Question 4)

Sage has a command “time” that works similar to “print.” Specifically “time <expr>” runs the expression <expr> and displays some timing information. This exercise is to use this time command to try some experiments timing modular exponentiation with different parametera. For varying values of *m* and *n* (positive integers) generate a prime *p* at most 2*m*, and a random positive integer *a* less than *p*, then time calculating ModExp(*a*, *e*, *p*), for e = 2*n* and 2*n*-1. For each different value of *e* run the experiment several times. Try this with at least two different values for (*m*,*n*). Be sure to try varying the sizes of these parameters drastically (i.e. on the order of 10s and 100s.) What do you notice? What does this tell you? [Hint: If you do the experiment correctly, you should make an observation that forms the basis for side channel attacks, a powerful type of attack on crypto systems.]

Question 4)

Sage has a command “time” that works similar to “print.” Specifically “time <expr>” runs the expression <expr> and displays some timing information. This exercise is to use this time command to try some experiments timing modular exponentiation with different parametera. For varying values of *m* and *n* (positive integers) generate a prime *p* at most 2*m*, and a random positive integer *a* less than *p*, then time calculating ModExp(*a*, *e*, *p*), for e = 2*n* and 2*n*-1. For each different value of *e* run the experiment several times. Try this with at least two different values for (*m*,*n*). Be sure to try varying the sizes of these parameters drastically (i.e. on the order of 10s and 100s.) What do you notice? What does this tell you? [Hint: If you do the experiment correctly, you should make an observation that forms the basis for side channel attacks.]

sage: p = random\_prime(p^10)

sage: a = randint(1,p); a

157

sage: time ModExp(a,2^1024,p);

CPU times: user 0.01 s, sys: 0.00 s, total: 0.02 s

Wall time: 0.02 s

sage: time ModExp(a,2^1024,p);

CPU times: user 0.01 s, sys: 0.00 s, total: 0.01 s

Wall time: 0.01 s

sage: time ModExp(a,2^1024,p);

CPU times: user 0.01 s, sys: 0.00 s, total: 0.02 s

Wall time: 0.02 s

sage: time ModExp(a,2^1024,p);

CPU times: user 0.01 s, sys: 0.00 s, total: 0.01 s

Wall time: 0.01 s

sage: time ModExp(a,2^1024-1,p);

CPU times: user 0.02 s, sys: 0.00 s, total: 0.02 s

Wall time: 0.02 s

sage: time ModExp(a,2^1024-1,p);

CPU times: user 0.02 s, sys: 0.00 s, total: 0.02 s

Wall time: 0.02 s

sage: time ModExp(a,2^1024-1,p);

CPU times: user 0.02 s, sys: 0.00 s, total: 0.02 s

Wall time: 0.02 s

sage: time ModExp(a,2^1024-1,p);

CPU times: user 0.02 s, sys: 0.00 s, total: 0.02 s

Wall time: 0.02 s

sage: p = random\_prime(2^512)

sage: a = randint(1,p)

sage: time ModExp(a,2^8096,p);

CPU times: user 0.10 s, sys: 0.02 s, total: 0.12 s

Wall time: 0.12 s

sage: time ModExp(a,2^8096,p);

CPU times: user 0.10 s, sys: 0.02 s, total: 0.12 s

Wall time: 0.12 s

sage: time ModExp(a,2^8096,p);

CPU times: user 0.10 s, sys: 0.02 s, total: 0.12 s

Wall time: 0.12 s

sage: time ModExp(a,2^8096,p);

CPU times: user 0.10 s, sys: 0.02 s, total: 0.12 s

Wall time: 0.12 s

sage: time ModExp(a,2^8096-1,p);

CPU times: user 0.18 s, sys: 0.03 s, total: 0.21 s

Wall time: 0.21 s

sage: time ModExp(a,2^8096-1,p);

CPU times: user 0.18 s, sys: 0.03 s, total: 0.21 s

Wall time: 0.21 s

sage: time ModExp(a,2^8096-1,p);

CPU times: user 0.18 s, sys: 0.03 s, total: 0.21 s

Wall time: 0.21 s

sage: time ModExp(a,2^8096-1,p);

CPU times: user 0.18 s, sys: 0.03 s, total: 0.21 s

Wall time: 0.21 s

In both cases you can notice that on average exponentiation with the power of two takes less time on than exponentiation with a power of two (minus 1), when the prime modulus is much bigger this takes MUCH more time. The reason is because when the exponent is a power of 2 there is only a single one in the binary expansion, and hence the square and multiply method executes only one conditional multiply. In the case of a power of two minus one, the exponent entirely consists of 1s, so every conditional multiply is hit. The time for the conditional multiplies adds up and we can determine how many were taken. From this we could conceivably recover the hamming weight. Indeed, this is not a cryptographic break of the exponentiation, but it certainly indicates how information can be leaked by timing a mathematical operation.