RSA Timing Attack

Chen Yang
Eric Hsieh
Xiaoxi Liu

Advised by:
Vinnie Hu

Abstract

The Rivest, Shamir Adleman (RSA) public-key cryptosystem is the industry standard for protecting both the confidentiality and integrity of sensitive data. It is widely deployed in secure systems involved in political dissents communicating outside of repressive regimes and in online shops handling customer financial data. The strength of RSA relies on the mathematical problem behind it: factoring large integers is extremely difficult. However, its weaknesses lie in that runtime of RSA encryption may reveal a significant amount of information about the secret key. This project will attempt to exploit such a weakness: using runtime measures to recover the private key for RSA cores.

Introduction

Background:

Because RSA is so widely used, it has a reputation as the golden standard for public key cipher based securing communications between two parties. With the recent revelations of government surveillance and the increasing number of large data breaches, protecting data has become a priority. Thus, existing cryptosystems must be evaluated for any weaknesses that can compromise them. While the mathematical problem behind the RSA system is difficult to solve, the weaknesses of RSA lie in the its implementations. These implementations can potentially leak information about the secret key, which will render the cryptosystem useless. The information can be leaked in many ways: by measuring the power consumed, recording the acoustic noise emitted by a fan on top of the encryption core, and in this case: the variation in the computation time.

Application:

To measure the computation time, a customized RSA attack framework is instantiated on a Virtex-7 FPGA. The framework consists of a UART core for serial communication, the RSA implementation, a pseudo random number generator for producing plaintext, and a counter for measuring clock cycles. A counter keeps track of the number of cycles and then sends it back to the MATLAB application. This approach is mainly for a proof of concept, but it could be expanded to include practical applications. For example by using a faster protocol, such as PCI-E, it allows for real time measurements. By shifting to PCI-E, and using the same techniques, it opens the possibility for an actual attack against implementations out in the real world.

The attack framework will have two main components: a MATLAB application to communicate with the FPGA device and analyze the data, the FPGA with the attack framework programmed into it, and the RSA implementation that is under attack.

Development:

The RSA Attack framework has the following requirements:

Hardware	Software
Xilinx Virtex-7 FPGA Board	Vivado Design Suite Used for synthesis of FPGA code
	MATLAB Used for analyzing collected data.

Advantages:

One of the main advantages of the using the FPGA is the speed and the flexibility. It only required about 2 seconds on average to encrypt 8000 plaintext (using a 32-bit RSA core) and send the cycle count to the MATLAB application. In fact, the main speed bottleneck in this RSA attack framework was the UART module.

The RSA Attack framework was also designed to be flexible. It would be trivial to instantiate multiple instances of the same implementation to gather more data in parallel. In addition, different implementations (i.e., RSA cores of different key size) could be swapped in and out.

Technical Material

Data Flow:

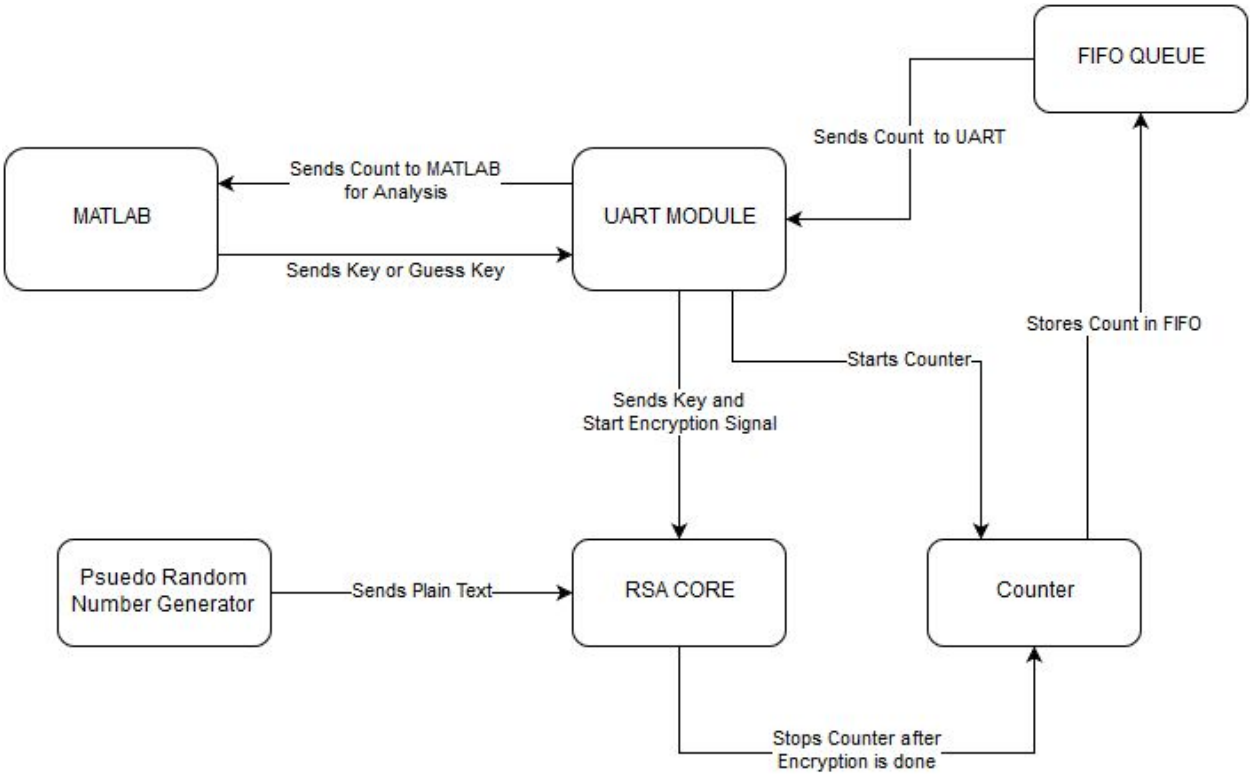

Figure 1: Flow Diagram of RSA Attack Core

MATLAB sends the key, modulus, random number seed in that order to the RSA Attack framework. The “START” command is then sent to the framework. Once it is received, the RSA Attack framework will then use the private key to encrypt 8000 plaintext values, and the cycle count for each entry is reported back to MATLAB.

MATLAB will now send the guess key. Since keys are odd numbers, the least significant bit (Bit 0) is always set to one. Therefore, the attack attempts to guess the second least significant bit (Bit 1). This is guess key is run against the same 8000 plaintext values, and all the cycle counts is reported back to MATLAB.

The variance of each guess key is measured, and MATLAB will use this to guide the next guess, and keeps the bit that leads to larger decrease in variance as the guessed key bit.

Implementation:

The algorithm runs as follows:

Let the cycle counts from the actual key be known as KEY. Let the cycle counts from the key 0001 be GUESS_0001, and the cycle counts of key 0011 be GUESS_0011. First take the variance decrease, VAR of the difference of the KEY and GUESS_0001, that is VAR(KEY - GUESS_0001). Then, find VAR(KEY - GUESS_0011). Whichever guess key yields the lower value, is kept as the guessed key bit and is compared to the next guess.

Another algorithm tested is to compare the values separately: instead of finding variance of the difference between the cycle count of each key, the entire distribution is evaluated together. That is, the variance of the key cycle count VAR(KEY) is compared to the variance of the guess key VAR(GUESS). The difference of VAR(KEY) - VAR(GUESS) should indicate how far from the key the guess key was.

In addition, using 8000 random plaintexts did not yield a normal distribution in the cycle counts. The plaintext generation was changed to use the random seed, and incrementing the value by 1 for 8000 iterations.

Data:

Below is the table of variances of the guess keys and the actual keys. The first column is the guess value of the 4 least significant bits. The row labeled “KEY” is the variance of the cycle count for the actual key. The hexadecimal values at the top at each column is the key that was used.

The table below was used for the VAR(KEY) - VAR(GUESS) analysis.

Table 1: Calculated Variances of least 4 significant bits

Variance of Cycle Counts						
	0x2A948815	0x05F33131	0x0676D829	0x00903AD9	0x6992FCD1	0x7B832F2FD
0001	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0011	2.031522	2.223499	2.239267	2.097927	2.238374	2.256977
0101	3.948747	4.673284	4.484885	4.321725	4.438128	4.550383
0110	5.981293	6.929706	6.641861	6.505288	6.682853	6.947632
0111	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1001	2.031522	2.223499	2.239267	2.097927	2.238374	2.256977
1011	3.948747	4.673284	4.484885	4.321725	4.438128	4.550383

1101	5.981293	6.929706	6.641861	6.505288	6.682853	6.947632
KEY	58.09158	58.132984	58.324112	47.38206	67.912124	67.13123

One of the observed issues was that the guess with the most 1s will a more significant reduction in variance. As a result, the algorithm will always guess a key that was all 1s. The algorithm was tweaked to look at the variance in the difference in the cycle counts, that is VAR(KEY - GUESS). In this case, the guess that yielded the variance closest to the actual key was chosen. This had the same result as the previous case.

Table 2: Variances in the differences of the cycle counts

Variance of Differences in Cycle Counts						
	0x2A948815	0x05F33131	0x0676D829	0x00903AD9	0x6992FCD1	0x7B832F2FD
0001	58.09158	58.132984	58.324112	47.382061	67.912124	67.131230
0011	55.92242	55.861092	56.325534	45.105285	65.541103	64.651431
0101	53.691889	53.497945	54.413323	43.003629	63.404915	61.894892
0110	51.715569	51.392056	52.002646	40.797642	60.779433	59.794349
0111	58.09158	58.132984	58.324112	47.382061	67.912124	67.131230
1001	55.92242	55.861092	56.325534	45.105285	65.541103	64.651431
1011	53.691889	53.497945	54.413323	43.003629	63.404915	61.894892
1101	51.715569	51.392056	52.002646	40.797642	60.779433	59.794349
KEY	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000

In the next attempt, it was observed that the some 4 most significant bits of the key yielded variances that were the same as the actual key (highlighted in yellow). While this did not yield the exact key, it can be used to reduce the search space in a brute force attack.

On average, there were 3 values of the 4 most significant bits that yielded the same variances as the actual key. Also, given that the value has to be odd, thus the total # of possible values remaining is:

$$Number\ of\ Values\ * [\ (2^{Remaining\ Bits})/2]$$

Using 3 as the “Number of Values” and 28 as the remaining number of bits, the number of remaining possible keys is 402653184. This only 10% of the original 2^32 search space, a reduction of 90%.

Discussion:

After a closer look at the RSA algorithm and the RSA code under attack. The runtime difference in the RSA algorithm flow is primarily caused by the additional modular multiplication performed when the key bit is 1.

Such runtime different will always be present for software based implementations, which run through the algorithm sequentially. For the RSA core tested, the additional modular multiplication runs in parallel with a modular square operation, which is performed regardless of the value of the key bit. The modular square operation tends to take longer time to complete than the additional modular multiplication. This will cause the runtime difference of the additional modular multiplication to be invisible due to the parallel implementation of RSA. When guessing a 1 bit, it will lead to more significant reduction in runtime and further a larger decrease in variance.

Table 3: Variance in the cycle counts of the 4 most significant bits.

Variance of Guessing 4 Most Significant Bits					
	0x2A948815	0x0676D829	0x00903AD9	0x6992FCD1	0x7B832F2FD
0x00000001	0.00000000	0.00000000	0.000000000	0.00000000	0.00000000
0x10000001	54.8141	60.4566694	57.85468594	64.3067449	62.5820378
0x20000001	57.5931544	62.9246924	59.53086994	66.4670198	64.7853098
0x30000001	57.5931544	62.9246924	59.53086994	66.4670198	64.7853098
0x40000001	59.0510878	65.032984	61.516856	69.564731	67.6910778
0x50000001	59.0510878	65.032984	61.516856	69.564731	67.6910778
0x60000001	59.0510878	65.032984	61.516856	69.564731	67.6910778
0x70000001	59.0510878	65.032984	61.516856	69.564731	67.6910778
0x80000001	61.4417484	67.807671	64.34192244	72.3228569	70.4520024
0x90000001	61.4417484	67.807671	64.34192244	72.3228569	70.4520024
0xA0000001	61.4417484	67.807671	64.34192244	72.3228569	70.4520024
0xB0000001	61.4417484	67.807671	64.34192244	72.3228569	70.4520024
0xC0000001	61.4417484	67.807671	64.34192244	72.3228569	70.4520024
0xD0000001	61.4417484	67.807671	64.34192244	72.3228569	70.4520024
0xE0000001	61.4417484	67.807671	64.34192244	72.3228569	70.4520024
0xF0000001	61.4417484	67.807671	64.34192244	72.3228569	70.4520024
KEY	57.5931544	55.293496	47.752511	69.564731	67.6910778

Milestones:

Original Milestones:

These were the original milestones for what we had wanted to accomplish on at the beginning quarter.

- Establish Serial Communication with MATLAB Application and UART Core
- Fully Functional RSA Attack Core with Counter, UART and RSA core.
- Launch attack against 32-Bit RSA Keys to Measure performance and gather data.
- Attack More Complex RSA Core
- Replace UART and PCI-E
- Attempt RSA attack using real time measurements instead of clock cycles.

Revisions

One of the biggest challenges that we encountered over the course of this project was the UART module. A large portion of our time was spent working having it talk to the MATLAB application on the computer. At this point it was assumed that everything would work fine, so we went ahead and implemented the entire RSA Attack Core. Again we ran into another issue where it was not functionally properly. Further diagnosis revealed that the core was buffering the data incorrectly. Yet, the timing diagrams from the simulation showed that it was working fine. A significant amount of time was spent trying to resolve this issue, and it had a huge impact on what could be accomplished later. The milestone list was then revised in order to ensure that we could accomplish our core deliverables.

Revised Milestones:

- Establish Serial Communication with MATLAB Application and UART Core
- Fully Functional RSA Attack Core with Counter, UART and RSA core.
- Add an extra RSA core to run attacks in parallel.
- Launch attack against 32-Bit RSA Keys to Measure performance and gather data

Due to the difficulties of getting our RSA core working, we were provided with an existing RSA core to do collect our data. In addition, we were unable to fully complete the GUI, but we have it's necessary functions spread out across multiple files.

Conclusion:

From this project, it can be shown that by measuring the time needed for calculations, an attack can be launched against RSA. It was observed that observing the least significant bits of the key did not yield any

meaningful results. This was caused due to the the parallelism built into the RSA core, which masked some of the timing information. Consequently, this is one potential way to protect against timing attacks.

However, by observing the most significant bits, it is possible to reduce the search space. In addition the search space can be reduced even further by calculating all the prime numbers and using those as all candidate space.

However, there are some modifications that would increase the impact of this experiment. Primarily, this experiment was done using only 5 RSA keys, and to have more meaningful results, a larger sample size of keys will be needed. In addition, 32-bit RSA keys are almost ever used as they are not secure enough. If the key length was increased to 1024-bits, and the same pattern was observed, then it would have more impact as 1024-bits is the minimum number of bits used in industry.