Indian Institute of Technology (IIT-Kharagpur)

SPRING Semester, 2018
COMPUTER SCIENCE AND ENGINEERING

CS60004: Hardware Security

End Semester Examination

Full Marks: 60

Time allowed: 3 hours

INSTRUCTIONS: THIS QUESTION PAPER HAS A CHOICE! Questions 1 and 2 are compulsory. However, you are supposed to answer only one out of questions 3 and 4.

1. Consider the following algorithm for computing modular exponentiation used in the RSA cipher. Our objective is to ascertain the scalar k using side-channel analysis.

Algorithm 1: RSA Modular Exponentiation

Data: Base: X, Secret Exponent $k = k_{n-1}, k_{n-2}, \ldots, k_0$ and modulus N
Result: $Q = X^k$

```
R_0 → 1 ; R_1 → X ;
for i = n − 1 downto 0 do
    R_[1−k_i] → (R_0 × R_1) mod N;
    R_{k_i} = (R_{k_i}^2) mod N ;
return Q = R_0 ;
```

You are also given the power trace values of the 10 exponentiations with different values of the base X, for 8 leakage points, as shown in Table 1. The value of N is 4763.

You are given that the value of $(n-1)^{th}$ bit of k is 1. Find out the value of $(n-2)^{th}$ bit of the k using Correlation Power Analysis (CPA). Assume that the leakage model is Hamming weight.

Table 1: Power Trace Value of RSA execution

Execution No	X	Leakage of $(n-1)^{th}$ bit	Leakage of $(n-2)^{th}$ bit	Leakage of $(n-3)^{th}$ bit	Leakage of $(n-4)^{th}$ bit	Leakage of $(n-5)^{th}$ bit	Leakage of $(n-6)^{th}$ bit	Leakage of $(n-7)^{th}$ bit	Leakage of $(n-8)^{th}$ bit
1	810	13	12	9	12	11	12	10	7
2	891	15	13	7	14	9	17	11	11
3	789	10	11	13	9	12	14	16	8
4	431	8	8	6	6	12	13	10	13
5	918	11	10	9	9	13	11	13	13
6	862	8	6	6	12	10	10	13	9
7	706	8	9	13	16	15	7	12	13
8	742	11	11	13	14	19	7	14	12
9	53	12	12	15	8	14	12	12	12
10	408	10	14	10	12	10	19	11	10

(20 marks)

2. Consider the implementation of binary extended Euclidean algorithm to compute the gcd of two numbers u and v, as below.

```
unsigned int gcd(unsigned int u, unsigned int v)
{
    // simple cases (termination)
    if (u == v)
        return u;

    if (u == 0)
        return v;

    if (v == 0)
        return u;

    // look for factors of 2
    if (u & 1==0) // u is even
    {
        if (v & 1==1) // v is odd
            return gcd(u >> 1, v);
        else // both u and v are even
            return gcd(u >> 1, v >> 1) << 1;
    }

    if (v & 1==0) // u is odd, v is even
        return gcd(u, v >> 1);

    // reduce larger argument
    if (u > v)
        return gcd((u    v) >> 1, v);

    return gcd((v    u) >> 1, u);
}
```

The algorithm has been shown in C-language. This code is recursive and not constant-time. As you can see that the timing would vary and depend on the input parameters, thus offering a side-channel of the parameters through timing. In order to convert this code to a constant time implementation answer the following questions:

(a) Convert this recursive implementation of GCD to an iterative algorithm which is free from recursive function calls. Complete the following code snippet using the comments as hints.

(10 marks)

```
unsigned int gcd(unsigned int u, unsigned int v)
{
  int shift;

  /* GCD(0,v) == v; GCD(u,0) == u, GCD(0,0) == 0 */
  /*           Enter your code here              */

  /* Let shift := lg K, where K is the greatest power of 2
       dividing both u and v. Compute shift*/
  /*           Enter your code here              */
```

```
  while ((u & 1) == 0)
    u >>= 1;

  /* From here on, u is always odd. */
  do {
       /* remove all factors of 2 in v    they are not common */
       /*   note: v is not zero, so while will terminate */
       /*           Enter your code here           */

       /* Now u and v are both odd. Swap if necessary so u <= v,
          then set v = v   u (which is even).  */
       /*           Enter your code here           */

     } while (v != 0);

  /* restore common factors of 2 and return*/
 /*           Enter your code here           */
}
```

(b) Convert the iterative implementation as constructed in the previous question to a constant-time implementation ie, free from if-else structure. (Consider the worst case run-time of the algorithm and make your algorithm run with same execution time irrespective of the input to the algorithm). You can assume that the maximum input size is 32 bits.

(15 marks)

```
unsigned int gcd(unsigned int u, unsigned int v)
{
        int shift=0;
        int uval,vval;
        int retu,retv,orgu,orgv;
        /*save the original inputs incase one of the argument is 0 */
        retu = (u == 0) ? 1:0;
        retv = (v == 0) ? 1:0;
        orgu = u;
        orgv = v;

        /*Calculate the shift := lg K, where K is the greatest power of 2
    dividing both u and v, Compute shift in worst case time*/
        for (int i = 0; i < 32; i++) {
                uval = u;
                vval = v;

         /*           Enter your code here           */

        }
    /*Remove powers of 2 from u in worst case time */
        for (int i = 0; i < 32; i++) {
                uval = u;
           /*           Enter your code here           */
        }
```

```
    /* From here on, u is always odd.*/
        for(int i = 0; i< 32; i++)
        {
                /*copy u and v values in temporary variables */
                uval = u;
                vval = v;
                /*remove all factors of 2 in v in worst case time*/
                for (int i = 0; i < 32; i++) {
            /*          Enter your code here           */
                }
                /*swap intermediate temporary values of u and v*/
                int utmp,vtmp;
                utmp = uval;
                vtmp = vval;
                 /*          Enter your code here           */

                /*Update u and v with the temporary values calculated,
        if (v != 0)*/
                u = (vval != 0) ? uval : u;
                v = (vval != 0) ? vval : v;
        }

        /*Calculate the final return value, handle all cases so that
    overall run time is also worst case and hence constant time*/


        int final;
         /*          Enter your code here           */
        return final;
}
```

3. Consider Fig 1(a), which shows the i^{th} stage of APUF (Arbiter PUF). We consider an alternative PUF design using a similar cascading of switches followed by the same arbiter logic, as in APUF. The alternative, is called as PAPUF. Its i^{th} stage is shown in Fig 1(b). Architecturally, the classic APUF is similar to the PAPUF, with only the stages (switching elements) modified.

We wish to build a linear additive mathematical model for PAPUF. In this context answer the following questions:

(a) As shown in Fig 1(b), every stage has 4 delay elements, denoted as p_i, r_i, q_i and s_i. Write expressions for the propagation delay of the input trigger signal at the output of the $(i+1)^{th}$ stage. Hence, enumerate the delay difference between the top and bottom paths, assuming that the challenge bit in the $(i+1)^{th}$ stage is $\mathbf{c}[i+1] \in \{+1, -1\}$.

(8 marks)

(b) Establish a compact representation for the final delay difference after the n^{th} stage assuming that the challenge is an n-bit vector where each entry is $+1$ or -1. The final expression should be in terms of a feature vector $\boldsymbol{\phi}$ (in terms of the challenge bits) and a weight vector $\mathbf{w}$ (in terms of the delay units).

(4 marks)

(c) Comment on how the linear model for the PAPUF differs from that of the classic APUF.

(3 marks)

Architecturally, the classic APUF and the PAPUF are very similar. The principal difference between APUF and PAPUF lies in the design of switching elements. In contrast to APUF which uses path-swapping switches, PAPUF uses non-swapping path based switches to reduce the implementation bias on FPGA. Unlike APUF, the top path and bottom path in PAPUF do not cross each other along the entire chain of switches.

Let $\Delta(n-1)$ be the delay difference of top and bottom paths after the final switch for a given n-bit challenge $\mathbf{c}$, where $\mathbf{c}$ follows a bipolar encoding. The sign of $\Delta(n-1)$ determines the response for a given challenge, as in the case of the classic APUF.

Next we develop a linear additive delay model for PAPUF. Let p_{i+1} and r_{i+1} be two delay components in the top part of $(i+1)$-th stage, and q_{i+1} and s_{i+1} be the other two delay components in the bottom part of $(i+1)$-th stage.

Let $\delta_t(i)$ and $\delta_b(i)$ be the propagation delays of the trigger signal along the top and bottom paths to the input of the $(i+1)$-th stage of PAPUF, respectively. For a given challenge $\mathbf{c} \in \{+1, -1\}$, propagation delay of the trigger signal at the output of the $(i+1)$-th stage is given by:

$$\delta_t(i+1) = \frac{1+\mathbf{c}[i+1]}{2}(\delta_t(i) + p_{i+1}) + \frac{1-\mathbf{c}[i+1]}{2}(\delta_t(i) + r_{i+1})$$
$$\delta_b(i+1) = \frac{1+\mathbf{c}[i+1]}{2}(\delta_b(i) + q_{i+1}) + \frac{1-\mathbf{c}[i+1]}{2}(\delta_b(i) + s_{i+1})$$

Then delay difference between top and bottom paths can be estimated as:

$$\begin{aligned}\Delta(i+1) &= \delta_t(i+1) - \delta_b(i+1) \\ &= \frac{1+\mathbf{c}[i+1]}{2}(\Delta(i) + p_{i+1} - q_{i+1}) + \frac{1-\mathbf{c}[i+1]}{2}(\Delta(i) + r_{i+1} - s_{i+1}) \\ &= \Delta(i) + \alpha_{i+1}\mathbf{c}[i+1] + \beta_{i+1},\end{aligned} \tag{1}$$

where

$$\alpha_{i+1} = \frac{p_{i+1} - r_{i+1} - q_{i+1} + s_{i+1}}{2}, \text{ and } \beta_{i+1} = \frac{p_{i+1} + r_{i+1} - q_{i+1} - s_{i+1}}{2}.$$

(a) APUF's i-th stage (b) PAPUF's i-th stage

Figure 1: Switching stages of classic APUF and PAPUF. Depending on the challenge bit $\mathbf{c}[i] \in \{+1, -1\}$, a pair of delay elements is selected.

To establish a compact representation for delay difference $\Delta(n-1)$ after the final stage, we follow the following sequence of substitutions:

$$
\begin{aligned}
\Delta(-1) &= 0 \qquad (2)\\
\Delta(0) &= \Delta(-1) + \alpha_0\mathbf{c}[0] + \beta_0 = \alpha_0\mathbf{c}[0] + \beta_0\\
\Delta(1) &= \alpha_1\mathbf{c}[1] + \alpha_0\mathbf{c}[0] + \beta_1 + \beta_0\\
\Delta(2) &= \alpha_2\mathbf{c}[2] + \alpha_1\mathbf{c}[1] + \alpha_0\mathbf{c}[0] + \beta_2 + \beta_1 + \beta_0\\
&\cdots\\
\Delta(n-1) &= \alpha_{n-1}\mathbf{c}[n-1] + \cdots + \alpha_0\mathbf{c}[0] + (\beta_{n-1} + \cdots + \beta_0)\\
&= \mathbf{w}[n]\mathbf{\Phi}[n] + \mathbf{w}[n-1]\mathbf{\Phi}[n-1] + \cdots + \mathbf{w}[0]\mathbf{\Phi}[0] = \langle \mathbf{w}, \mathbf{\Phi} \rangle,
\end{aligned}
$$

where

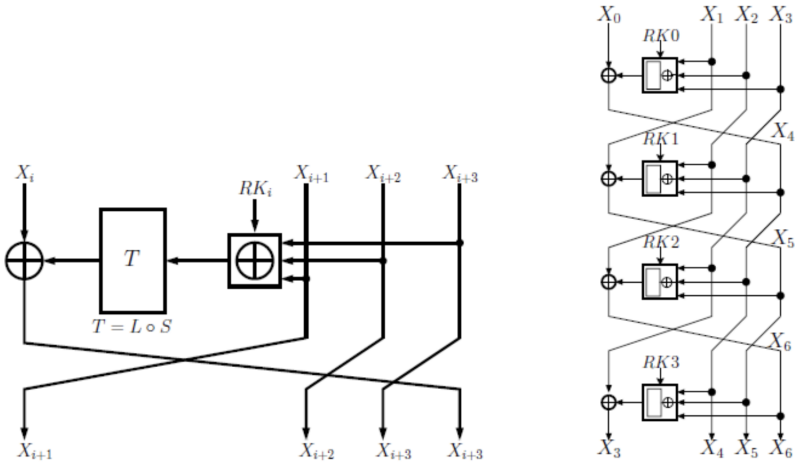
$$
\mathbf{w}[i] = \begin{cases} \beta_{n-1} + \cdots + \beta_0 & \text{if } i = n \\ \alpha_i & \text{if } i = 0, \ldots, n-1, \end{cases}
$$

$$
\mathbf{\Phi}[0 : n-1] = \mathbf{c}[0 : n-1], \text{ and } \mathbf{\Phi}[n] = 1.
$$

Thus, PAPUF also follows a linear additive delay model, very similar to the classic APUF. However, there are also important differences between the delay models of classic APUF and PAPUF:

(a) The feature vector $\mathbf{\Phi}$ in PAPUF model is the challenge $\mathbf{c}$ itself while $\mathbf{\Phi}$ in APUF is the parity vector of challenge $\mathbf{c}$.

(b) All elements of $\mathbf{w}$ (except $\mathbf{w}[n]$) in PAPUF model depend only on the α values, whereas most of the elements of $\mathbf{w}$ in APUF depend on both α and β values. In addition, $\mathbf{w}[n]$ in APUF depends on the β_{n-1} value, but $\mathbf{w}[n]$ in PAPUF depends on $\beta_0, \cdots, \beta_{n-1}$. In both the cases, α_i and β_i have different definitions.



(a) Round i [24] (b) First 4 Rounds

Figure 2: Round Structure of SMS4

4. Figure 2 shows the structure of a block cipher SMS4. Figure 2(a) shows one round of the cipher, while Figure 2(b) shows 4 rounds of the cipher. The inputs to round i ($0 \leq i \leq 31$) are four 32 bit words $X_i, X_{i+1}, X_{i+2}, X_{i+3}$, and a round key RK_i, which also of size 32 bits. In each round, the 32-bit inputs (X_i) and round key RK_i are divided into 4 parts, each of 8 bits. These are respectively denoted $X_{i,0}$, $X_{i,1}$, $X_{i,2}$, $X_{i,3}$ and $RK_{i,0}, RK_{i,1}, RK_{i,2}, RK_{i,3}$.

Each round of SMS4 generates the word X_{i+4} as:
$X_{i+4} = F(X_i, X_{i+1}, X_{i+2}, X_{i+3}) = X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus RK_i)$
. The transformation T is a composition of S-Boxes (S) and diffusion layer (L). You do not need the details of the layer L. The layer S is made of 4 smaller and same s-boxes of dimension 8×8 each. The s-box is implemented as a table in the software implementation of the cipher, and thus leads to cache hit-misses.

The round function has thus 4 cache accesses from each of the four words of $X_{i+1}, X_{i+2}, X_{i+3}$. For example, the first access to the s-box is $X_{1,0} \oplus X_{2,0} \oplus X_{3,0} \oplus RK_{0,0}$.

(a) Let a plaintext (X_0, X_1, X_2, X_3) be chosen such that there are 7 cache hits out of the 8 table accesses in the first 2 rounds (Note that the first memory access can never be a cache hit by the assumption of a flushed cache at the start of encryption). Derive the cache collision equations for Round-0 and Round-1.

(6 marks)

(b) Let P = (X_0, X_1, X_2, X_3) and $\hat{P} = (\hat{X_0}, \hat{X_1}, \hat{X_2}, \hat{X_3})$ be two such plaintexts which satisfy your derived equations and $P \neq \hat{P}$. This set of equations help in removing wrong candidates of RK0. Derive the equations in such scenario.

(6 marks)

(c) Provide an estimate to the number of plaintext pairs $(P, \hat{P})$ required to uniquely retrieve RK0. Assume that the cache line is of size m bytes, and the table used to implement the s-box is of 2^l bytes. The number of most significant bits revealed from each cache access is therefore $n = l - log_2 m$.

(3 marks)