# Implementing RSA and DES Cryptographic Algorithms in an Academic Environment

## Using the Python Programming Language

Zach Gilmer
Missouri State University
1932 Wallamaloo Lane
Wallamaloo, New Zealand
@live.missouristate.edu

James Hibben
Missouri State University
419 W. Loren St.
Springfield, MO 65807
Hibben87@live.missouristate.edu

## ABSTRACT

Implementing a cryptographic algorithm is generally considered to be a difficult task that should be left to the experts. While studying computer security, however, it is important to understand the complexity behind them, as even implementing a simple version of any particular algorithm is a challenge. As such, we elected to implement a simple (and insecure) RSA asymmetric-key algorithm and a DES symmetric-key algorithm.

## Categories and Subject Descriptors

H.4 [**Cryptography**]: Computer Science; D.2.8 [**Software Engineering**]: Academic

## Keywords

RSA, DES

## 1. BACKGROUND AND REVIEW

Introduction (obviously).

## 2. IMPLEMENTING DES

## 3. IMPLEMENTING RSA

While RSA is a simple asymmetric-key algorithm to implement, the complexity and computations necessary to run also make it a time-consuming algorithm to use; even generating keys can take an extraordinary amount of time. The algorithms used to generate keys run in $O(n)$ time, but are typically used in conjunction with large-magnitude keys; key sizes on the order of $10^{100}$ are not uncommon, and often take several seconds to generate on modern processors. Encryption and decryption, however, are significantly more time-consuming when increasing the size of the input; part of the encryption and decryption process runs in $O(n^2)$ time.



### 3.1 Key Generation

When generating RSA keys, three keys in total are needed: Public keys $(n, e)$, used for encryption, and the private key $d$, used for decryption. $n$ is usually generated first by randomly generating prime integers $p$ and $q$; these are multiplied together, then applied using Euler's totient[1] function $\phi((p-1)(q-1))$; this totient is then used as a base from which the next prime integers, $e$ and $d$ are generated. Both $e$ and $d$ must be less than the totient, and one is selected to have the additional requirement of being coprime to the totient; for our implementation, we selected $e$ as the coprime. Once $e$ has been generated, then $d$ can be generated using the modular multiplicative inverse function:

$$e \equiv d^{-1} \pmod{\phi(n)}$$

.

### 3.2 Encryption and Decryption

## 4. CONCLUSIONS

## 5. REFERENCES

[1] Feb 2016. Page Version ID: 705422305.