**Research Summary and Questions Raised: RSA Python Encryption**Fall 2019 | Aron Schwartz

**Introduction** The mathematical foundations of RSA encryption were first proposed in 1977 in a paper by Ron Rivest, Adi Shamir, and Leonard Adleman[1]. Shortly after this initial paper was released, another paper was published describing a peculiar flaw in the RSA algorithm—the so called occurrence of “Fixed Points”[2]. In this paper by Blakely and Borosh, a phenomenon is described in which RSA encryption can sometimes fail to “hide” an encrypted integer for certain choices of parameters. In other words, the mathematical result of encrypting an integer (which should by all means be different than the original integer) is identical to the original integer…as if encryption did not occur at all. This so called phenomenon of “fixed points” (also referred to as “holes” in this report) has been well known since this paper was released in 1979. However while the existence of this phenomenon is well known, methods correlating the relationship between the 7 input parameters (also referred to as a “septuple”) and the resulting likelihood of fixed point occurrence has been lacking. Current methods rely on manually checking for fixed points by encrypting a set of integers and observing the results…however this method is computationally expensive and inefficient, particularly if exhaustive hole analysis is desired ( “exhaustive” referring to an explicit check for fixed point occurrence across the set of encryptable integers, i.e. 0 to N-1).

The motivations of the research described in this report are rooted in the search for a computationally cheap and mathematically reliable check for fixed point occurrence. This is done by analyzing the relationship of initial parameter choice compared against the resulting “transparency” for given initial parameters utilizing python as a data collection tool. A detailed overview of the work performed, conclusions reached, and questions raised can be found in the following sections.

A link to the repository containing all code and data that is referenced in this report can be found at the following link:

<https://github.com/aronjschwartz/RSA_Encryption.git>

**Phase 1—RSA Algorithm Study and Python Implementation**

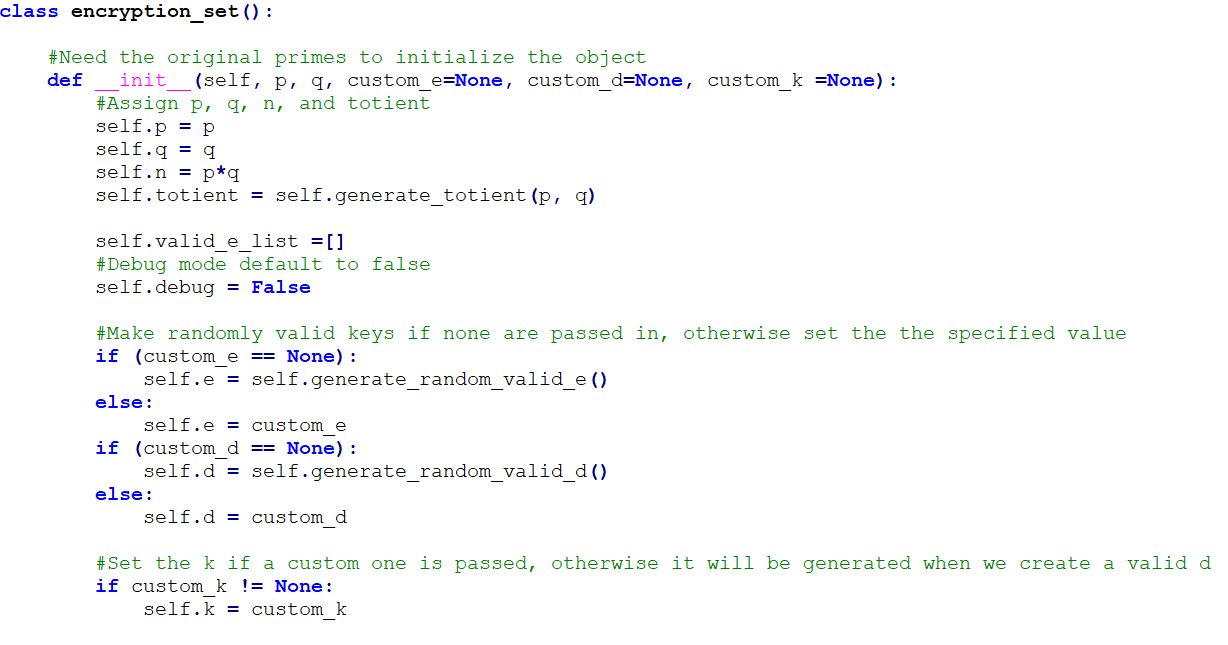
The first phase of research involved a general overview and study of the RSA Encryption algorithm, followed by implementation of the algorithm in Python. Initially, this process was qualitative in nature beginning with an analysis the Master’s Thesis by Behnaz[3]: “Finding Cases of Ciphertext equal to Plaintext in the RSA Algorithm”. A brief overview of the required parameters and their role in the RSA algorithm can be seen below:

**(NOTE:** This report does not cover the mathematics behind the RSA algorithm in detail. For a concise mathematical overview, see Behnaz Thesis)

* **P and Q:** Two initially chosen prime numbers which govern other parameter choices
* **N:** Product of P and Q
* **Totient**: Product of (P-1)\*(Q-1)
* **E:** Encryption exponent, part of the public key (E, N)
* **D, K:** Parameters related to decryption

**Phase 1 Goal:** **Encrypt a string to ciphertext, and de-encrypt it back to plaintext, thus demonstrating proper setup of the RSA mathematics and python code structure.**

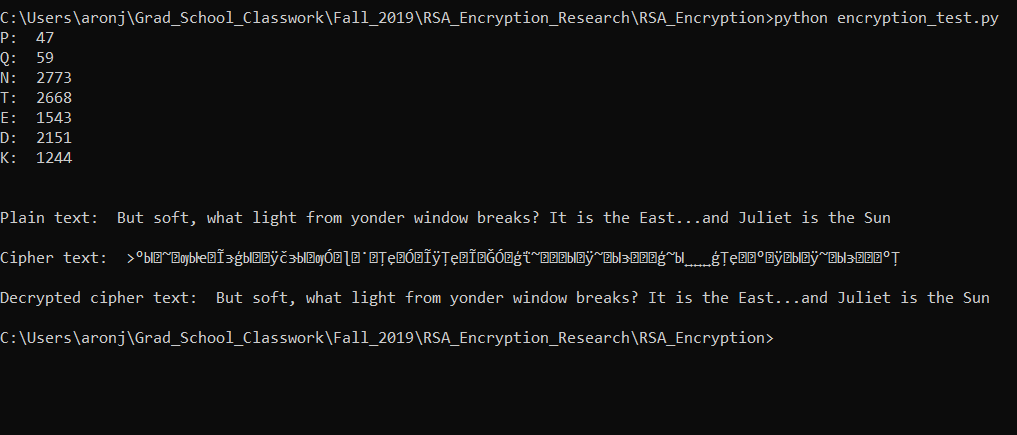
The design decision to implement the algorithm using Python’s OOP (Object Oriented Programming) framework was chosen for code readability and scalability. All functionality needed to encrypt and decrypt integers is comprised in the file **encrypt.py**. A short snippet of this code can be seen below:



**Figure 1:** Encrypt.py implements an OOP approach to RSA algorithm

As shown in the screenshot, initialization of the object ‘encryption\_set’ accepts at a minimum the initial P and Q values. If no other parameters are provided, the code will find valid E, D, and K values to complete the set. However, more pragmatically it allows the user to choose all parameters of the set if they wish. This flexibility allows creation of specific “encryption objects” to exist as unique python objects with their own unique initialization parameters. More importantly, this flexibility allows easy creation and distinction arbitrary numbers of encryption objects.

Once the code was implemented, various septuples were created and tested against basic strings. A screenshot demonstrating the output of **encryption\_test.py** can be seen below:

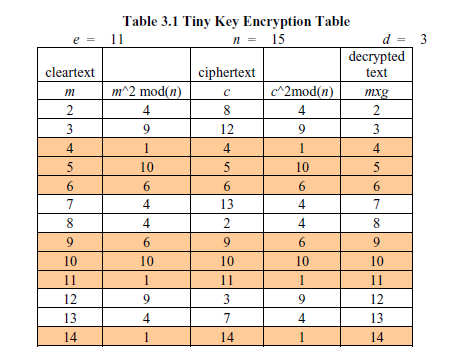


**Figure 2:** Output of encryption\_test.py demonstrating algorithm functionality

**Phase 2: Focus shift to “holes”- Recreating the “Tiny Key Encryption Table” to verify hole detection**

After functionality of the RSA algorithm was verified, the focus was shifted toward fixed points and the ability to detect them. Ensuring proper “hole searching” functionality on a pre-known solution was critical to expanding the work to further phases, thus the motivation for recreating the chart from the Behnaz thesis.

**Phase 2 Goal: Recreate the “Tiny Key Encryption Table” from Behnaz thesis thus demonstrating proper functionality of the “hole searcher” algorithm**



**Figure 3:** The “Tiny Key Encryption Table” showing hole occurrences highlighted in orange

The chart shown in the screenshot demonstrates the concept of holes visually. With a given choice of parameters, all values from 2 to N-1 are encrypted. The rows highlighted in orange represent a “hole”, as evidenced by the fact that the cipher text is identical to the plaintext. In summary, this chart was recreated with simple print statements validating the ability to calculate and analyze all holes for chosen initial parameters.

**Phase 3: Large scale data collection mapping input parameter choice to hole occurrence**

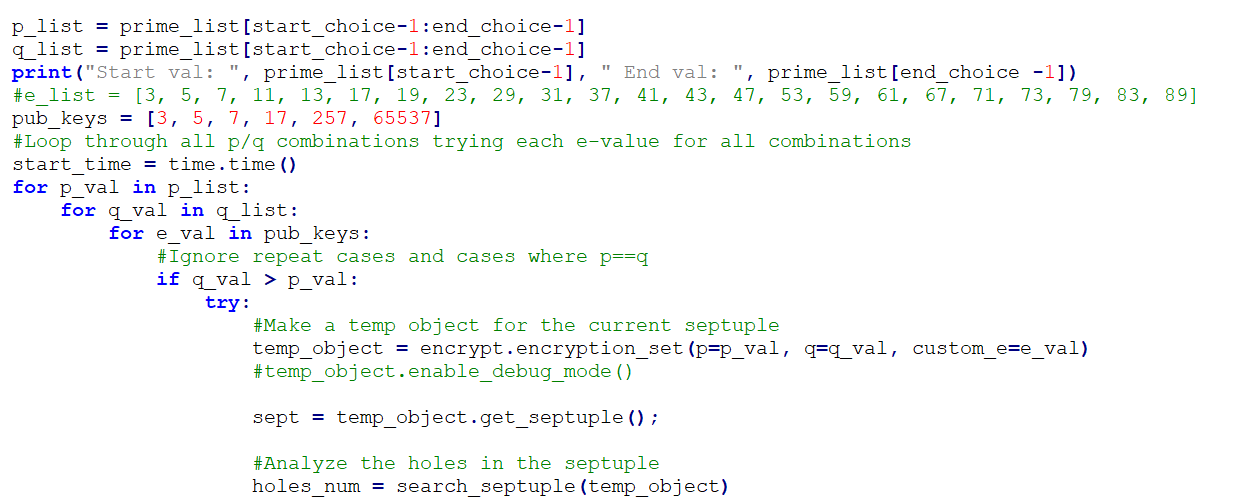
Once the ability to find holes via code was confirmed, open ended data collection was initiated without regards to a particular pattern being searched for**.** The goal was to generate statistically significant numbers of septuples and analyze the occurrence of holes, with the hopes of a explorable relationship emerging.

**Phase 3 Goal: Systematically generate hundreds of parameter combinations searching for discernible or explorable relationships between input parameter choice and the resulting relative occurrence of fixed points**

To accomplish this, code was written to generate various septuple combinations and encrypt all possible values (0 to N-1) to exhaustively check for holes. Since this process is computationally expensive for large values of N, lower initial primes (P and Q) were chosen for easier computation. An overview of how the input parameters were chosen and generated can be seen in the following sections.

**Choosing P and Q**

As mentioned, the higher the value of N, the more computation time is needed to analyze all values from 0 to N-1. To accommodate this fact, P and Q were chosen to be all possible combinations of prime numbers in the first 100 primes, **approximately 25,000 different combinations**. This was accomplished by downloading a text file of the first 1,000,000 primes and doing some simple parsing, as well as some additional logic to prevent repeated combinations of P and Q. A screenshot of this code illustrating the logic flow can be seen below.



**Figure 3:** Code allows choice of starting and ending index in first million primes

As shown, the code utilizing for-loops to generate all possible prime combinations, with the lowest prime being 3 and the largest prime being 541. For example, the first few combinations of P and Q and 3/5, 3/7, 3/11…ending with 521/523, 523/541.

**Choosing E (and by extension, the public keys (E, N)):**

The choice of E values for this analysis was chosen to represent the six most commonly recommend values of E following independent research on stack overflow and encryption forums: specifically, the values {3, 5, 17, 257, 65537). The public recommendation of these E-values is rooted in the fact that these values only have two hot bits, thus making modular exponentiation faster as well as easing the computational burden on the end-user (since huge exponential calculations is often required). The following online forum posts help illuminate the common use of these keys in the security community:

<https://security.stackexchange.com/questions/2335/should-rsa-public-exponent-be-only-in-3-5-17-257-or-65537-due-to-security-c>

<https://www.di-mgt.com.au/rsa_alg.html>

For the purposes of this research, these values were used as the values of E provide consistency and alignment with commonly used values in the real world. All P/Q combinations as described above were combined with each of this values of E to ultimately **generate 24256 unique sets.**

**(NOTE:** The choice of D and K are not mentioned, since they are not relevant for this work. After all, one only needs to encrypt to observe the existence (or lack of existence) of a hole.)

**Phase 4: Analysis of obtained data**

After large amounts of data were generated as described in the previous section, the data was analyzed and reviewed.

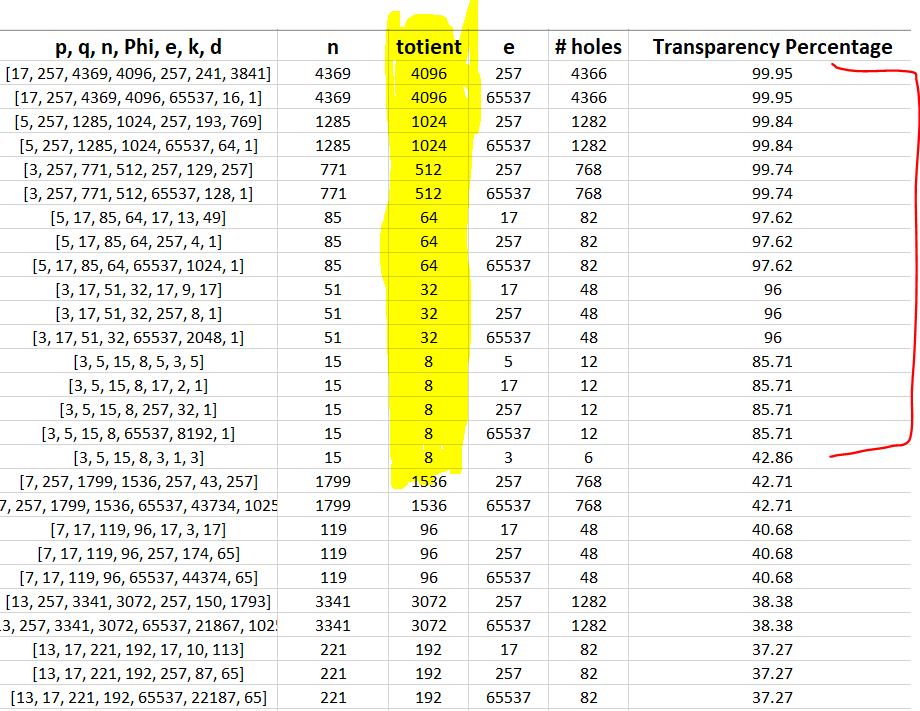
**Phase 4 Goal: Determine any discernible, repeatable, or describable relationship between the choice of input parameters (P, Q, N, Totient, E) chosen to create the encryption object, and the resulting occurrence of holes**

For this analysis, the term “transparency” is often used to describe the relative strength of a septuple. This refers to the degree by which an encryption set has (or does not have) holes. For example, a set that has 100% transparency means that all possible values that can be encrypted for that set are holes (the worst possible scenario). Likewise, 0% transparency would indicate no holes, although this is not possible in reality due to the existence of the six principal holes[3].

In summary, 0% *opacity* is impossible, however 100% transparency is certainly possible and was in fact observed in some cases.

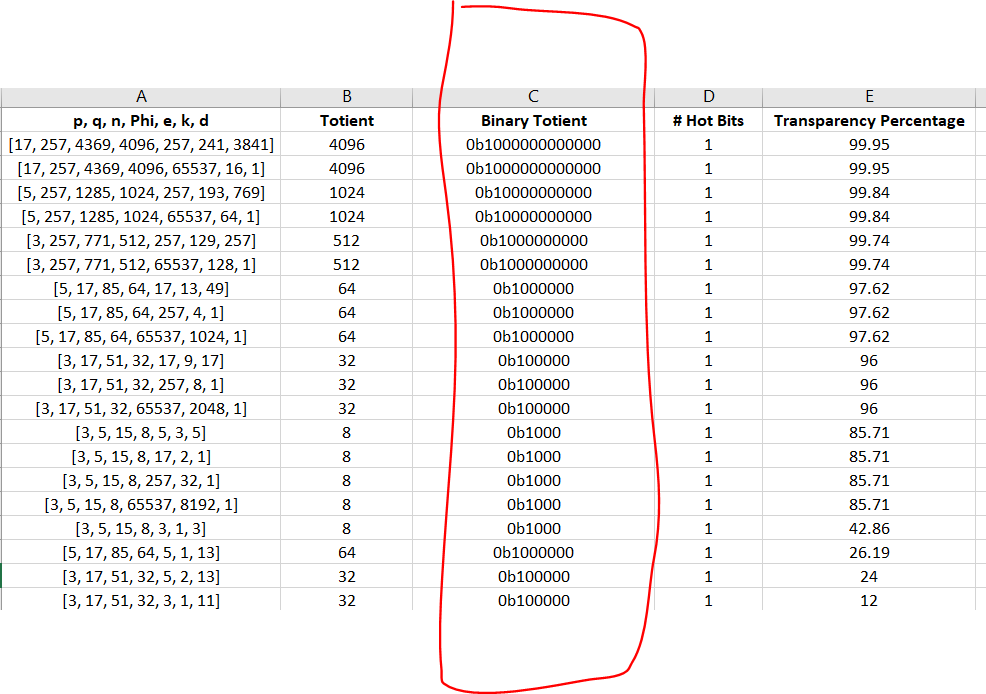
**Observation 1:** The highest transparency septuples are associated with totients that are a power-of-two, or near a power-of-two

Upon analysis, a mathematical commonality among the highest transparency septuples was found in the totient parameter (Totient = (P-1)\*(Q-1)). The following screenshots and charts further illustrate the peculiar pattern. The screenshot below is taken from the file “primes\_1\_to\_100\_holes.csv”, and can be found in its entirety in the given repository.

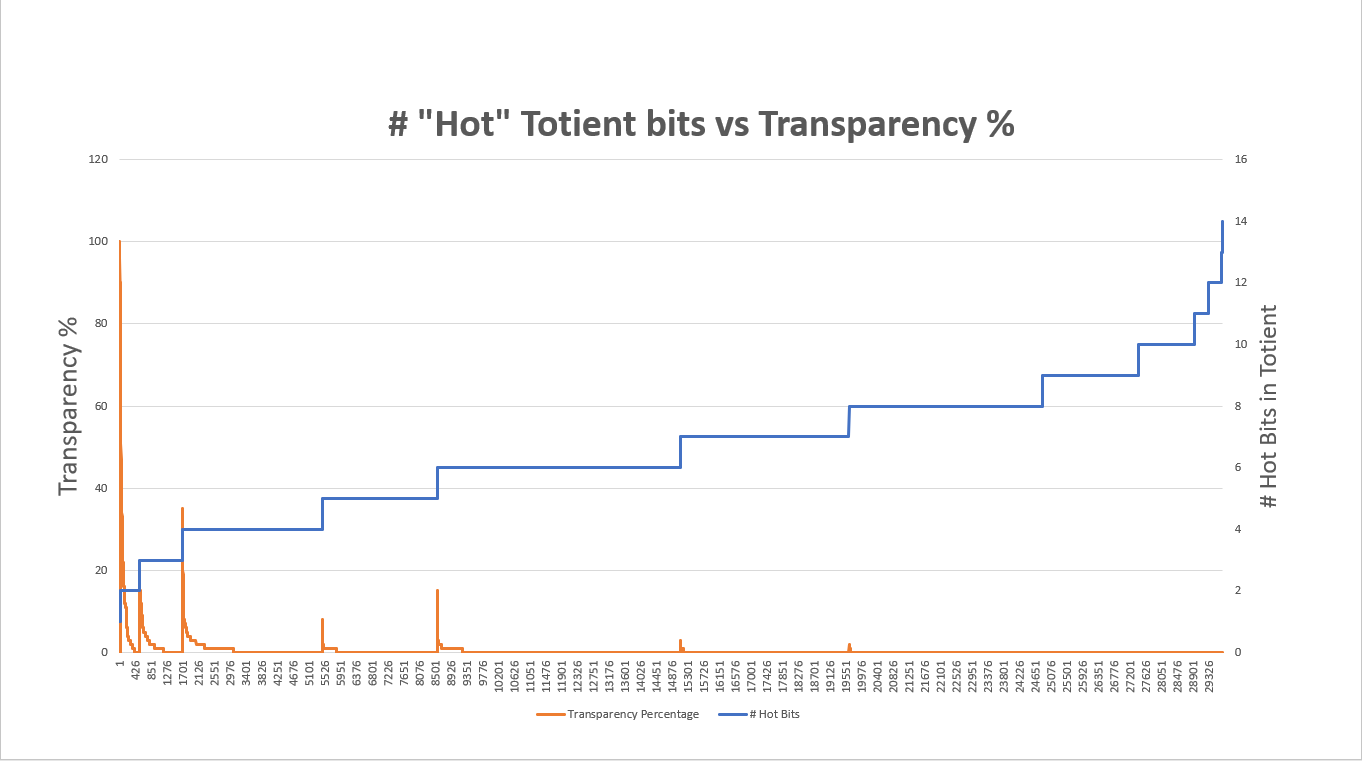


**Figure 4:** The highest transparency septuples had totients equal to a power of two

The screenshot shows the highest transparency septuples after sorting the data by transparency value. One can see the commonality of the totients all being a direct power-of-two, or a multiple of a power of two for the highest transparency sets. Upon this discovery, some charts were generated in an attempt to further understand the pattern. The code was also re-spun to output the binary representation of the totient as an additional parameter. Screenshots of this data re-visualization and a representative graph illustrating the trend across the data set can be seen below.



**Figure 5:** Data from figure 4 re-spun to show bit pattern and totient hot bits



**Figure 6:** Graph showing relationship between hot bits in the totient and exponentially decreasing transparency

The blue line in the graph demonstrates increasing number of hot bits of the totient vs the relative transparency % in orange. As evident by the graph, the peak transparency values occur when the totients are direct powers of two. As the number of hot bits in the totient increases (as by extension, as the totient trends away from a direct power of two), the peak transparency decreases. Each subsequent “bit zone” has a lower and lower peak transparency. By the time we reach about 8-9 hot bits in the totient, transparency percentages are nearly negligible (transparency < 0.1%).

A few specific septuples and their associated data can be seen below to further visualize these findings.

1 Hot Bit Examples:

1. **Totient = 512**

* **Binary = 0010 0000 0000**
* **Septuple = [3, 257, 771, 512, 257, 129, 257]**
* **Transparency = 99.74%**

1. **Totient = 4096**

* **Binary = 0001 0000 0000 0000**
* **Septuple = [17, 257, 4369, 4096, 257, 241, 3841]**
* **Transparency = 99.95%**

1. **Totient = 64**

* **Binary = 0100 0000**
* **Septuple = [5, 17, 85, 64, 65537, 1024, 1]**
* **Transparency = 97.62%**

2 Hot Bit Examples:

1. **Totient = 192**

* **Binary = 1100 0000**
* **Septuple = [13, 17, 221, 192, 17, 10, 113]**
* **Transparency = 37.27%**

1. **Totient = 2560**

* **Binary = 1010 0000 0000**
* **Septuple = [11, 257, 2827, 2560, 257, 180, 1793]**
* **Transparency = 27.18%**

1. **Totient = 1152**

* **Binary (0100 1000 0000)**
* **Septuple = [7, 193, 1351, 1152, 257, 143, 641]**
* **Transparency = 14.22%**

3 hot Bit Examples:

1. **Totient = 448**

* **Binary = 0001 1100 0000**
* **Septuple = [17, 29, 493, 448, 65537, 37596, 257]**
* **Transparency = 16.67%**

1. **Totient = 1792**

* **Binary = 0111 0000 0000**
* **Septuple = [17, 113, 1921, 1792, 257, 147, 1025]**
* **Transparency = 14.90%**

1. **Totient = 114688**

* **Binary = 0001 1100 0000 0000 0000**
* **Septuple = [257, 449, 115393, 114688, 257, 183, 81665]**
* **Transparency = 14.47%**

**…**

13 Hot Bit Examples:

1. **Totient = 131028**

* **Binary = 0001 1111 1111 1101 0100**
* **Septuple = [359, 367, 131753, 131028, 17, 15, 115613]**
* **Transparency < 0.01%**

1. **Totient = 259956**

* **Binary = 0011 1111 0111 0111 0100**
* **Septuple = [499, 523, 260977, 259956, 3, 1, 346608]**
* **Transparency < 0.01%**

14 Hot Bit Examples:

1. **Totient = 253692**

* **Binary = 0011 1101 1110 1111 1100**
* **Septuple = [487, 523, 254701, 253692, 17, 16, 238769]**
* **Transparency < 0.01%**

1. **Totient = 262044**

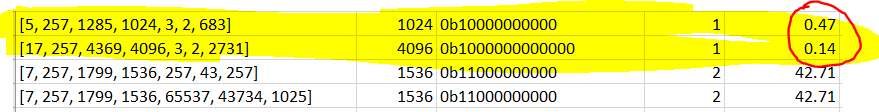
* **Binary = (0011 1111 1111 1001 1100)**
* **Septuple = [503, 523, 263069, 262044, 65537, 34659, 138581]**
* **Transparency < 0.01%**

**Questions raised from the totient-pattern observation**

Upon the discovery of the demonstrated totient relationship, a series of questions were postulated regarding the nature, repeatability, and significance of the described pattern. Some fundamental questions that arose from this discovery and the progress toward answering those questions can be seen below.

**Question 1:** **Does the “pattern” always hold?** In other words, does the hot-bit to totient relationship demonstrated in the previous section always hold?

**Answer:** **No, the pattern is not deterministic and does not hold in all cases.** Cases in which power-of-two totients and/or nearly power-of-two totients result in acceptable encryption (no holes other than principle) were found. A few screenshots can be seen below demonstrating this:



**Figure 7:** Low hot-bits in the totient does not always mean poor encryption

The screenshot shows two septuples with “1 hot bit” totients that have acceptable transparency percentages of 0.47% and 0.14%. Other occurrences of “low hot bit” totients having acceptable transparency values were also found in the data alongside this example.

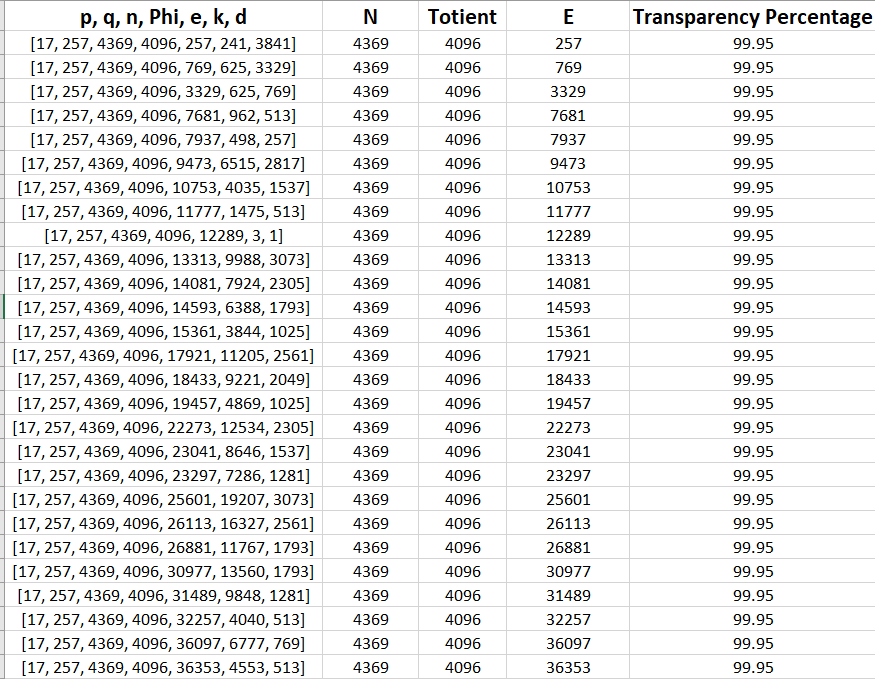
**Question 2:** It can be seen that the totient alone is not enough to make a reliable prediction with regards to transparency**. If the totient alone is not enough to predict transparency, what additional parameters play a role?**

It is from this question that the final phase of the work was initiated, which can be found described in detail in the next section.

**Phase 5: Targeted analysis and the emergence of common transparency profiles**

In order to further tease out and understand the observed pattern in the totient and its relationship to transparency, various septuples from the original data set (“primes\_1\_to\_100\_holes.csv”) were isolated and tested against different choices of E. As a reminder, the original data was analyzed only for the common public keys of 3, 5, 17, 257, and 65537. However the following analysis involves an analysis of many different values of E with the goal being to observe how the transparency percentages change while only varying this parameter. The values of E were chosen to be all prime numbers between 3 and 65537, such that coprimality conditions are satisfied[3]

The first septuples analyzed were taken from the highest transparency sets in the original data, specifically with totients that were “1 hot bit”, or a direct power of two. A screenshot demonstrating the output of a septuple from this set [17, 257, 4369, 4096, 257, 241, 3841] sorted by largest-to-smallest transparency can be seen below. Data of this format is referred to in this report as a “transparency profile”, showing the value of E used and the associated transparency for all values of E.



**Figure 7:** Snippet of transparency profile for septuple with Totient = 4096

As shown in the screenshot, the different transparency values with regards to varying values of E is seen. As data in this manner was generated, commonalties were observed for certain septuples, leading to observation 2:

**Observation 2:** Certain septuples have identical “transparency profiles”, indicated by identical transparency percentages for a given value of E

It was found that some septuples had identical results with regards to the value of E chosen and the resulting transparency percentages. For example, consider the following three septuples taken from the “1 hot bit” set:

**Septuple 1:** [3, 257, 771, 512, 257, 129, 257]

**Septuple 2:** [5, 257, 1285, 1024, 257, 193, 769]

**Septuple 3**: [17, 257, 4369, 4096, 257, 241, 3841]

The transparency profiles for these three septuples can be seen below:

**References**

1. R. L. Rivest, A. Shamir, and L. Adleman. 1978. *A Method for Obtaining Digital Signatures and Public-key Cryptosystems,* Commun. ACM, Vol. 21, no. 2, pp. 120–126. DOI:[10.1145/359340.359342](https://doi.org/10.1145/359340.359342)

2. G. Blakley and I. Borosh. 1979. *Rivest-Shamir-Adleman Public Key Cryptosystems do not Always Conceal Messages*, Computers and Mathematics with Applications, Vol. 5, Issue 3, pp.169-178. DOI: <https://doi.org/10.1016/0898-1221(79)90039-7>

3. Sadr, Behnaz. 1992. *Finding Cases of Ciphertext equal to Plaintext in the RSA Algorithm,* School of Electrical and Computer Engineering, OSU. DOI: <https://hdl.handle.net/11244/10269>