SENG440 Embedded Systems

– Lesson 108: RSA Cryptography –

Mihai SIMA

msima@ece.uvic.ca

Academic Course

Copyright © 2019 Mihai SIMA

Disclaimer

The purpose of this course is to present general techniques and concepts for the analysis, design, and utilization of embedded systems. The requirements of any real embedded system can be intimately connected with the environment in which the embedded system is deployed. The presented design examples should not be used as the full design for any real embedded system.

Lesson 108: RSA Cryptography

RSA Cryptography – background

- RSA: invented by **R**ivest, **S**hamir, and **A**dleman in 1978
- Usage: secure transmissions over wireless channels
- It uses keys with the length ranging from 512 to 2048 bits
- RSA is based on two distinct odd prime numbers P and Q
- These prime numbers are used to generate two key-pair values:
  - **Public key-pair** (E, PQ) used to encrypt data
  - **Private key-pair** (D, PQ) used to decrypt data
- D, E, and M are very long integers of 512 - 2048 bits

RSA Cryptography – the algorithm I

1. Find P and Q, two large (e.g., 1024-bit) prime numbers.
2. Choose E such that: $E > 1$, $E < PQ$, and E and $(P-1)(Q-1)$ are relatively prime (they have no prime factors in common). E does not have to be prime, but it must be odd. $(P-1)(Q-1)$ can't be prime because it's an even number.
3. Compute D such that $(DE-1)$ is evenly divisible by $(P-1)(Q-1)$.
   - This is written as $DE = 1 \mod [(P-1)(Q-1)]$
   - The number D is called the multiplicative inverse of E.

   This is easy to do – simply find an integer X which causes

$$D = (X(P-1)(Q-1)+1)/E$$

   to be an integer, then use that value of D.

RSA Cryptography – the algorithm II

4. The encryption function is

$$C = T^E \mod PQ$$

where C is the ciphertext (a positive integer), T is the plaintext (a positive integer). The message being encrypted, T, must be less than the modulus, PQ.

5. The decryption function is

$$T = C^D \mod PQ$$

where C is the ciphertext (a positive integer), T is the plaintext (a positive integer).

RSA Cryptography – the algorithm III

- Your public key is the pair (PQ, E).
- Your private key is the number D (reveal it to no one).
- The product PQ is the modulus (often called N in the literature).
- E is the public exponent.
- D is the secret exponent.
- You can publish your public key freely, because there are no known easy methods of calculating D, P, or Q given only (PQ, E) (your public key).
- If P and Q are each 1024 bits long, the sun will burn out before the most powerful computers can factor your modulus into P and Q.

RSA Cryptography – an example I

- Choose the **first prime number** (destroy it after computing E and D):

$$P = 61$$

- Choose the **second prime number** (destroy it after computing E and D):

$$Q = 53$$

- Calculate the **modulus** (give this to others)

$$PQ = 3233$$

- Choose the **public exponent** (give this to others)

$$E = 17$$

- Calculate the **private exponent** (keep this secret!)

$$D = 2753$$

RSA Cryptography – an example II

- Your public key is (E, PQ). Your private key is D.
- The encryption function is:

$$\text{encrypt}(T) = (T^E) \mod PQ = T^{17} \mod 3233$$

- The decryption function is:

$$\text{decrypt}(C) = (C^D) \mod PQ = C^{2753} \mod 3233$$

- To encrypt the plaintext value 123, do this:

$$\text{encrypt}(123) = 123^{17} \bmod 3233 =$$
$$= 337587917446653715596592958817679803 \bmod 3233 = 855$$

- To decrypt the ciphertext value 855, do this:

$$\text{decrypt}(855) = 855^{2753} \bmod 3233 = 123$$

RSA Cryptography – implementation challenges

- Arithmetic operations on very long integers: T^E and S^D cannot be computed using common techniques
  - Ideally: suport long-word arithmetic in hardware
  - The bandwidth from register file to functional units is limited and cannot be increased easily
- True data dependencies
  - Rewriting the algorithm to expose the parallelism
  - Collapsing operations and suport the compound in hardware
- Expensive modular and multiplication operations
  - Montgomery Modular Multiplication (MMM)
  - Montgomery Modular Exponentiation (MME)

How to calculate the value of $855^{2753} \mod 3233$ I

- We know that $2753 = 101011000001_2$, therefore

$$2753 = 1 + 2^6 + 2^7 + 2^9 + 2^{11} = 1 + 64 + 128 + 512 + 2048$$

- Consider this table of powers of 855:

$$\begin{aligned}
855^1 &= 855 \pmod{3233} \\
855^2 &= 367 \pmod{3233} \\
855^4 &= 367^2 \pmod{3233} = 2136 \pmod{3233} \\
855^8 &= 2136^2 \pmod{3233} = 733 \pmod{3233} \\
855^{16} &= 733^2 \pmod{3233} = 611 \pmod{3233}
\end{aligned}$$

How to calculate the value of $855^{2753} \mod 3233$ II

- The table of powers of 855 (cont'd):

$$\begin{aligned}
855^{32} &= 611^2 \pmod{3233} = 1526 \pmod{3233}\\
855^{64} &= 1526^2 \pmod{3233} = 916 \pmod{3233}\\
855^{128} &= 916^2 \pmod{3233} = 1709 \pmod{3233}\\
855^{256} &= 1709^2 \pmod{3233} = 1282 \pmod{3233}\\
855^{512} &= 1282^2 \pmod{3233} = 1160 \pmod{3233}\\
855^{1024} &= 1160^2 \pmod{3233} = 672 \pmod{3233}\\
855^{2048} &= 672^2 \pmod{3233} = 2197 \pmod{3233}
\end{aligned}$$

How to calculate the value of $855^{2753} \mod 3233$ III

- Given the above, we know this:

$$
\begin{aligned}
& 855^{2753} \pmod{3233} = \\
& = 855^{1+64+128+512+2048} \pmod{3233} = \\
& = 855^{1} \times 855^{64} \times 855^{128} \times 855^{512} \times 855^{2048} \pmod{3233} = \\
& = 855 \times 916 \times 1709 \times 1160 \times 2197 \pmod{3233} = \\
& = 794 \times 1709 \times 1160 \times 2197 \pmod{3233} = \\
& = 2319 \times 1160 \times 2197 \pmod{3233} = \\
& = 184 \times 2197 \pmod{3233} = \\
& = 123 \pmod{3233} = \\
& = 123
\end{aligned}
$$

How to calculate the value of $855^{2753} \mod 3233$ IV

- The modulus is not changed frequently (it is the same per transaction), thus the table of powers can be computed off-line.
- What is the size of this table of powers?
- The table of powers is too large for an embedded system
- Many other techniques to calculate the modular exponentiation have been proposed
  - Modular multiplication is the core of modular exponentiation
- Montgomery arithmetic – recommended literature:
  John Fry and Martin Langhammer, RSA & Public Key Cryptography in FPGAs, Altera Corporation.

Modular exponentiation I

- A common way: the **multiply and square algorithm**

$$Z = X^E \mod M \qquad \text{where } E = \sum_{i=0}^{n-1} e_i 2^i$$

```
1 $Z_0 = 1$, and $P_0 = X$
2 FOR $i = 0$ to $n - 1$ LOOP
3     $P_{i+1} = P_i^2 \mod M$
4     IF $e_i = 1$ THEN $Z_{i+1} = Z_i \cdot P_i \mod M$ ELSE $Z_{i+1} = Z_i$
5 END FOR
```

Modular exponentiation II

- The multiply and square algorithm is a running accumulation of squaring and multiplication steps
- At each stage the *modulo* function is performed to keep any intermediate variables within the integer range of M
- A second option is to allow the intermediate variables to grow and perform the *modulo* function as a single final operation
- The first option is typically more desirable as it will keep the multiplication functions down to a practical bit width
- The brute force approach in implementing the *modulo* function involved a divide operation to discover the remainder
- Division is expensive and should be avoided
- The efficiency of the modular multiplier used in the multiply and square algorithm is key to the performance of RSA-based crypto systems.

Montgomery Modular Multiplication (MMM) I

- Assume we want to calculate $X \cdot Y \mod M$
- Notation: m is the bit-length of M, and $R = 2^m$.
- Definition: the inverse of R is the number R^{-1} such as:

$$R \cdot R^{-1} \mod M = 1$$

- MMM (efficiently) calculates

$$Z = X \cdot Y \cdot R^{-1} \mod M$$

Montgomery Modular Multiplication (MMM) II

- A way to deal with this scale factor is to pre-scale up each operand using MMM:

$$\bar{X} = X \cdot R^2 \cdot R^{-1} \mod M = X \cdot R \mod M$$
$$\bar{Y} = Y \cdot R^2 \cdot R^{-1} \mod M = Y \cdot R \mod M$$

- Then, the product is also scaled up with R:

$$\bar{Z} = (X \cdot R) \cdot (Y \cdot R) \cdot R^{-1} \mod M = X \cdot Y \cdot R \mod M$$

- All MMM operations with scaled-up arguments generate a scaled-up result.

Montgomery Modular Multiplication Algorithm I

- MMM calculates $X \cdot Y \cdot R^{-1} \mod M$
- $M = (M(m-1) \ldots M(1)M(0))$

  m is the bit-length of M

  $M(0)$ is the least-significant bit, $M(m-1)$ is the most-significant bit, etc.
- $X = (X(m-1) \ldots X(1)X(0))$ and $Y = (Y(m-1) \ldots Y(1)Y(0))$

  m is also the bit-length of X and Y

  $X(0)$ is the LS bit of X, $Y(m-1)$ is the MS bit of Y, etc.
- $R = 2^m$
- Only the **bit-wise implementation** is presented.

Montgomery Modular Multiplication Algorithm II

- $Z = X \cdot Y \cdot R^{-1} \mod M$
- The pseudocode of the bit-wise MMM:

```
$T = 0$
**for** $i = 0$ to $m - 1$ **do**
    $\eta = T(0)\ \mathrm{XOR}\ (X(i)\ \mathrm{AND}\ Y(0))$
    $T = (T + X(i)Y + \eta M) \gg 1$
**if** $T \geq M$ **then**
    $T = T - M$
$Z = T$
```

Montgomery Modular Multiplication – Example

- $X = 17 = 10001_2$, $Y = 22 = 10110_2$, $M = 23 = 10111_2$
- $m = 5$, $R = 2^5 = 32$, $R^{-1} = -5$ since $32 \cdot (-5) = -160 = (-7) \cdot 23 + 1$
- $X \cdot Y \cdot R^{-1} \mod M = 17 \cdot 22 \cdot (-5) \mod 23 = 16$
- The same result is obtained using the MMM algorithm:

i	$X(i)$	$\eta = T(0)\hat{}(X(i)\,\&\,Y(0))$	$T = (T + X(i) \cdot Y + \eta \cdot M) \gg 1$
			0
0	1	$0 + 1 \cdot 0 = 0$	$(0 + 1 \cdot 22 + 0 \cdot 23) \gg 1 = 11$
1	0	$1 + 0 \cdot 0 = 1$	$(11 + 0 \cdot 22 + 1 \cdot 23) \gg 1 = 17$
2	0	$1 + 0 \cdot 0 = 1$	$(17 + 0 \cdot 22 + 1 \cdot 23) \gg 1 = 20$
3	0	$0 + 0 \cdot 0 = 0$	$(20 + 0 \cdot 22 + 0 \cdot 23) \gg 1 = 10$
4	1	$0 + 1 \cdot 0 = 0$	$(10 + 1 \cdot 22 + 0 \cdot 23) \gg 1 = 16$

RSA Cryptography – project requirements I

- Determine the word length needed to implement the example presented in class
  - If word-length is less than 32 bits, then you can implement the RSA algorithm within the standard instruction set
  - If word-length is greater than 32 bits, then you need to write routines to implement long-word arithmetic
- Provide a pure-software solution and determine its performance (cycle count)
  - The look-up table (LUT) will be stored into memory
  - How large this LUT should be? Will cache misses be encountered?
  - Try to implement the Montgomery Modular Multiplication and Modular Exponentiation in software
- Try a firmware solution
  - The code is sequential – any improvements possible?
  - The firmware engine can be geared to implement long-word arithmetic

RSA Cryptography – project requirements II

- Support computation in hardware
  - Try to implement the modular operations on powers of the ciphertext value in parallel
  - Add the results of these modular operations using a multi-operand adder
  - Try to implement the Montgomery Modular Multiplication and Modular Exponentiation in hardware
- Specify how many gates are needed to support all or part of the RSA algorithm in hardware

References

- John Fry and Martin Langhammer, *RSA & Public Key Cryptography in FPGAs*, Altera Corporation, 2005.
- Steve Burnett, *RSA Security's Official Guide to Cryptography*, McGraw-Hill, 2001.
- Bruce Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*, Wiley, 2015.
- Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno, *Cryptography Engineering: Design Principles and Practical Applications*, Wiley, 2010.
- Christof Paar, Jan Pelzl, and Bart Preneel, *Understanding Cryptography: A Textbook for Students and Practitioners*, Springer, 2010.

Questions, feedback

Notes I

Notes II

Notes III

Notes IV

Project Specification Sheet

- **Student name:** . . . .
- **Student ID:** . . . .
- **Function to be implemented:** . . . .
- **Argument range:** . . . .
- **Wordlength:** . . . .