):

    Purpose: Loads power consumption traces from a binary file, from the start byte.

    Implementation: Reads the binary file and extracts the specified number of traces, each of a defined size. It processes the raw binary data into a list of byte values.

  - loadData(fileName):

    Purpose: Loads additional cryptographic data, which might include plaintexts or ciphertexts.

    Implementation: Parses a file containing cryptographic data and converting the contents into a 2D array of integers.

- Main.py:

Loading plaintext and corresponding power trace data.

Iterating through each byte of the key.

For each key byte, iterating through all possible values (0 to 255).

Separating traces into two groups based on whether the output of the S-Box operation with a guessed key results in a specific bit being set.

Computing the mean of these groups and analyzing the differential to pinpoint the correct key byte.

The **start** and **end** arrays are used to define specific segments of the traces for analysis. Each entry in these arrays corresponds to a byte of the key. The values 20 * 1000, 25 * 1000, etc., are chosen based on observations.

bitNum Array is used to select a specific bit in the output of the S-Box operation for each byte of the key.

Results:

The implementation successfully revealed the AES secret key using DPA.

Original Key: 0x0 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xaa 0xbb 0xcc 0xdd 0xee 0xff

Recovered Key: 0x0 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88 0x99 0xaa 0xbb 0xcc 0xda 0xee 0xff

* They only differ in half of a byte!

Conclusion

In this assignment, we successfully implemented and demonstrated a Differential Power Attack (DPA) using Python. The results reveal that the recovered key closely resembles the original AES key, differing by a mere half-byte. Furthermore, the exploration of different conditions - varying segment lengths and select bits - provided insightful data on how these parameters impact the efficiency of key extraction.

Appendix

Table 1- This table presents the results of our Differential Power Analysis (DPA) for each byte of the cryptographic key, using different select bits. This shows how varying the select bit influences the accuracy and effectiveness of the DPA in extracting the correct byte values of the key.

0	1	2	3	4	5	6	7	Select bit
0x0	0x3f	0xbc	0xff	0xf0	0x8e	0x30	0x16	0
0x2a	0x11	0xb1	0xd2	0xb5	0xaf	0xc0	0xbf	1
0x4b	0xdc	0xf3	0x8a	0x37	0x1d	0x23	0x4a	6
0x33	0xaf	0xa9	0xb2	0x41	0xb4	0x2d	0xa1	0
0x1c	0x44	0xd3	0x3c	0xa0	0x71	0x20	0xe	1
0x55	0x54	0xe0	0x1	0xc	0xc9	0x20	0x4	0
0x3c	0xb9	0x6e	0x83	0xdc	0xf3	0xbc	0xf1	2
0x2b	0x25	0x40	0x71	0xcc	0x1c	0x3d	0xf0	3
0x4	0xf8	0x5d	0xc1	0x5	0xff	0x4f	0xbe	1
0x36	0x99	0x37	0xa3	0x85	0xa3	0x30	0x7b	1
0xfa	0xdd	0x49	0xe8	0xda	0x84	0x2a	0xe3	0, 4, 6
0xbb	0x1e	0x6d	0xa0	0xd9	0x15	0xeb	0x46	0
0xb5	0x9a	0xd7	0x47	0x29	0x71	0xaf	0xbe	
0x45	0x1a	0x6b	0x42	0x2e	0x65	0x3a	0x15	
0xcc	0xfe	0xda	0x28	0x3	0xfe	0x1c	0x20	1, 5
0xff	0xfc	0xa8	0x9	0xef	0xda	0x71	0x7f	0

Table 2- This table shows the extracted key values under four distinct conditions: 1) Using a segment length of 50,000 with select bit 0, 2) The same segment length with select bit as i % 8 (where i is the byte index), 3) Segment length of 50,000 with select bit 1, and 4) Round of 8,000 and step of 2,000. It illustrates the impact of varying segment lengths and select bits on the key extraction process.

	Segment length = 50,000 Select bit = 0	Segment length = 50,000 Select bit = 1	Segment length = 50,000 Select bit = i % 8	Round size = 8,000 Step = 2,000	Select bit , Condition
0x00					0, 1
0x11					1
0x22					1
0x33					0
0x44					0, 1
0x55					0, 1
0x66					0
0x77					0
0x88					0 (20 - 25)
0x99					0, 1
0xaa					0, 1
0xbb					0, 1
0xcc					0 (25 - 30)
0xdd					7 (35-40)
0xee					1
0xff					0, 1