RSAB, the RSA box

Noah Stebbins (nes2137)
Adam Incera (aji2112)
Jaykar Nayeck (jan2150)
Emily Pakulski (enp2111)

February 2015

Contents

1 Overview

We plan on building a hardware accelerator for the RSA encryption algorithm. There are three main steps: key generation, encryption, and decryption.

1.1 Generating Keys

Below are the steps to generate the private and public keys, `d` and `e`, respectively.

- Select two distinct prime numbers p and q. A list of large prime numbers can be stored on an SD card on the FPGA. To select the prime numbers we can use the `$random` function on the index of the large prime numbers. However, there might be better ways to generate random numbers, and we might explore interfacing with a *true random number generator*, or having a dedicated portion of the chip generate random numbers.
- Multiply both numbers, yielding $n (n = pq)$. This might require our own SystemVerilog implementation of multiplication because we expect the numbers to be in the 400 bit range. Then we need to multiply $p - 1$ and $q - 1$; this should not be a problem if we use the function from the previous step.
- Find a positive integer that is less than $(p - 1)(q - 1)$ and is coprime with $(p - 1)(q - 1)$, a value called e. This may prove difficult because we will not want to use loops, so we need to do more research. (However, a quick initial search revealed that there are efficient ways to compute e by raising something to the power of 2 and adding 1. So something like $2^{37}+1$ is probably coprime with $(p-1)(q-1)$. This can be done extremely efficiently on an FPGA because raising 2 to the power of something would just be bit shifts.)
- Determine d, the multiplicative inverse of $e(\bmod (p-1)(q-1))$. This can be done efficiently using the extended Euclid's algorithm.

The FPGA would then publish (e, n) and keep d secret.

1.2 Encrypting Messages

To encrypt a message m, another person would compute c, the cypher text, using the (e, n) values published by means of the operation $c \equiv m^e (\bmod n)$. This can be done efficiently using modular exponentiation.

1.3 Decrypting Messages

To decrypt a message c, the person would compute $m \equiv c^d (\bmod n)$. This can also be done efficiently using modular exponentiation. We might want to

explore binary exponentiation to drastically reduce the time in the modular exponentiation.

1.4 Software Portion

If our implementation is sufficiently fast, we could add an extra layer of software on top of the FPGA so two FPGAs running this code could demonstrate usage. We could build a very secure message service where we would send RSA encrypted messages with constantly changing public and private keys.

This would demonstrate the value of a hardware-level implementation, because the encryption speed gains could allow this kind of optimization. Changing keys periodically would make it even more difficult to decrypt the messages the keys.