Analyzing timing attack potential against RSA on the UDOO board

Hans-Olav Hessen hansolavhessen@gmail.com
&
Arve Nygård mail@arve.in

***Abstract*—For our project we want to implement the RSA algorithm on the UDOO board.**
RSA relies on modular exponentiation. There are several ways to do this. A commonly used technique for doing the exponentiation is called Montgomery exponentiation. However, this technique is vulnerable to timing attacks. We will implement RSA using both the standard Montgomery exponentiation and an alternative technique called Montgomery powering ladder, and attempt to show whether the UDOO board is susceptible to such a timing attack.

I. Introduction

The RSA algorithm was invented by Ron Rivest, Adi Shamir and Leonard Adleman in 1977. The algorithm is a public-key cryptographic algorithm that is based on the difficulty of the factoring problem.
The algorithm can be divided into three different steps:

- Key generation: Create a private and a public key
- Encryption of the message
- Decryption of the message

An important part of the RSA algorithm is the modular exponentiation. This process consist of reducing the the temporary result modulo n at each step of the exponentiation. In this article we are going to focus on two different methods of doing this: The standard Montgomery method and the Montgomery power ladder.

II. Algorithm walkthrough

A. RSA algorithm

Key generation:

- Choose two large prime numbers p and q and calculate $n = q * p$
- Compute $\phi(n) = \phi(n)\phi(q) = (p-1)(q-1)$
- Choose an e, where $1 < e < \phi(n)$ and where $gcd(\phi(n), e) = 1$. The e is then the public key.
- Compute $d = e^{-1} mod(\phi(n))$ using the extended Euclidean algorithm. d is then the secret key.

Encryption:

- Chipertext $= C = M^e mod(n)$, where M is the original message.

Decryption:

- Message $= M = C^d mod(n)$, where C is the encrypted message.

B. Square-and-Multiply algorithm

[H] Input: $m, d = (d_{k-1}, ..., d_0)_2, N$
Output: $S = m^d modN$
1. $R_0 = 1$
2. For $i = k-1$ to 0 do
   - $R_0 \leftarrow R_0^2 (modN)$
   - if $(d_i = 1)$ then $R_0 \leftarrow R_0$ m (mod N)
3. Return R_0

C. Montgomery powering ladder

[H] Input: $m, d = (d_{k-1}, ..., d_0)_2, N$
Output: $S = m^d modN$
1. $R_0 = 1, R_1 = m$
2. For $i = k - 1 to 0 do$
   - $b \leftarrow 1 - d_i; R_b \leftarrow R_0 R_1 (modN)$
   - $R_{d_i} \leftarrow R_{d_i}^2 (modN)$
3. Return R_0

III. Timing attack

A timing attack is a side-channel attack where the attacker is trying to recreate the secret key of a cryptographic algorithm. The attacker times the duration spent by a process signing a number of messages and analyses the result by simulating the RSA algorithm in a local setting. By doing this the attacker can recover the key bit by bit. The execution time for the Square-and-Multiply algorithm increases depending on the numbers of one-bits in the secret key, and enables the attacker to extract information about the key. While performing these attacks, there are several sources of noise. This could be disk access, latency in network, OS scheduling, et cetera.
The Montgomery powering ladder addresses this problem by always having a fixed set of operations regardless of the current bit value. These operations are squaring and multiplying.

IV. UDOO board

The UDOO is a development platform consisting of a single-board computer with an integrated Arduino compatible microcontroller. The UDOO board runs an optimized version of Ubuntu(Linux).

V. Implementation

Our UDOO board is equipped with a fully operational Linux operating system. We have chosen to implement the server in C++. C++ has the power of a low-level language and combines this with some levels of abstractions. We wrote a program that signs a number of random messages, along with the time it took to sign the messages. This program acts like a signing server. In order to avoid noise due to network communication, we did the timing inside the server process. This gives the attack a bigger chance of success. The signing program outputs a CSV file with the format 'message, signature, duration', which we can then analyze in order to recover the secret private key.

We implemented the attack itself in python.

The attack is an iterative process where we initially guess that the secret key is one. We then divide the set generated by the server depending if there was any subtraction in the simulated RSA algorithm with the guessed key. We then calculate the average execution time of these to sets. If the average time of the set with subtractions is significantly larger then the set with no subtractions we append the key with one, else we append it with zero. We reuse the same data set for each iteration. At the end of each iteration the attacker signs a couple of messages from the data set with the guessed key, and then checks if it matches with the signature from the server. If this signature is equal to the real signature, we know that we have recovered the secret key, and can stop. This approach lets the attack be agnostic to key length - the attacker does not have to know the length of the private key.

VI. Results

Before attempting to attack the UDOO board, we did a few test runs on our laptop, in order to verify that our code was bug free, and that the approach we selected works.

Since the laptop contains a relatively fast processor and runs a full fledged OS, the extra time of a subtraction would completely drown in temporal noise caused by scheduling etc. To alleviate this, we decided to add a 2 millisecond sleep at each subtraction in the Montgomery product routine. This ensured that the time variance caused by the subtractions was indeed observable. See figure 1 and 2 for an illustration of the observable difference.

With this managed to recover a 33bit key with just 10k messages. This confirmed that our approach worked.

We then signed 10k messages on the UDOO board (with no sleep delay), using a 12 bit private key. We failed to recover the key.

Finally, we signed 1M messages on the UDOO board, using a 669 bit private key. Here too we failed to recover the key.

VII. Conclusion

On a sufficiently slow computer, this attack is definitively viable. However, it seems that if the signing process is running on a full general OS introduces too much temporal noise in the form of scheduling, garbage collection and all the other functions an OS has to do.

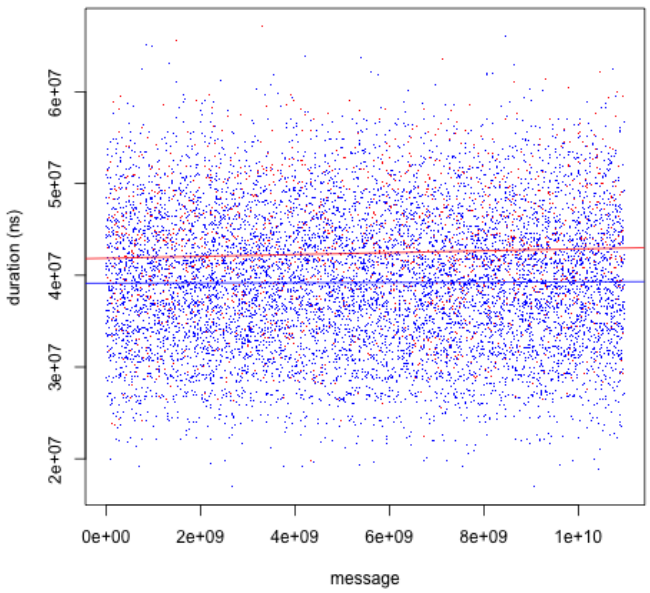

Fig. 1. The data set for one of the bits that was Zero

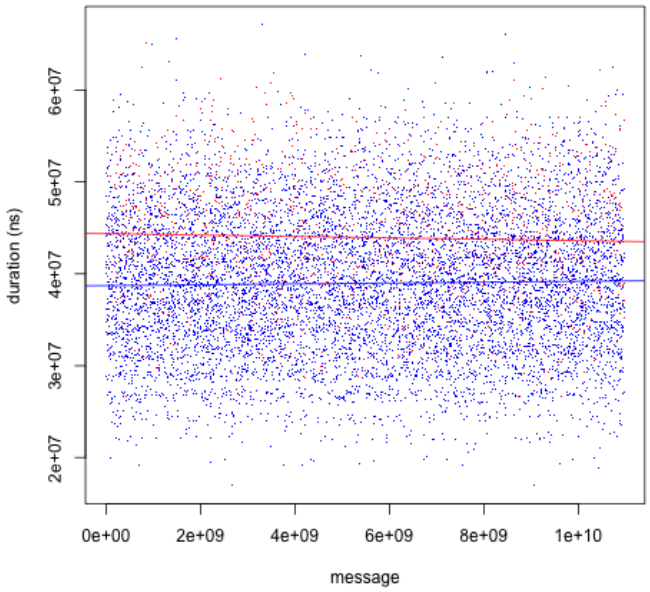

Fig. 2. The data set for one of the bits that was One

It turns out the UDOO board's processor is fast enough that the subtraction operation takes way too little time, so it's not observable whether an extra subtraction was made or not from the data we generated.

Below is an example from the four-day process of signing 1M messages with a large key. The graph clearly demonstrates how OS originated delays affected the timing of certain signatures in a substantial way.

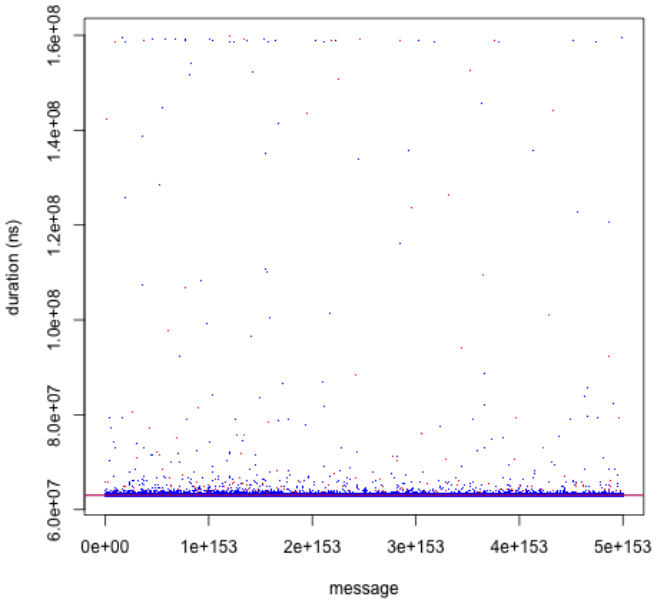

Fig. 3. 1 million signatures. Vertical axis is the time in nanoseconds spent on signing the message. Horizontal axis is the message. Red / blue designates which group the message landed in during the attack.

Performing a successful timing attack on a process running on the UDOO board seems to be an intractable problem, due to the issues mentioned earlier. Our initial plan was to demonstrate that using the Powering Ladder method for RSA exponentiation would work as a countermeasure to the timing attack explained in the introduction. However, since we were unable to replicate the timing attack on real hardware, we saw no reason to run a bunch of messages through the Powering Ladder algorithm. We would have liked to demonstrate the difference compared to the run with a 1 ms sleep, but we see no obvious way to insert a corresponding sleep in the Powering Ladder routine.

VIII. Further work

It would be nice to implement this on the Arduino part of the UDOO board, where it would not run on top of an OS, but just on the bare metal. This should both remove the OS related variations, as well as provide an even slower processor where hopefully the extra delay of the subtraction instruction would be significant enough to exploit.

A. References

- Implementing the RSA - Marc Joye (slides from course)
- Side-Channel Attacks on Cryptographic Tokens, Countermeasures for Preventing Side-Channel Attacks - Marc Joye (slides from course)
- Source code: https://github.com/stoutbeard/crypto