Implementation of the RSA algorithm in MIPS assembly

Daniel Federschmidt, Dominic Pfeil

Computer Architecture, DHBW Stuttgart, TINF13A
Email: daniel.federschmidt@googlemail.com, dominic.pfeil@web.de

This article describes an implementation of the RSA cryptosystem for the mips processor architecture. The assembly implementation leverages algorithms often found in current RSA products on the market. It was created as an assignment for the computer architectures course at the DHBW Stuttgart.

Keywords: Cryptography, Assembly, Processor Architecture, MIPS, Encryption

Filing Date: August 17, 2015

1. INTRODUCTION TO RSA

RSA is a cryptosystem for public-key encryption published by Ron Rivest, Adi Shamir and Leonard Adleman at the Massachusetts Institute of Technology in 1978.[1] It uses several operations and properties of modulus arithmetic and makes decryption of the keys is an problem in the class NP. If keys are generated with a sufficient size, an enormous amount of computational power is needed to extract the private key. Current RSA implementations usually use RSA keys of the size 2048 bits.

2. HIGH-LEVEL IMPLEMENTATION

The RSA algorithm used is implemented in the high-level language python. It is required to run under python 3 because it uses some of the newer features of the language. It requires the user to put in relevant information and en- and decrypts it. Additionally, there is an second version of the program, which is encrypting a file named `clear.txt` and encrypts it again.

3. PRIMENUMBER GENERATION

The generation of prime numbers boils down to the problem if a certain number is a prime or not. This problem was proven to be of polynomial complexity. The algorithm used in the Mips implementation is the Miller-Rabin Algorithm, which determines whether a number is prime in a polynomial time across all inputs. It is a non-deterministic monte-carlo algorithm, meaning it takes randomly generated values to execute. Additionally, it's correctness depends on the not yet proven Riemann Hypothesis. [2]In 2004, M. Agrawal and colleagues proposed an algorithm, the AKS primality test with polynomial runtime which is deterministic and does not rely on an unproven hypothesis.[3] While it's properties certainly look better than the Miller-Rabin algorithm, it is not used in practice because it performs worse than the Miller-Rabin algorithm does.

In the implementation, it is executed 30 times. As correctness increases by 75% every additional test, the probability for an false positive is close to nothing. If more performance is needed, the number of tests could be decreased.

Other approaches which could be taken to generate the two prime numbers necessary is an sieve-based approach, like the sieve of Eratosthenes or the sieve of Atkin. They both are most effective if many primes in a certain area need to be found, because they eliminate multiples of numbers until only primes are left in the area, which is often represented by an array.

In order to find the two primes, the user enters a prime seed n, which is linearly incremented until two primes are found.

For this implementation, the Miller-Rabin algorithm with linear probing was chosen, because the algorithm is used in professional solutions, even if it performs not that good for small numbers and because it was fun to implement.

[1] http://math.arizona.edu/sites/math.arizona.edu/files/webfm/undergrad/uta/Spring12UTATalkSalterbergJake.pdf
[2] http://www.cs.cornell.edu/courses/cs4820/2010sp/handouts/MillerRabin.pdf
[3] http://mathworld.wolfram.com/AKSPrimalityTest.html

4. CALCULATION OF MODULAR MULTIPLICATIVE INVERSE

The implementation uses euclid's extended algorithm to calculate the modular multiplicative inverse, needed to generate the private exponent from the public exponent and the totient. It is a much faster method of computing the private exponent than just linearly testing values.

5. MODULAR EXPONENTIATION

In this implementation, modular exponentiation by squaring is used to calculate $x^k \mod n$ efficiently.

6. ADDITIONAL IMPLEMENTATION DETAILS

The implementation is logically structured in subroutines, called by a top-level `main()` function. It is responsible for I/O and calling the appropriate functions. Depending on the complexity of the subroutines, they call subroutines themselves, respecting the mips calling convention. For example, the Miller-Rabin test function `checkprime()` is called by the `generateprimes()` function and calls `trycomposite()`, which checks if a number is a composite. Every time, the saved registers are stored on the stack by the callee and restored at the end of the routine. No frame pointer is used, as there is no point in the program where the stack allocation is determined at runtime where a frame pointer would be useful.

7. PROGRAM FLOW AND FEATURES

At first, the user is asked to input a prime seed n, which is the starting point for the prime generation by linear probing using Miller-Rabin. If two primes are found, the program calculates the modulus, the totient and prompts for an public exponent. Using that public exponent, a private exponent is calculated.

Then, the program asks whether it should en- and decrypt an text-file or input from the console. The text file must be located in the same directory where the MARS simulator is executed. When the file is chosen, it's contents are read and en- and decrypted. Of console input is chosen, the program prompts for text input which is then en- and decrypted. After successful en- and decryption, the program loops and asks for input again.

8. LIMITATIONS AND POSSIBLE OPTIMIZATIONS

This implementation does not feature the possibility to have prime numbers larger than about 120. This is due to the 32-bit architecture. Having larger prime numbers generated will overflow a register which stores the modulus, which is calculated by multipling both primes. To extend the range of possible primes to pick, the implementation must feature a 64-bit modulus. Additionally, input is limited to 100 characters.

Additionally the prime numbers can't be too small, because otherwise, the modulus is too small. If the modulus is smaller than 127, it's not possible to en-decrypt every Ascii-character.

At the moment, the program must be executed using the Mars-simulator. It features the syscall 42, which simulators like qtSpim, xSpim or conventional Spim do not implement. It generates a random number between 0 and an upper bound. This behaviour could be emulated using a pseudo-random number generation function in order to achieve emulator-independancy.

When an text file is chosen as an input, an file called output txt is generated which contains the encrypted text of the input file. In the current implementation, there is no option to decipher this text file, which could be implemented by adding another menu flow which guides the user through decipher process, which involves reading in the file, calling the decipher function and printing the output.

In this implementation, each character is encrypted using the key individually. This is not advisable for an real RSA implementation, as each character always has the same ciphertext representation, which would not be better than a conventional monoalphabetic substituton. Monoalphabetic substitutions are known to be vulnerable to quantity analysis of the individual characters due to this nature, which could easily be used to decrypt the ciphertext, if enough ciphertext is obtained by the attacker. So a possible optimization for the implementation, larger keylenghts would be a big benefit.