FC1:
A Powerful, Non-Deterministic, Symmetric Key Cipher

Michele Fabbrini[*]

May 8, 2022

Abstract

In this paper we describe a symmetric key algorithm that offers an unprecedented grade of confidentiality. Based on the uniqueness of the modular multiplicative inverse of a positive integer a modulo n and on its computability in a polynomial time, this non-deterministic cipher can easily and quickly handle keys of millions or billions of bits that an attacker does not even know the length of. The algorithm's primary key is the modulo, while the ciphertext is given by the concatenation of the modular inverse of blocks of plaintext whose length is randomly chosen within a predetermined range. In addition to the full specification, we present a working implementation of it in Julia Programming Language, accompanied by real examples of encryption and decryption.

1 Introduction

In a symmetric key encryption scheme, a single key is used for both encryption and decryption. An algorithm can be considered safe if the only way to guess the key is to explore all the possibilities given by the different combinations of zeros and ones. This is called a "brute-force attack" and under certain circumstances it can be very difficult, if not impossible, to implement. A 256-bit key (the length used by the current AES encryption standard) is at present considered “unbreakable” even by the next generation of quantum computers. So it appears that approved standards can ensure a good level of confidentiality for many decades to come. Neverthless, if we look at some structural aspects of them, we can find some relevant weaknesses that could jeopardize the security of encrypted data in light of some new challenges that we are likely to face in a near future. But before going into the technical details of the weaknesses, it is useful to dwell on the implicit hypothesis that underlies the alleged security of encryption standards. In fact, they are designed to instantly transfer encrypted data between two different points in space. But if we consider sending data to a different point in time, then the algorithms used would be perhaps inadequate to protect the confidentiality of the original text. For example, suppose Alice wants to transmit an AES-256 encrypted message to Bob who will use the symmetric key to decrypt it in 50 years. How can Alice be sure that the technological development of the coming years will not lead to the ability of calculating 2^{256} in a reasonable time, so making the key in Bob's possession useless? Now let's see the structural aspects that may compromise the safety of the accepted standards. Current standards have four main aspects in common. The first two are related to the key: its length is known and does not exceed 256 bits. The last two relate to the algorithms: they are deterministic and convert a fixed block of plaintext into a ciphertext block of the same length. An algorithm is deterministic if a given plaintext always produces a given ciphertext. These four points are generally considered irrelevant and do not raise security concerns. In our opinion, there are however some scenarios that could change the rules of the game. One of these, for instance, relates to the prospect of an upcoming world war, or at least of a long period of involution in the reciprocal relations among states. A tragic war is hitting Europe. Again. Diplomatic relations are cooling down and scientific discoveries become military secrets. In this context of separation and conflict how can one be sure that a certain technological level has not already been reached by the adversary? So, for example, how can we be sure that no one in the world is able to test

[*] Mathematics Department, University of Pisa. Email: 13518203@studenti.unipi.it

2^{256} different possibilities in a reasonable time? Since a brute-force attack implies in any case the use of computing power, it may be a good idea "to raise the bar", thus passing from keys of 256 bits to keys of hundreds of thousands or millions of bits. Likewise, it can be helpful not to publicly disclose the key length. But in the face of considerable computational capacity, the use of a larger key may not be sufficient. It is necessary to break the mold and move without delay from deterministic to non-deterministic algorithms, making the relationship between input and output more complex and unpredictable. These are the main lines that have guided the construction of the FC1 algorithm, the first one we made public in a class of algorithms designed to face the big challenges of our future.

2 Specification

2.1 Modular Multiplicative Inverse

Definition

For a positive integer n, and $a \in \mathbb{Z}$ we say that $a' \in \mathbb{Z}$ is a multiplicative inverse modulo n if

$$aa' \equiv 1 \mod n$$

It can be proven [1] that:

1. a has a multiplicative inverse modulo n if and only if a and n are relatively prime
2. if a' exists, then it is unique

Computation

There are various methods to compute the inverse modulo n in a polynomial time [2] [3]which, if implemented in languages like Julia[1] having built-in support for Arbitrary Precision Arithmetic, make it possible to calculate a' in a few fractions of a second even for numbers with hundreds of thousands of digits (see Appendix B).

2.2 Description

Basic concept FC1 essentially relies on the uniqueness of the modular multiplicative inverse of a positive integer a modulo n and on the fact that it can be calculated in a polynomial time. Here the modulo is the main key which, due to the algorithm's design, can be any positive integer, while the ciphertext is the modular multiplicative inverse. The plaintext, once tagged with a hash, is divided into blocks, the length of which is chosen by a random number generator, converted into ciphertext and sent over an insecure channel.
Keys Keys to be kept secret and transferred over a secure channel are primary key (the modulo), and secondary key. The latter represents the length of a random string that is placed at the beginning of the ciphertext.

2.2.1 Encryption

Hash The very first operation that is performed is the computation of a hash of the plaintext using the SHA-256 function. This tag is then appended at the end of the text. The purpose is to ensure the integrity of the data transmitted. We denote the plaintext with the final tag by 'tplain':

$$tplain = plaintext \| hash$$

[1] With origins in the Computer Science and Artificial Intelligence Laboratory (CSAIL) and the Department of Mathematics, Julia is a programming language created in 2009 by Jeff Bezanson, former MIT Julia Lab researchers Stefan Karpinski, and Viral B. Shah, and professor of mathematics Alan Edelman. The Julia programming language is a flexible dynamic language, appropriate for scientific and numerical computing. Julia provides software support for Arbitrary Precision Arithmetic, which can handle operations on numeric values that cannot be represented effectively in native hardware representations, but at the cost of relatively slower performance. To allow computations with arbitrary-precision integers and floating point numbers, Julia wraps the GNU Multiple Precision Arithmetic Library (GMP) and the GNU MPFR Library, respectively. In an APA application the size of the integer is limited only by the available memory. Website: https://julialang.org/

Ciphertext initialization With the value of the secondary key, a random string is created which we denote by 'startpad'. This is the initial ciphertext:

$$c = startpad$$

Fencrypt A main function named 'fencrypt' has the task of controlling the flow, switching between different sections of the algorithm in relation to a certain threshold value of the length of the tagged plaintext that still remains to be encrypted. The threshold is fixed at 1.5 times the length of the modulo.

Frand In the first part of the algorithm fencrypt calls a random number generator in a given range, which we denote by 'frand'. The generated random integer represents the length of the i-block of tagged plaintext to be encrypted. We denote by $|modulo|$ the length of the modulo. The frand function generates a random value between 1 and $|modulo| - 3$. From the length of the modulo, 3 bits are subtracted to define the upper limit of the random function because 2 bits space is used to append the leading and trailing 1 (see next point 'Fintgen'). Moreover, since we want that the integer, whose modular inverse we are going to calculate, is less than the modulo, another bit is dropped:

$$1 \leq frandvalue \leq |modulo| - 3$$

Fintgen A leading and a trailing '1' are appended at each chunk of tagged plaintext whose length is randomly selected by frand function. The leading 1 is meant to make sure that the input of the function computing the modular inverse is a positive integer since the block could start with '0'. The trailing '1' serves to prevent the algorithm from blocking in the case of an even modulo and a tagged plaintext to be encrypted containing a long row of 0's. We denote by $tplain_i$ the i-block of tagged plaintext; then is:

$$input_i = 1 \| tplain_i \| 1$$

Finv Once the input has been prepared, it is possible to attempt to compute the modular inverse using the 'finv' function. If the input and the modulo are not coprime, finv cannot produce a result and it calls the main function fencrypt which calls frand again in order to try with a different random integer. Else, if they are coprime, the modular multiplicative inverse is computed in a polynomial time and passed to the next step.

Fblockgen If the modular inverse is computable, a function called 'fblockgen' comes into play comparing the modulo length with that of the modular inverse generated by finv. If the lengths are the same, fblockgen does not modify the string:

If $|modulo| = |finvvalue|_i$

$$output_i = finvvalue_i$$

Otherwise, if the modular inverse length is less than modulo length, fblockgen adds one or more leading zeros so that the lengths match:

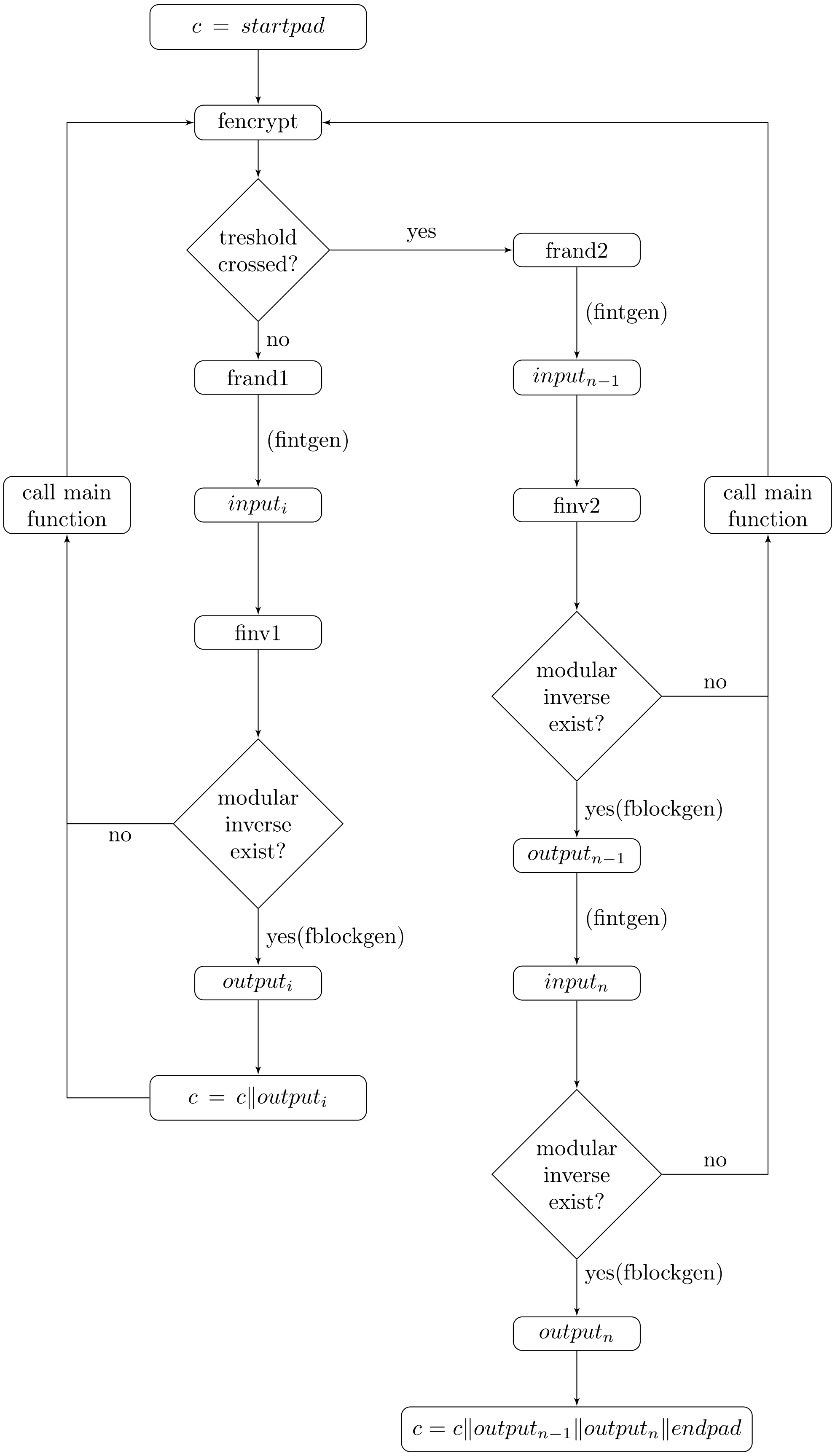
If $|modulo| > |finvvalue|_i$

$$output_i = 0..0 \| finvvalue_i$$

Final step of the first part The block created by fblockgen is concatenated to the existing ciphertext and the main function fencrypt is called.

Second part: ciphertext finalization When threshold is crossed, the finalization functions are called. They have the task of simultaneously calculating the last and the second-last block of ciphertext. This design solution is necessary to prevent the case the modular inverse does not exist for the last portion of tagged plaintext, with the consequence of blocking the whole encryption process. The last step involves adding a random final padding whose length must be less than modulo length. This final padding, that we call 'endpad', is actually a third key that we can consider inferred from the other two. It is automatically added by the encryption algorithm. In the subsequent decryption phase, the algorithm will recognize it as its length is less than that of the modulo and finally it will discard it without attempting to decrypt it.

Encryption flowchart



2.2.2 Decryption

We omit a complete description of the decryption algorithm since it is trivial, referring to Appendix A for a code example in Julia. Note that, once the whole tagged plaintext has been decrypted, it is checked, through the hash function, that the final tag is correct and that therefore the integrity of the data is not compromised.

2.3 Recommended parameters set

Primary key We recommend a minimum length of 501 bits. At the same time, we encourage the use of 50.000-100.000 bits keys to fully exploit the potential offered by the algorithm. To maximize the speed we suggest the use of a modulo having as factors non-trivial prime numbers. If, on the other hand, the aim is to create further problems for a potential attacker, we recommend the inclusion of some trivial factors such as 3, 5, 11 and so on. Remember that you can safely use an even modulo without absolutely slowing down the algorithm.
Secondary key It has no upper limit and can even be 0.

2.4 Security

The minimum recommended primary key length we have seen is 501 bits. The maximum length is instead not defined because it depends on the limits of the system on which the algorithm is run. In our tests we went as far as keys of over one Gigabit, which means a length of over a billion bits. Now, if by hypothesis the attacker knew the length of the key, the startpad was zero and he could have any information about the content of the first block of plaintext, for a brute-force attack he would have to try about $2^{1.000.000.000}$ different combinations. Since the attacker does not normally know the length of the key, assuming the startpad equals zero, the number of attempts would be:

$$\sum_{i=501-1}^{1.000.000.000-1} = 2^i$$

We denote by $|maxmodulo|$ the longest key that a system can handle in a "reasonably short time"[2] and by $|minmodulo|$ the minimum recommended length of the primary key. Generalizing and assuming that $|tplain| > |maxmodulo|$ we have:

$$\sum_{i=|minmodulo|-1}^{|maxmodulo|-1} = 2^i$$

FC1 therefore provides an incredible grade of confidentiality, compared to the standards currently in use, which makes it suitable for facing the difficult challenges of the next future. As far as integrity is concerned, it is ensured by adding a tag generated by a SHA-256 function. In a related work we discuss in detail other possible attacks (such as the 'replay attack') and we show how FC1 is immune to them.

References

[1] Victor Shoup (2009) *A Computational Introduction to Number Theory and Algebra*, Cambridge University Press; 2nd ed.

[2] Michele Bufalo, Daniele Bufalo, Giuseppe Orlando (2021) *A Note on the Computation of the Modular Inverse for Cryptography*, Axioms

[3] Niels Moeller (2007) *On Schoenhage's algorithm and subquadratic integer GCD computation*, MATHEMATICS OF COMPUTATION

[2] We should talk a lot about the meaning of "reasonably short time", but this is not the proper place because the subject deserves a separate work.

Appendices

A Code in Julia Programming Language

```
NOTE: The following code was tested on a Windows OS. This is a beta version that
is still under development and you might encounter some bugs. We would welcome
general comments and feedback at fc1@fabbrini.org.
Download latest Julia stable release at https://julialang.org/downloads. In 'bin'
folder create following .txt files: plaintext.txt, primarykey.txt, secondarykey.txt,
ciphertext.txt, decryptedplaintext.txt. Input a binary string in plaintext.txt, a
binary integer in primarykey.txt, a decimal integer in secondarykey.txt.
```

A.1 Encryption

File: 'FC1Encryption.jl'

```
#
# FC1 algorithm code by Michele Fabbrini in Julia Programming Language is made available
# under the Creative Commons Attribution license. The following is a human-readable
# summary of (and not a substitute for) the full legal text of the CC BY 4.0 license
# https://creativecommons.org/licenses/by/4.0/.
#
# You are free:
#
#  to Share-copy and redistribute the material in any medium or format
#  to Adapt-remix, transform, and build upon the material
#
# for any purpose, even commercially.
#
# The licensor cannot revoke these freedoms as long as you follow the license terms.
#
# Under the following terms:
#
# ATTRIBUTION - You must give appropriate credit (mentioning that your work is derived
# from work by Michele Fabbrini), provide a link to the license, and indicate if
# changes were made.
# You may do so in any reasonable manner, but not in any way that suggests the licensor
# endorses you or your use.
#
# No additional restrictions - You may not apply legal terms or technological measures
# that legally restrict others from doing anything the license permits.
# With the understanding that:
#
# Notices:
#
# You do not have to comply with the license for elements of the material in the public
# domain or where your use is permitted by an applicable exception or limitation.
# No warranties are given. The license may not give you all of the permissions necessary
# for your intended use. For example, other rights such as publicity, privacy, or moral
# rights may limit how you use the material.
#
# JULIA LICENCE
#
# MIT License
```

```
#
# Copyright (c) 2009-2022: Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and other
# contributors: https://github.com/JuliaLang/julia/contributors
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this
# software and associated documentation files (the "Software"), to deal in the Softwareù
# without restriction, including without limitation the rights to use, copy, modify,
# merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
# permit persons to whom the Software is furnished to do so, subject to the following
# conditions:
#
# The above copyright notice and this permission notice shall be included in all copies
# or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
# INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
# PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
# HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
# OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
# SOFTWARE OR THE USE OR  OTHER DEALINGS IN THE SOFTWARE.
#
# end of terms and conditions
#
# Please see THIRDPARTY.md for license information for other software used in this
# project https://github.com/JuliaLang/julia/blob/master/THIRDPARTY.md

# FC1Encryption - Version 1.0.0-beta

using SHA
using Random

####################
# Input Validation #
####################

# Opening plaintext.txt
plain=open(f->read(f, String), "plaintext.txt")

# Opening primarykey.txt
pkeybin=open(f->read(f, String), "primarykey.txt")

# Opening secondarykey.txt
skeystring=open(f->read(f, String), "secondarykey.txt")

# Checking plaintext
function fplainvalid()
try
        parse(BigInt,plain,base=2)
    catch err
        if isa(err, ArgumentError)
            println("Plaintext MUST be a binary string!")
            sleep(10)
            exit()
        end
end
end
fplainvalid()
```

```
# Checking primarykey
function fpkeyvalid1()
    try
    parse(BigInt,plain,base=2)
    catch err
        if isa(err, ArgumentError)
            println("Primarykey MUST be a binary string!")
            sleep(10)
            exit()
        end
end
end
fpkeyvalid1()

function fpkeyvalid2()
    if startswith(pkeybin, "0")
    println("Primarykey MUST NOT start with ’0’!")
sleep(10)
        exit()
end
end
fpkeyvalid2()

# Checking secondarykey
function fskeyvalid()
    try
       parse(BigInt,skeystring,base=10)
    catch err
        if isa(err, ArgumentError)
            println("Skey MUST be an integer!")
            sleep(10)
            exit()
end
end
end
fskeyvalid()

###############
# Tag SHA-256 #
###############

taghex = bytes2hex(sha256(plain))
println("This is the 256-bit hash of Plaintext (hex): ", taghex)
tagdec = parse(BigInt,taghex,base=16)
tagbintemp = string(tagdec,base=2)
tagbintemplen = length(tagbintemp)

function ftagbitscheck()
    if tagbintemplen == 256
        global tagbin = tagbintemp
    else
       global tagbin = lpad(tagbintemp,256,"0")
    end
end
ftagbitscheck()

##############
# Sys. Init. #
```

```
##############

# Tagged Plaintext
tplain=string(plain,tagbin)

skey=parse(BigInt,skeystring)

# Start random string whose length is the value of secondarykey
startpad=randstring("01",skey)

plainlen=length(plain)
tplainlen=length(tplain)

pkeybinlen=length(pkeybin)

#Checking tagged plaintext length: it must be greater than modulo length.
function ftplainlenvalid()
    if tplainlen < pkeybinlen
        println("Tagged plain text string must be longer than modulo string!")
        sleep(20)
        exit()
end
end
ftplainlenvalid()

# The start value of tplain is passed to remtplain
remtplain=tplain

# The start value of tplain length is passed to remtplain length
remtplainlen=tplainlen

# Start ciphertex
cipher=startpad

# Treshold value
theshold=3/2*pkeybinlen

####################
# Encryption Start #
####################

function fencrypt()

    # Case 1
    if remtplainlen >= theshold
function frand1()
    rand(1:pkeybinlen-3)
end
# Random integer generated by frand1 function
    frand1value=frand1()
# First random length bits from tagged plaintext
        tplain1=first(remtplain,frand1value)

        # Preparing the input for the modular multiplicative inverse function.
# A leading e a trailing '1' are appended at each chunk of tagged plaintext
# whose length is randomly selected by frand function. The leading 1 is meant
# to make sure that the input of finvmod is a positive integer since tplain1
# could start with '0'. The trailing '1' serves to prevent the algorithm from
# blocking in the case of an even module and a text to be encrypted containing
# a long row of 0's.
        function fintgen1()
    string(1,tplain1,1)
end
input1bin=fintgen1()
input1=parse(BigInt,input1bin,base=2)
        pkey=parse(BigInt,pkeybin,base=2)

        function finv1(input1,pkey)
            try
                @time invmod(input1,pkey)
            catch err
                if isa(err, DomainError)
                    println("Input1 and pkey are not coprime.")
                    fencrypt()
    end
            end
        end
        c1=finv1(input1,pkey)
    c1bin=string(c1,base=2)
c1binlen=length(c1bin)

        function fc1blockgen()
            if c1binlen == pkeybinlen
                global c1block = c1bin
            else
       function fzerospad1()
           lpad(c1bin,pkeybinlen,"0")
   end
   global c1block = fzerospad1()
end
end
# Block of ciphertext whose length matches modulo length
        fc1blockgen()
        # Tagged plaintext length residue
        global remtplainlen=remtplainlen-frand1value
        # Remaining tagged plaintext
        global remtplain=last(tplain,remtplainlen)
        # Ciphertext genesis
global cipher=string(cipher,c1block)

        fencrypt()


    # Case 2
    elseif pkeybinlen <= remtplainlen <= theshold
function frand2()
    rand(remtplainlen-pkeybinlen+3:pkeybinlen-3)
    end
# Random integer generated by frand2 function
frand2value=frand2()
        # Inferred last block length
tplain2lastlen=remtplainlen-frand2value

tplain2=first(remtplain,frand2value)
        input2bin=string(1,tplain2,1)
        input2=parse(BigInt,input2bin,base=2)
```

```
pkey=parse(BigInt,pkeybin,base=2)

function finv2(input2,pkey)
            try
                @time  invmod(input2,pkey)
            catch err
                if isa(err, DomainError)
                    println("Input2 and pkey are not coprime.")
                    fencrypt()
                end
            end
         end
         c2=finv2(input2,pkey)
         c2bin=string(c2,base=2)
         c2binlen=length(c2bin)

         function fc2blockgen()
            if c2binlen == pkeybinlen
                global c2block = c2bin
            else
       function fzerospad2()
           lpad(c2bin,pkeybinlen,"0")
   end
   global c2block = fzerospad2()
end
 end
         fc2blockgen()

 tplain2last=last(tplain,tplain2lastlen)
         input2lastbin=string(1,tplain2last,1)
         input2last=parse(BigInt,input2lastbin,base=2)
         pkey=parse(BigInt,pkeybin,base=2)

 function finv2last(input2last,pkey)
             try
                 @time  invmod(input2last,pkey)
             catch err
                 if isa(err, DomainError)
                     println("Input2last and pkey are not coprime.")
                     fencrypt()
 end
             end
         end
         c2last=finv2last(input2last,pkey)
         c2lastbin=string(c2last,base=2)
         c2lastbinlen=length(c2lastbin)

         function fc2lastblockgen()
             if c2lastbinlen == pkeybinlen
                global c2lastblock = c2lastbin
             else
        function fzerospad2last()
            lpad(c2lastbin,pkeybinlen,"0")
    end
    global c2lastblock = fzerospad2last()
   end
     end
         fc2lastblockgen()
         function frandend2()
             rand(0:pkeybinlen-1)
 end
 frandend2value=frandend2()
 # End padding generated by a random function
         ikey=randstring("01",frandend2value)

         global ikeylen=length(ikey)

         # Final ciphertext
 global cipher=string(cipher,c2block,c2lastblock,ikey)
 global clen=length(cipher)

 function fwritetofile2()
     open("ciphertext.txt", "w") do f
                 write(f, cipher)
             end
 end
 fwritetofile2()
         println("Ciphertext has been generated and updated in ’ciphertext.txt’.")

    # Case 3
    elseif remtplainlen == pkeybinlen - 1
function frand3()
    rand(2:pkeybinlen-2)
    end
frand3value=frand3()

        tplain3lastlen=remtplainlen-frand3value
        tplain3=first(remtplain,frand3value)
        input3bin=string(1,tplain3,1)
        input3=parse(BigInt,input3bin,base=2)
        pkey=parse(BigInt,pkeybin,base=2)

    function finv3(a3,pkey)
            try
                @time  invmod(input3,pkey)
            catch err
                if isa(err, DomainError)
                    println("Input3 and pkey are not coprime.")
                    fencrypt()
     end
            end
        end
        c3=finv3(input3,pkey)
        c3bin=string(c3,base=2)
        c3binlen=length(c3bin)

        function fc3blockgen()
            if c3binlen == pkeybinlen
                global c3block = c3bin
            else
        function fzerospad3()
            lpad(c3bin,pkeybinlen,"0")
    end
    global c3block = fzerospad3()
  end
```

```
    end
        fc3blockgen()

tplain3last=last(tplain,tplain3lastlen)
input3lastbin=string(1,tplain3last,1)
        input3last=parse(BigInt,input3lastbin,base=2)
        pkey=parse(BigInt,pkeybin,base=2)

function finv3last(input3last,pkey)
            try
                @time invmod(input3last,pkey)
            catch err
                if isa(err, DomainError)
                    println("Input3last and pkey are not coprime.")
                     encrypt()
    end
            end
        end
        c3last=finv3last(input3last,pkey)
        c3lastbin=string(c3last,base=2)
        c3lastbinlen=length(c3lastbin)

        function fc3lastblockgen()
            if c3lastbinlen == pkeybinlen
                global c3lastblock = c3lastbin
            else
        function fzerospad3last()
            lpad(c3lastbin,pkeybinlen,"0")
    end
    global c3lastblock = fzerospad3last()
             end
     end
         fc3lastblockgen()

         function frandend3()
             rand(0:pkeybinlen-1)
 end
 frandend3value=frandend3()
 # End padding generated by a random function
         ikey=randstring("01",frandend3value)

 global ikeylen=length(ikey)

         # Final ciphertext
         global cipher=string(cipher,c3block,c3lastblock,ikey)
 global clen=length(cipher)

 function fwritetofile3()
     open("ciphertext.txt", "w") do f
                  write(f, cipher)
 end
         end
 fwritetofile3()
 println("Ciphertext has been generated and updated in 'ciphertext.txt'.")

    # Case 4
    elseif remtplainlen <= pkeybinlen - 2
function frand4()
```

```
    rand(1:remtplainlen-1)
end
frand4value=frand4()
        tplain4lastlen=remtplainlen-frand4value
        tplain4=first(remtplain,frand4value)
        input4bin=string(1,tplain4,1)
        input4=parse(BigInt,input4bin,base=2)
        pkey=parse(BigInt,pkeybin,base=2)

function finv4(input4,pkey)
            try
                @time  invmod(input4,pkey)
            catch err
                if isa(err, DomainError)
                    println("Input4 and pkey are not coprime.")
                    fencrypt()
    end
            end
        end
        c4=finv4(input4,pkey)
        c4bin=string(c4,base=2)
        c4binlen=length(c4bin)

        function fc4blockgen()
            if c4binlen == pkeybinlen
                global c4block = c4bin
            else
        function fzerospad4()
            lpad(c4bin,pkeybinlen,"0")
     end
    global c4block = fzerospad4()
            end
    end
        fc4blockgen()
tplain4last=last(tplain,tplain4lastlen)
input4lastbin=string(1,tplain4last,1)
        input4last=parse(BigInt,input4lastbin,base=2)
        pkey=parse(BigInt,pkeybin,base=2)

function finv4last(input4last,pkey)
            try
                @time  invmod(input4last,pkey)
            catch err
                if isa(err, DomainError)
                    println("Input4last and pkey are not coprime.")
                    fencrypt()
    end
            end
        end
        c4last=finv4last(input4last,pkey)
        c4lastbin=string(c4last,base=2)
        c4lastbinlen=length(c4lastbin)

        function fc4lastblockgen()
            if c4lastbinlen == pkeybinlen
                global c4lastblock = c4lastbin
            else
        function fzerospad4last()
```

```
            lpad(c4lastbin,pkeybinlen,"0")
    end
    global c4lastblock = fzerospad4last()
  end
    end
        fc4lastblockgen()

        function frandend4()
            rand(0:pkeybinlen-1)
end
frandend4value=frandend4()
# End padding generated by a random function
        ikey=randstring("01",frandend4value)

global ikeylen=length(ikey)

# Final ciphertext
        global cipher=string(cipher,c4block,c4lastblock,ikey)
        global clen=length(cipher)

        function fwritetofile4()
    open("ciphertext.txt", "w") do f
                 write(f, cipher)
            end
    end
fwritetofile4()
println("Ciphertext has been generated and updated in 'ciphertext.txt'.")

        end
end

function fend()
     try
        fencrypt()
     catch err
         if isa(err, MethodError)
     end
     end
end
fend()
println(" ")
println("----------------- ")
println("Encryption Report")
println("----------------- ")
println("Primary Key Length: ", pkeybinlen)
println("Secondary Key: ", skey)
println("Inferred Key Length: ", ikeylen)
println("Tagged Plaintext Length: ", tplainlen)
println("Plaintext Length: ", plainlen)
println("Ciphertext Length: ", clen)
```

A.2 Decryption

File: 'FC1Decryption.jl'

```
#
# FC1 algorithm code by Michele Fabbrini in Julia Programming Language is made available
# under the Creative Commons Attribution license. The following is a human-readable
# summary of (and not a substitute for) the full legal text of the CC BY 4.0 license
# https://creativecommons.org/licenses/by/4.0/.
#
# You are free:
#
#  to Share-copy and redistribute the material in any medium or format
#  to Adapt-remix, transform, and build upon the material
#
# for any purpose, even commercially.
#
# The licensor cannot revoke these freedoms as long as you follow the license terms.
#
# Under the following terms:
#
# ATTRIBUTION - You must give appropriate credit (mentioning that your work is derived
# from work by Michele Fabbrini), provide a link to the license, and indicate if
# changes were made.
# You may do so in any reasonable manner, but not in any way that suggests the licensor
# endorses you or your use.
#
# No additional restrictions - You may not apply legal terms or technological measures
# that legally restrict others from doing anything the license permits.
# With the understanding that:
#
# Notices:
#
# You do not have to comply with the license for elements of the material in the public
# domain or where your use is permitted by an applicable exception or limitation.
# No warranties are given. The license may not give you all of the permissions necessary
# for your intended use. For example, other rights such as publicity, privacy, or moral
# rights may limit how you use the material.
#
# JULIA LICENCE
#
# MIT License
#
# Copyright (c) 2009-2022: Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and other
# contributors: https://github.com/JuliaLang/julia/contributors
#
# Permission is hereby granted, free of charge, to any person obtaining a copy of this
# software and associated documentation files (the "Software"), to deal in the Softwareù
# without restriction, including without limitation the rights to use, copy, modify,
# merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
# permit persons to whom the Software is furnished to do so, subject to the following
# conditions:
#
# The above copyright notice and this permission notice shall be included in all copies
# or substantial portions of the Software.
#
```

```
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
# INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
# PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
# HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
# OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
# SOFTWARE OR THE USE OR  OTHER DEALINGS IN THE SOFTWARE.
#
# end of terms and conditions
#
# Please see THIRDPARTY.md for license information for other software used in this
# project https://github.com/JuliaLang/julia/blob/master/THIRDPARTY.md

# FC1Decryption - Version 1.0.0-beta

using SHA

####################
# Input Validation #
####################

# Opening ciphertext.txt
cipher=open(f->read(f, String), "ciphertext.txt")

# Opening primarykey.txt
pkeybin=open(f->read(f, String), "primarykey.txt")

# Opening secondarykey.txt
skeystring=open(f->read(f, String), "secondarykey.txt")

#Checking plaintext
function fplainvalid()
try
        parse(BigInt,plain,base=2)
    catch err
        if isa(err, ArgumentError)
            println("Plaintext MUST be a binary string!")
            sleep(10)
            exit()
        end
end
end
fplainvalid()

#Checking primarykey
function fpkeyvalid1()
    try
    parse(BigInt,plain,base=2)
    catch err
        if isa(err, ArgumentError)
            println("Primarykey MUST be a binary string!")
            sleep(10)
            exit()
        end
end
end
fpkeyvalid1()

function fpkeyvalid2()
```

```
    if startswith(pkeybin, "0")
    println("Primarykey MUST NOT start with ’0’!")
sleep(10)
        exit()
end
end
fpkeyvalid2()

#Checking secondarykey
function fskeyvalid()
    try
       parse(BigInt,skeystring,base=10)
    catch err
        if isa(err, ArgumentError)
            println("Skey MUST be an integer!")
            sleep(10)
            exit()
end
end
end
fskeyvalid()

##############
# Sys. Init. #
##############

skey=parse(Int64,skeystring)
clen=length(cipher)
remclen=clen-skey
remc=last(cipher,remclen)
pkeybinlen=length(pkeybin)
tplain=""

####################
# Decryption Start #
####################

function fdecrypt()

    # Case1
    if remclen >= pkeybinlen
        cblock=first(remc,pkeybinlen)
inputbin=lstrip(cblock,[’0’])
        input=parse(BigInt,inputbin,base=2)
        pkey=parse(BigInt,pkeybin,base=2)

        function finv(input,pkey)
    try
               @time invmod(input,pkey)
catch err
                if isa(err, DomainError)
                    println("ATTENTION! CIPHERTEXT COULD BE CORRUPTED!")
                    sleep(10)
exit()
    end
            end
        end
        output=finv(input,pkey)
        outputbin=string(output,base=2)
        tplain1=chop(outputbin, head = 1, tail = 1)
global remclen=remclen-pkeybinlen
        global remc=last(remc,remclen)
global tplain=string(tplain,tplain1)
    fdecrypt()


    # Case 2
    elseif remclen < pkeybinlen
global tplainlen=length(tplain)
global plainlen=tplainlen-256
    plaincheck=first(tplain,plainlen)

########################
# Data Integrity Check #
########################

    tagcheck=last(tplain,256)
        taghex= bytes2hex(sha256(plaincheck))
        println("This is the 256-bit hash of Plaintext check (hex): ", taghex)
        tagdec=parse(BigInt,taghex,base=16)
        tagbintemp=string(tagdec,base=2)
        tagbintemplen = length(tagbintemp)
global ikeylen=remclen

        function ftagbitscheck()
            if tagbintemplen == 256
                global tagbin = tagbintemp
            else
                global tagbin = lpad(tagbintemp,256,"0")
            end
end
ftagbitscheck()

function fintegritycheck()
    check=cmp(tagcheck::AbstractString, tagbin::AbstractString)
                  if check==0
      function fwritetofile()
                  open("decryptedplaintext.txt", "w") do f
                               write(f, plaincheck)
                          end
                  end
  fwritetofile()
                      println("Decrypted Plaintext has been generated.")
              println("SUCCESS!!!")
          else
                      println("DATA INTEGRITY ALERT: CORRUPTED CIPHERTEXT!")
                  end

         end
 fintegritycheck()
    end
end

fdecrypt()

println(" ")
println("----------------- ")
println("Decryption Report")
println("----------------- ")
println("Primary Key Length: ", pkeybinlen)
println("Secondary Key: ", skey)
println("Inferred Key Length: ", ikeylen)
println("Tagged Plaintext Length: ", tplainlen)
println("Decrypted Plaintext Length: ", plainlen)
println("Ciphertext Length: ", clen)
```

B Tests on algorithm performance

Below are the results of some encryption and decryption tests.

B.1 Primary Key Length= 10.053 bits

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _‘ |  |
  | | |_| | | | (_| |  |  Version 1.7.2 (2022-02-06)
 _/ |\__’_|_|_|\__’_|  |  Official https://julialang.org/ release
|__/                   |

julia> include("FC1Encryption.jl")
This is the 256-bit hash of Plaintext (hex): 53ca1929e5493eb9aa61dd3c5a75274309b833a1caec38ec2ee2a
  0.001701 seconds (9 allocations: 1.391 KiB)
  0.000167 seconds (9 allocations: 1.375 KiB)
  0.000031 seconds (9 allocations: 1.391 KiB)
  0.000074 seconds (9 allocations: 1.391 KiB)
  0.000009 seconds (9 allocations: 1.391 KiB)
  0.000069 seconds (9 allocations: 1.391 KiB)
  0.000173 seconds (9 allocations: 1.383 KiB)
  0.000094 seconds (9 allocations: 1.383 KiB)
  0.000159 seconds (9 allocations: 1.383 KiB)
  0.000084 seconds (9 allocations: 1.383 KiB)
Ciphertext has been generated and updated in ’ciphertext.txt’.

-----------------
Encryption Report
-----------------
Primary Key Length: 10053
Secondary Key Length: 150
Inferred Key Length: 9956
Tagged Plaintext Length: 51119
Plaintext Length: 50863
Ciphertext Length: 110636

julia> include("FC1Decryption.jl")
  0.000132 seconds (9 allocations: 496 bytes)
  0.000249 seconds (9 allocations: 1.219 KiB)
  0.000044 seconds (9 allocations: 352 bytes)
  0.000106 seconds (9 allocations: 680 bytes)
  0.000015 seconds (9 allocations: 200 bytes)
  0.000164 seconds (9 allocations: 672 bytes)
  0.000213 seconds (9 allocations: 1.328 KiB)
  0.000136 seconds (9 allocations: 848 bytes)
  0.000344 seconds (9 allocations: 1.289 KiB)
  0.000122 seconds (9 allocations: 776 bytes)
This is the 256-bit hash of Plaintext check (hex): 53ca1929e5493eb9aa61dd3c5a75274309b833a1caec38e
Decrypted Plaintext has been generated.
SUCCESS!!!

-----------------
Decryption Report
-----------------
Primary Key Length: 10053
Secondary Key Length: 150
Inferred Key Length: 9956
Tagged Plaintext Length: 51119
Decrypted Plaintext Length: 50863
Ciphertext Length: 110636
```

B.2 Primary Key Length= 150.816 bits

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _‘ |  |
  | | |_| | | | (_| |  |  Version 1.7.2 (2022-02-06)
 _/ |\__’_|_|_|\__’_|  |  Official https://julialang.org/ release
|__/                   |

julia> include("FC1Encryption.jl")
This is the 256-bit hash of Plaintext (hex): 3f23d78c51359f21d2fa5ff53dea807bbf63b39e3bbb6a91006b1
Input1 and pkey are not coprime.
  0.010415 seconds (125 allocations: 697.438 KiB)
  0.001637 seconds (9 allocations: 18.562 KiB)
  0.004405 seconds (12 allocations: 170.078 KiB)
Input1 and pkey are not coprime.
  0.007846 seconds (125 allocations: 680.000 KiB)
Input1 and pkey are not coprime.
Input1 and pkey are not coprime.
  0.001338 seconds (9 allocations: 18.562 KiB)
Input1 and pkey are not coprime.
  0.000693 seconds (9 allocations: 18.562 KiB)
Input1 and pkey are not coprime.
Input1 and pkey are not coprime.
  0.002875 seconds (11 allocations: 97.602 KiB)
Input1 and pkey are not coprime.
Input1 and pkey are not coprime.
  0.004881 seconds (13 allocations: 218.570 KiB)
  0.002747 seconds (11 allocations: 96.555 KiB)
  0.008237 seconds (125 allocations: 684.242 KiB)
  0.010545 seconds (125 allocations: 701.188 KiB)
  0.003932 seconds (11 allocations: 110.820 KiB)
  0.005098 seconds (13 allocations: 221.781 KiB)
  0.000851 seconds (9 allocations: 18.562 KiB)
  0.005709 seconds (13 allocations: 236.375 KiB)
  0.000275 seconds (9 allocations: 18.570 KiB)
Input1 and pkey are not coprime.
  0.010648 seconds (125 allocations: 701.234 KiB)
Input1 and pkey are not coprime.
```

```
  0.008399 seconds (12 allocations: 215.562 KiB)
  0.005922 seconds (12 allocations: 174.695 KiB)
Input2last and pkey are not coprime.
Input2 and pkey are not coprime.
  0.006031 seconds (13 allocations: 225.695 KiB)
  0.007911 seconds (61 allocations: 541.859 KiB)
Ciphertext has been generated and updated in 'ciphertext.txt'.

-----------------
Encryption Report
-----------------
Primary Key Length: 150816
Secondary Key Length: 150
Inferred Key Length: 13396
Tagged Plaintext Length: 1600039
Plaintext Length: 1599783
Ciphertext Length: 3029866

julia> include("FC1Decryption.jl")
  0.007869 seconds (75 allocations: 352.625 KiB)
  0.001753 seconds (43 allocations: 270.891 KiB)
  0.004601 seconds (75 allocations: 344.273 KiB)
  0.007823 seconds (75 allocations: 351.375 KiB)
  0.001400 seconds (43 allocations: 270.281 KiB)
  0.000841 seconds (43 allocations: 281.500 KiB)
  0.002724 seconds (75 allocations: 353.172 KiB)
  0.004973 seconds (75 allocations: 345.117 KiB)
  0.002812 seconds (75 allocations: 352.961 KiB)
  0.008790 seconds (75 allocations: 351.562 KiB)
  0.008859 seconds (75 allocations: 352.875 KiB)
  0.003611 seconds (75 allocations: 342.117 KiB)
  0.005166 seconds (75 allocations: 345.352 KiB)
  0.001081 seconds (43 allocations: 277.445 KiB)
  0.006333 seconds (75 allocations: 346.328 KiB)
  0.000589 seconds (43 allocations: 284.883 KiB)
  0.009282 seconds (75 allocations: 352.883 KiB)
  0.007083 seconds (75 allocations: 347.602 KiB)
  0.005848 seconds (75 allocations: 345.641 KiB)
  0.008215 seconds (75 allocations: 350.570 KiB)
This is the 256-bit hash of Plaintext check (hex): 3f23d78c51359f21d2fa5ff53dea807bbf63b39e3bbb6a9
Decrypted Plaintext has been generated.
SUCCESS!!!

-----------------
Decryption Report
-----------------
Primary Key Length: 150816
Secondary Key Length: 150
Inferred Key Length: 13396
Tagged Plaintext Length: 1600039
Decrypted Plaintext Length: 1599783
Ciphertext Length: 3029866
```

B.3 Primary Key Length= 1.357.350 bits

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _‘ |  |
  | | |_| | | | (_| |  |  Version 1.7.2 (2022-02-06)
 _/ |\__’_|_|_|\__’_|  |  Official https://julialang.org/ release
|__/                   |

julia> include("FC1Encryption.jl")
This is the 256-bit hash of Plaintext (hex): 001d52e69ca6e2cb0f36b5a85abdf88ee35d045c731d9213d0417
Input1 and pkey are not coprime.
  0.079337 seconds (2.10 k allocations: 16.634 MiB, 6.43% gc time)
  0.022644 seconds (751 allocations: 5.753 MiB)
  0.155660 seconds (3.52 k allocations: 33.514 MiB)
  0.026288 seconds (776 allocations: 6.973 MiB)
  0.098616 seconds (2.38 k allocations: 19.620 MiB, 15.75% gc time)
Input1 and pkey are not coprime.
  0.141321 seconds (2.50 k allocations: 23.380 MiB, 27.64% gc time)
Input1 and pkey are not coprime.
Input1 and pkey are not coprime.
  0.054470 seconds (1.37 k allocations: 12.989 MiB)
  0.130649 seconds (3.33 k allocations: 31.576 MiB)
  0.167194 seconds (5.06 k allocations: 41.872 MiB, 0.14% gc time)
Input1 and pkey are not coprime.
Input1 and pkey are not coprime.
  0.153433 seconds (5.04 k allocations: 39.909 MiB, 0.31% gc time)
Input1 and pkey are not coprime.
  0.028637 seconds (832 allocations: 7.536 MiB)
  0.034337 seconds (1.07 k allocations: 8.412 MiB)
  0.006335 seconds (15 allocations: 1.137 MiB)
Input1 and pkey are not coprime.
  0.030510 seconds (824 allocations: 7.385 MiB)
  0.123584 seconds (4.00 k allocations: 34.440 MiB)
Input1 and pkey are not coprime.
  0.210985 seconds (4.99 k allocations: 40.337 MiB, 24.55% gc time)
  0.195240 seconds (5.96 k allocations: 51.011 MiB)
  0.125717 seconds (3.32 k allocations: 30.290 MiB)
  0.175762 seconds (5.06 k allocations: 46.493 MiB)
Input2last and pkey are not coprime.
  0.138269 seconds (3.46 k allocations: 33.089 MiB)
  0.106131 seconds (2.60 k allocations: 24.115 MiB)
Ciphertext has been generated and updated in ’ciphertext.txt’.

-----------------
Encryption Report
-----------------
Primary Key Length: 1357350
Secondary Key Length: 150
Inferred Key Length: 566031
Tagged Plaintext Length: 13889203
Plaintext Length: 13888947
Ciphertext Length: 27713181

julia> include("FC1Decryption.jl")
  0.067702 seconds (2.23 k allocations: 18.535 MiB)
  0.019518 seconds (661 allocations: 7.289 MiB)
  0.122074 seconds (3.95 k allocations: 32.618 MiB)
  0.025134 seconds (856 allocations: 8.547 MiB)
  0.076639 seconds (2.64 k allocations: 21.379 MiB)
  0.093663 seconds (3.33 k allocations: 26.467 MiB)
  0.059261 seconds (2.01 k allocations: 16.877 MiB)
  0.120642 seconds (4.06 k allocations: 31.656 MiB)
  0.186309 seconds (4.58 k allocations: 37.238 MiB, 21.03% gc time)
  0.140148 seconds (4.53 k allocations: 36.074 MiB, 0.91% gc time)
  0.029147 seconds (888 allocations: 8.639 MiB)
  0.033718 seconds (1.26 k allocations: 10.939 MiB)
  0.010100 seconds (336 allocations: 4.878 MiB)
  0.028500 seconds (855 allocations: 8.553 MiB)
  0.106262 seconds (3.59 k allocations: 30.286 MiB)
  0.141627 seconds (4.50 k allocations: 36.895 MiB)
  0.178383 seconds (5.64 k allocations: 44.613 MiB)
  0.111479 seconds (3.53 k allocations: 28.733 MiB)
  0.124857 seconds (4.00 k allocations: 32.596 MiB, 0.71% gc time)
  0.097205 seconds (3.13 k allocations: 25.953 MiB)
This is the 256-bit hash of Plaintext check (hex): 001d52e69ca6e2cb0f36b5a85abdf88ee35d045c731d921
Decrypted Plaintext has been generated.
SUCCESS!!!

-----------------
Decryption Report
-----------------
Primary Key Length: 1357350
Secondary Key Length: 150
Inferred Key Length: 566031
Tagged Plaintext Length: 13889203
Decrypted Plaintext Length: 13888947
Ciphertext Length: 27713181
```

B.4 Primary Key Length= 112.068.017 bits

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _‘ |  |
  | | |_| | | | (_| |  |  Version 1.7.2 (2022-02-06)
 _/ |\__’_|_|_|\__’_|  |  Official https://julialang.org/ release
|__/                   |

julia> include("FC1Encryption.jl")
This is the 256-bit hash of Plaintext (hex): 6b84e48dde47d5c99b388744e0d8e4cd57ffe379fc1527adb6011
 51.120033 seconds (924.40 k allocations: 14.229 GiB, 0.04% gc time)
 27.430180 seconds (475.51 k allocations: 7.870 GiB, 0.05% gc time)
Ciphertext has been generated and updated in ’ciphertext.txt’.

-----------------
Encryption Report
-----------------
Primary Key Length: 112068017
Secondary Key Length: 150
Inferred Key Length: 97244088
Tagged Plaintext Length: 166408768
Plaintext Length: 166408512
Ciphertext Length: 321380272

julia> include("FC1Decryption.jl")
 40.472839 seconds (618.78 k allocations: 10.403 GiB, 0.14% gc time)
 22.585723 seconds (336.12 k allocations: 5.673 GiB, 0.34% gc time)
```

```
This is the 256-bit hash of Plaintext check (hex): 6b84e48dde47d5c99b388744e0d8e4cd57ffe379fc1527a
Decrypted Plaintext has been generated.
SUCCESS!!!

-----------------
Decryption Report
-----------------
Primary Key Length: 112068017
Secondary Key Length: 150
Inferred Key Length: 97244088
Tagged Plaintext Length: 166408768
Decrypted Plaintext Length: 166408512
Ciphertext Length: 321380272
```

B.5 Primary Key Length= 1.011.517.781 bits

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _‘ |  |
  | | |_| | | | (_| |  |  Version 1.7.2 (2022-02-06)
 _/ |\__’_|_|_|\__’_|  |  Official https://julialang.org/ release
|__/                   |

julia> include("FC1Encryption.jl")
This is the 256-bit hash of Plaintext (hex): 81b3beefd6679ff707b77d5c54e88155179202054d7c75fb2091b
568.425280 seconds (10.21 M allocations: 162.935 GiB, 0.06% gc time)
183.227033 seconds (3.60 M allocations: 54.665 GiB, 0.06% gc time)
Ciphertext has been generated and updated in ’ciphertext.txt’.

-----------------
Encryption Report
-----------------
Primary Key Length: 1011517781
Secondary Key Length: 150
Inferred Key Length: 24143458
Tagged Plaintext Length: 1073742079
Plaintext Length: 1073741823
Ciphertext Length: 2047179170

julia> include("FC1Decryption.jl")
423.218162 seconds (5.99 M allocations: 109.935 GiB, 0.05% gc time)
156.789606 seconds (2.21 M allocations: 42.008 GiB, 0.06% gc time)
This is the 256-bit hash of Plaintext check (hex): 81b3beefd6679ff707b77d5c54e88155179202054d7c75f
Decrypted Plaintext has been generated.
SUCCESS!!!

-----------------
Decryption Report
-----------------
Primary Key Length: 1011517781
Secondary Key Length: 150
Inferred Key Length: 24143458
Tagged Plaintext Length: 1073742079
Decrypted Plaintext Length: 1073741823
Ciphertext Length: 2047179170
```