*Christian Jarani, CS1699: Privacy, Project 1*

**W0**: Modular Exponentiation

- Inputs: base, exponent, modulus -> represented by b, e, and m respectively.

- Outputs: b^e mod m

**W1**: Modular exponentiation is widely used in many modern cryptosystems, where large exponentiations are commonplace: it allows us to quickly find the remainder of large integers. It allows RSA, one of the most well-known and widely-used encryption algorithms today, to be practical when used as part of a larger cryptosystem. Since finding e in the output above is considered a hard problem, it is unrealistic, even for a computer, to solve.

**W4**: The table & graph below show runtimes (in seconds) of algorithm A for two different exponent constructions with varying bit sizes over 3 test runs:

|  |  |  |
| --- | --- | --- |
| ***Bit Size*** | ***0 Bits*** | ***1 bits*** |
| *256* | 0.00024 | 0.00098 |
| 0.00023 | 0.00119 |
| 0.00018 | 0.00101 |
| *512* | 0.00121 | 0.00192 |
| 0.00102 | 0.00256 |
| 0.00128 | 0.00228 |
| *1024* | 0.00565 | 0.01354 |
| 0.00684 | 0.01348 |
| 0.00704 | 0.00978 |
| *2048* | 0.03982 | 0.07328 |
| 0.03779 | 0.07142 |
| 0.03484 | 0.07143 |

* To compile, run “javac p1.java”. To execute, run “java p1”
* At the start of the program, you will be prompted to choose Algorithm A or B, labelled 1 or 2 respectively.
* Bit size was calculated as ARRAY\_LENGTH \* 8, or the bit size of a byte.
* Values of graph were taken as averages of the three trials for each exponent type (all 0 vs. all 1) for each bit size
* Refer to lines 54 and 55 in p1.java for the choice of operation (“square” vs. “square and multiply”) based on the next bit of the exponent
  + The next byte of the exponent ‘e’ is selected by the outermost for loop on line 50 and 51, which cycles through all bytes of ‘e’ from lowest order to highest. I then logical right shift on line 53 by the current iteration of the innermost for loop (where i = [0,7]) and mask it by 1 to isolate the lowest order bit. This innermost loop cycles through all bits in the current byte before moving on to the next byte of the exponent.
* Algorithm A was tested on exponents constructed in 2 separate ways:
  + entirely of 0 bits
  + entirely of 1 bits
* RUNTIME: Algorithm A varies in runtime based on **two** criteria:

1. *The number of 1 bits in the binary form of the exponent,*
   1. SECURITY VULNERABILITY: Because the time to calculate the square operation is much less than for the square and multiply operation, precise measurements of them allows an attacker to work backwards to the input.
      1. *Environment*: types of attacks possible with respect to the given environment. Ciphertext-only attacks imply the others are possible, as well; known-plaintext implies chosen plain-text is also possible; chosen plaintext implies it is the only attack possible in the given environment.
         * The attacker may have knowledge of the internals of the hardware implementation (primarily, of the CPU), but doesn’t have physical access to the machine. This includes statistics such as CPU usage and speed.
           1. This would imply a **known-plaintext** attack, since the attacker can observe timing differences guided by plaintext/ciphertext pairs
         * Attacker also has physical access to the machine
           1. They can use a sound attack by monitoring the noise generated by the CPU during calculation. Louder noise usually means the CPU is doing more work, i.e. calculating a square and multiply operation rather than square operation.
           2. This would allow a **ciphertext-only attack**. Timing differences before ciphertext is generated can leak information about the bit of the exponent and allow the attacker to backtrack to the input.
         * If attacker knows the cryptographic algorithm being used in addition to one or both of the environment above, he could execute a **chosen-plaintext** attack with relative ease.
2. *and the bit size of the exponent.*
   1. SECURITY VULNERABILITY: Since the runtime of Algorithm A increases exponentially based on the size of the exponent, multiple rounds of monitoring can reveal this information.
      1. Environment: same as above

**W7**: For Algorithm B, I chose to implement the side-channel protected Montgomery Ladder function as described in Marc Joy and Sung-ming Yen’s paper.

|  |  |  |
| --- | --- | --- |
| ***Bit Size*** | ***0 Bits*** | ***1 bits*** |
| *256* | **0.00036** | **0.00036** |
| **0.00035** | **0.00037** |
| **0.00036** | **0.00035** |
| *512* | **0.00193** | **0.00217** |
| **0.00233** | **0.00168** |
| **0.00145** | **0.00202** |
| *1024* | **0.01283** | **0.01294** |
| **0.01371** | **0.01296** |
| **0.01159** | **0.01004** |
| *2048* | **0.06794** | **0.06783** |
| **0.06379** | **0.06373** |
| **0.06478** | **0.06503** |

* Bit size was calculated as ARRAY\_LENGTH \* 8, or the bit size of a byte.
* Values of graph were taken as averages of the 3 trials for each exponent type (all 0 vs. all 1) for each bit size.
* Algorithm B was tested on exponents constructed in 2 separate ways:
  + entirely of 0 bits
  + entirely of 1 bits
* SIDE-CHANNEL MITIGATION
  + *Branch Condition*: Since the main vulnerability of Algorithm A lies in execution time differences due to its conditional (line 74), Algorithm B attempts to remedy this by making execution time constant regardless of the current bit value of the exponent. Looking at lines 103 to 109, we perform two multiplications and two modulo operations for either condition. Compared to Algorithm A, our runtime is now become dependent on key length rather than the number of 1’s in the binary form of the exponent. This helps mitigate threats from side-channels that reverse-engineer the private key based on the state of the CPU.
    - *Test Results*
      * Although not truly constant time, Algorithm B is a significant improvement over Algorithm A in terms of runtime consistency over varying input sizes (256, 512, 1024, 2048 bits) and formats (all 0’s and all 1’s). However, **on the first iteration of Algorithm B** for an arbitrary exponent bit size, **the runtime is significantly higher than the other trials** (should multiple trials be chosen to run). **I could not find the cause of this anomaly.**
    - *Tradeoffs*
      * The cost of this improved security is an increased runtime for exponents whose binary representations contain very few 1 bits. This is due to the runtime-equalized conditionals in Algorithm B (0 bit branch performs same number of operations as 1 bit branch).

**W8**: Algorithm B still leaks information concerning the bit length of our key. Because the runtime still increases proportional to the key’s bit length, astute observation through a side channel could still reveal this info.

* Could watch for repetitive CPU behavior with knowledge of the system specs and the algorithm being used
* Other than that, if the multiplication algorithm used has been optimized for certain inputs, it could be possible to determine the bits of the key based on certain values of r0 and r1 (lines 103 through 109). Since r0 will never be that large, it could be contained in a different sized type than r1, and the multiplication operation could be centered around this assumption. Unlikely, but possible.