Advanced Encryption Standard: Galois Counter Mode in Julia

Kyle Kotowick

18.377, 2016

To ensure that code examples in this notebook run as intended, please run the following code to install required libraries.

In [ ]:
```
Pkg.add("Nettle")
Pkg.add("BenchmarkTools")
Pkg.add("JLD")
Pkg.add("PyPlot")
Pkg.update();
```

1) Advanced Encryption Standard (AES)

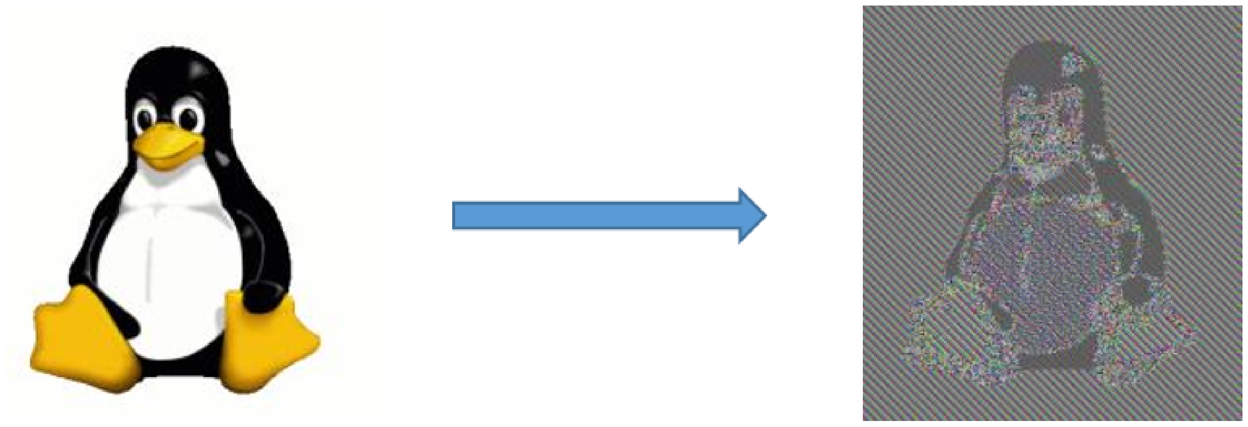
AES is a specification by the National Institute of Standards and Technology (NIST), published in 2001, for encryption of electronic data. It replaced the older specfication, Data Encryption Standard (DES), which had been in use since 1977. AES is a subset of the Rijndael cipher.

AES is a 128-bit block cipher, meaning that it can only encrypt fixed-size blocks of 128 bits. It is also a symmetric cipher, meaning that the same key is used for encryption and decryption. It supports key lengths of 128, 192, or 256 bits. AES is one of the world's most commonly used encryption algorithms.

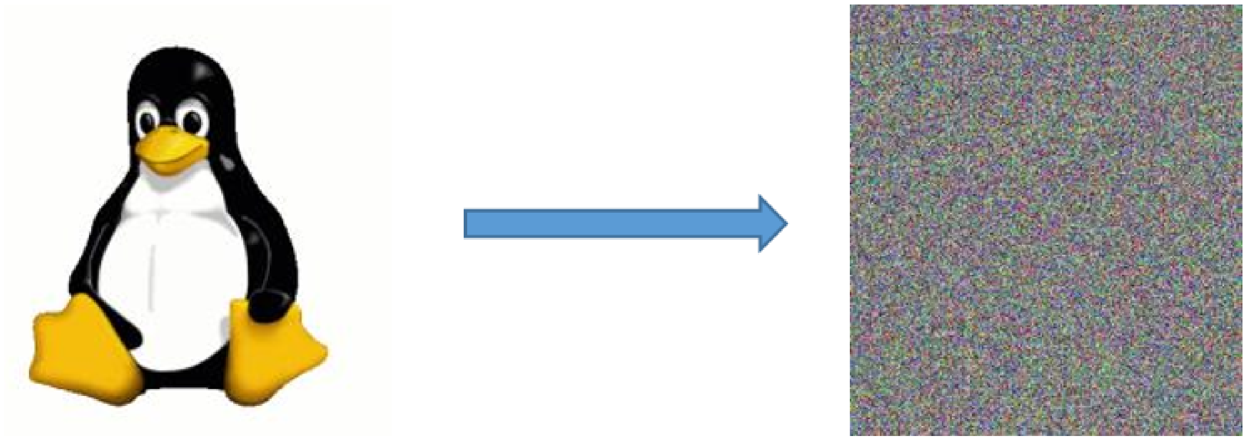
2) Modes of Operation

As mentioned, the AES block cipher operates on fixed-size blocks of 128 bits. Unfortunately, very few applications require encryption of only 128 bits. The use of block ciphers with this limitation has lead to the development of "modes of operation," which are methods for encrypting larger files.

The most simple mode of operation is also the most intuitive, called "Electronic Code Book" mode. This is where the plaintext (unencrypted) data is segmented into 128-bit blocks, each block is encrypted using the block cipher, and the encrypted blocks are concatenated to form the ciphertext (encrypted) data. Since each block is encrypted independently of any other, this offers the feature of it being a fully parallelizable process. Unfortunately, this is an extremely insecure method because blocks with the same plaintext values will be encrypted to produce blocks with the same ciphertext values, resulting in problems such as the encryption of this image:



As you can see, even though each 128-bit block is encrypted, it is still easy to gather information about what the original plaintext data was. To handle this issue, modes of operation were created that "chain" blocks (using the output of one block in the encryption of another block). By doing so, even blocks that have the same plaintext data will produce different ciphertext data:



3) Integrity

Using AES with a strong mode of operation allows secrecy of data. That is to say, someone who views the ciphertext data during network transmission, for example, can not determine its content. However, that does not prevent such a person from changing the ciphertext data, even if he/she does not know what it is he/she is changing. This risk highlights another requirement for data: integrity and authenticity. It is usually necessary to know not only that the data has been kept secret, but also that it has not been tampered with.

There are multiple tools for ensuring integrity and authenticity, such as Message Authentication Codes (MACs). These can be used in combination with encryption to provide both secrecy and integrity/authenticity. The simplest technique is to encrypt the plaintext data, then use a MAC for the resulting ciphertext data. Modern modes of operation, however, allow both encryption and authentication together in one algorithm, usually providing an increase inperformance. This is known as "authenticated encryption."

4) Modes Comparison

There are many different modes of operation, each offering different performance and features. For example, some offer the ability to parallelize the encryption algorithm, the decryption algorithm, or both. Some offer authenticated encryption. Below is a non-exhaustive comparison of several modes:

Mode	Acronym	Encryption Parallelizable	Decryption Parallelizable	Authenticated
Cipher Block Chaining	CBC	No	Yes	No
Cipher Feedback	CFB	No	Yes	No
Output Feedback	OFB	No	No	No
Counter	CTR	Yes	Yes	No
Counter CBC-MAC	CCM	No	No	Yes
Galois Counter	GCM	Yes	Yes	Yes

Of particular note is the last row of the table, the "Galois Counter Mode" (GCM). GCM offers fully parallelizable encryption and decryption, as well as built-in authentication. GCM is relatively new, first published in 2004 and with the first NIST standards document for it released in 2007.

5) Encryption in Julia

There are two (easily-accessible) existing encryption packages for Julia.

The first, Nettle.jl (https://github.com/staticfloat/Nettle.jl), is a wrapper for the GNU Nettle library. It provides the base AES block cipher, as well as support for the Cipher Block Chaining mode. It has the benefit of being extremely fast (as a wrapper for a very well-developed library), but does not support Galois Counter Mode. Below is an example showing use of Nettle's AES block cipher using a 128-bit key.

In [ ]:
```
workspace()
using Nettle

# generate a random 128-bit key
key = rand(UInt8, 16)

# generate some random plaintext, must be 128 bits (block size)
plaintext = rand(UInt8, 16)

# encrypt it using the block cipher
enc = Encryptor("AES128", key)
ciphertext = encrypt(enc, plaintext)

# decrypt it
dec = Decryptor("AES128", key)
deciphertext = decrypt(dec, ciphertext)

# show that the decrypted version is the same as the plaintext version
plaintext == deciphertext
```

The second package, AES.jl (https://github.com/faf0/AES.jl), is a native Julia implementation of the AES block cipher. It also supports native implementations of the Electronic Code Book, Cipher Block Chaining, Cipher Feedback, Output Feedback, and Counter modes. It does not, however, support the Galois Counter Mode.

The main point of note here is that neither existing Julia encryption package supports the Galois Counter Mode. This offers an excellent opportunity for a final project for this course.

6) Project Plan

The initial plan for conducting this project was as follows:

1. Implement Galois Counter Mode as explicitly defined in the NIST recommendation 800-38D (http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf), using Nettle.jl's AES block cipher implementation
2. Refactor the code to optimize performance in serial
3. Implement the AES block cipher from the NIST FIPS 197 (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf) natively in Julia to remove need for external packages
4. Parallelize the GCM and AES code to improve performance

7) Step 1: implement GCM

The first step was to implement GCM from the NIST recommendation. Below is the resulting code. I used the same function names as in the NIST recommendation for clarity.

```
In [ ]:
```

```
workspace()
using Nettle
import Base.*

R = falses(128);
R[1] = true;
R[2] = true;
R[3] = true;
R[8] = true;

tag_length = UInt64(128)

function read_file_data_bits(filename)
    s = stat(filename)
    filesize = s.size
    data = falses(filesize*8);
    open(filename) do f
        read!(f, data)
    end
    return data
end;

function MSB(X::BitArray, s::UInt64)
    X[1 : s]
end;

function LSB(X::BitArray, s::UInt64)
    X[end - s + 1 : end]
end;

function RS!(X::BitArray, s::UInt64)
    deleteat!(X, collect(length(X) - s + 1 : length(X)))
    prepend!(X, falses(s))
end;

function RS(X::BitArray, s::UInt64)
    RS!(copy(X), s)
end;

function LS!(X::BitArray, s::UInt64)
    deleteat!(X, collect(1:s))
    append!(X, falses(s))
end;

function LS(X::BitArray, s::UInt64)
    LS!(copy(X), s)
end;

function int(X::BitArray)
    if(length(X) > 64)
        throw(ArgumentError("BitArray to Int: Too many bits"))
    end
    reinterpret(UInt64, reverse(X).chunks)[1]
end;

function binary(bytes::Array{UInt8,1}, len::Int64)
    n = length(bytes)
    m = ceil(UInt64, n / 8) * 8
    X = falses(m*8)
    X.chunks = reinterpret(UInt64,vcat(bytes, zeros(UInt8, m - n)))
    X.len = len
    X
end;

function bits2bytes(X::BitArray)
    reinterpret(UInt8, X.chunks)[1:ceil(UInt64, X.len / 8)]
end;

function binary(v::UInt64)
    t = falses(64)
    t.chunks = [v]
    reverse!(t)
    t
end;

function binary(v::UInt64, s::UInt64)
    if(v >= UInt128(2)^s)
        throw(ArgumentError("binary: insufficient bits for value"))
    end
    LSB(binary(v), s)
end;

function inc(X::BitArray, s::UInt64)
    if(s > 63)
        throw(ArgumentError("BitArray increment: Too many bits"))
    end
    right = binary((int(LSB(X, s)) + 1) % UInt64(2)^s, s);
    vcat(MSB(X, length(X) - s), right)
end;

function xor(X::BitArray, Y::BitArray)
    if(length(X) != length(Y))
        throw(ArgumentError("BitArray xor: unequal lengths"))
    end
    Z = falses(length(X));
    for i = 1:length(X)
        Z[i] = (X[i] + Y[i] == 1)
    end
    Z
end;

function Base.:*(X::BitArray, Y::BitArray)
    if(length(X) != 128 || length(Y) != 128 || length(R) != 128)
        throw(ArgumentError("BitArray multiplication: incorrect lengths"))
    end
    Z = falses(128);
    V = Y;
    for i = 1:128
        if(X[i])
            Z = xor(Z, V)
        end

        if(LSB(V, UInt64(1))[1])
            V = xor(RS(V, UInt64(1)), R)
        else
```

```
            V = RS(V, UInt64(1))
        end
    end
    Z
end;

function GHASH(X::BitArray, H::BitArray)
    if(X.len % 128 != 0 || length(H) != 128)
        throw(ArgumentError("GHASH: incorrect lengths"));
    end
    m = round(UInt64, X.len / 128);
    Y = falses(128);
    for i = 1:m
        Xi = X[128*(i-1) + 1 : 128*i]
        Y = xor(Y, Xi) * H
        # TODO: an array view of X so it doesn't copy?
    end
    Y
end;

function CIPH(X::BitArray, encryptor)
    binary(Nettle.encrypt(encryptor, bits2bytes(X)), X.len)
end

function GCTR(ICB::BitArray, X::BitArray, encryptor)
    if(ICB.len != 128)
        throw(ArgumentError("GCTR: incorrect lengths"))
    end
    if(X.len == 0)
        return BitArray[];
    end
    n = ceil(UInt64, X.len / 128)
    CB = [ICB]
    # TODO: do CB and Y calcs in the same loop so don't have to store all CB
    for i = 2:1:n
        push!(CB, inc(CB[i - 1], UInt64(32)))
    end
    Y = falses(X.len);
    for i = 1:1:n-1
        Xi = X[128*(i-1) + 1 : 128*i];
        Y[128*(i-1) + 1 : 128*i] = xor(Xi,CIPH(CB[i], encryptor))
    end
    X_star = LSB(X, UInt64(X.len - 128*(n-1)))
    Y[end - X_star.len + 1 : end] = xor(X_star, MSB(CIPH(CB[n], encryptor), 
    Y
end;

function GCMAE(IV::BitArray, P::BitArray, A::BitArray, Key::BitArray)

    encryptor = Nettle.Encryptor("AES128", bits2bytes(Key))

    if(P.len > 2^39 - 256)
        throw(ArgumentError("GCMAE: P too long"))
    end
    if(P.len % 8 != 0)
        throw(ArgumentError("GCMAE: P length not multiple of 8"))
    end
    if(A.len > UInt128(2)^64 - 1)
        throw(ArgumentError("GCMAE: A too long"))
    end
    if(A.len % 8 != 0)
        throw(ArgumentError("GCMAE: A length not multiple of 8"))
    end
    if(IV.len > UInt128(2)^64 - 1)
        throw(ArgumentError("GCMAE: IV too long"))
    end
    if(IV.len < 1)
        throw(ArgumentError("GCMAE: IV too short"))
    end
    if(IV.len % 8 != 0)
        throw(ArgumentError("GCMAE: IV length not multiple of 8"))
    end

    if(!in(tag_length, UInt64[128,120,112,104,96]))
        throw(ArgumentError("GCMAE: tag_length not a valid length"))
    end

    H = CIPH(falses(128), encryptor)
    J0 = [];
    if(IV.len == 96)
        J0 = vcat(IV, falses(31), trues(1))
    else
        s = 128 * ceil(UInt64, IV.len / 128) - IV.len
        J0 = GHASH(vcat(IV, falses(s + 64), binary(UInt64(IV.len), UInt64(64
    end
    C = GCTR(inc(J0,UInt64(32)), P, encryptor)
    u = 128 * ceil(UInt64, C.len / 128) - C.len
    v = 128 * ceil(UInt64, A.len / 128) - A.len

    S = GHASH(vcat(A, falses(v), C, falses(u), binary(UInt64(A.len), UInt64(
    T = MSB(GCTR(J0,S, encryptor), tag_length)
    C, T
end

function GCMAD(IV::BitArray, C::BitArray, A::BitArray, T::BitArray, Key::Bit

    encryptor = Nettle.Encryptor("AES128", bits2bytes(Key))

    if(C.len > 2^39 - 256)
        throw(ArgumentError("GCMAD: C too long"))
    end
    if(C.len % 8 != 0)
        throw(ArgumentError("GCMAD: C length not multiple of 8"))
    end
    if(A.len > UInt128(2)^64 - 1)
        throw(ArgumentError("GCMAD: A too long"))
    end
    if(A.len % 8 != 0)
        throw(ArgumentError("GCMAD: A length not multiple of 8"))
    end
    if(IV.len > UInt128(2)^64 - 1)
        throw(ArgumentError("GCMAD: IV too long"))
    end
    if(IV.len < 1)
        throw(ArgumentError("GCMAD: IV too short"))
    end
    if(IV.len % 8 != 0)
        throw(ArgumentError("GCMAD: IV length not multiple of 8"))
    end
    if(T.len != tag_length)
        throw(ArgumentError("GCMAD: T not of tag_length"))
    end
    if(!in(tag_length, UInt64[128,120,112,104,96]))
        throw(ArgumentError("GCMAD: tag_length not a valid length"))
    end

    H = CIPH(falses(128), encryptor)
    J0 = [];
    if(IV.len == 96)
        J0 = vcat(IV, falses(31), trues(1))
    else
        s = 128 * ceil(UInt64, IV.len / 128) - IV.len
        J0 = GHASH(vcat(IV, falses(s + 64), binary(UInt64(IV.len), UInt64(64
    end
    P = GCTR(inc(J0,UInt64(32)), C, encryptor)
    u = 128 * ceil(UInt64, C.len / 128) - C.len
    v = 128 * ceil(UInt64, A.len / 128) - A.len
    S = GHASH(vcat(A, falses(v), C, falses(u), binary(UInt64(A.len), UInt64(
    T_prime = MSB(GCTR(J0, S, encryptor), tag_length)
    if(T != T_prime)
        throw(ArgumentError("GCMAD: invalid decryption"))
        return
    end
    P
end;
```

Once implemented, I needed to test the code to ensure that it would work.

In [4]:
```
# generate a random encryption key
key = bitrand(128)

# generate a random initialization vector
IV = bitrand(64)

# generate 10KB of random plaintext data
plaintext = bitrand(80000)

# authenticated_data is any additional plaintext data we want to send and dor
# In this case, we want to send the Initialization Vector as well
authenticated_data = copy(IV)

# encrypt it
ciphertext, tag = GCMAE(IV,plaintext,authenticated_data, key)

# decrypt it
IV = copy(authenticated_data)
deciphertext = GCMAD(IV, ciphertext, authenticated_data, tag, key)

# ensure the decrypted version is the same as the original
deciphertext == plaintext
```

Out[4]:
```
true
```

Now I wanted to do some benchmarking, so I used the BenchmarkTools package.

In [5]:
```
using BenchmarkTools
using JLD

# generate a random encryption key
key = bitrand(128)

# generate a random initialization vector
IV = bitrand(64)

# make sure the IV is authenticated as well
authenticated_data = copy(IV)

version_1_benchmarks = Dict(
    "filesizes" => [],
    "mean_times" => [],
    "median_times" => [],
    "max_times" => [],
    "min_times" => [],
    "allocs" => [],
    "memories" => []
)

# benchmark sizes ranging from 100 bytes to 10000 bytes, and every 200-byte inter
for i = 100:200:10000
    plaintext = bitrand(8*i);

    benchmark_results = @benchmark begin
        ciphertext, tag = GCMAE(IV,plaintext,authenticated_data, key)
        deciphertext = GCMAD(IV, ciphertext, authenticated_data, tag, key)
    end;
    push!(version_1_benchmarks["filesizes"], 8*i);
    push!(version_1_benchmarks["mean_times"], mean(benchmark_results.times)/10000
    push!(version_1_benchmarks["median_times"], median(benchmark_results.times)/1
    push!(version_1_benchmarks["max_times"], maximum(benchmark_results.times)/100
    push!(version_1_benchmarks["min_times"], minimum(benchmark_results.times)/100
    push!(version_1_benchmarks["allocs"], benchmark_results.allocs);
    push!(version_1_benchmarks["memories"], benchmark_results.memory);
end;

save("version_1_benchmarks.jld", "version_1_benchmarks", version_1_benchmarks);
```

The benchmarking results will be displayed at the end of this report for comparison to other versions of this code.

8) Step 2: Optimize in Serial

The next step was to use some of the tips and tricks learned in class (and in online browsing) for improving Julia code performance. Below is a second version of the code, similar to the first but with some major improvements. Note how most functions that previously created copies now have in-place versions. Also note that several "for" loops have been merged, unnecessary data is no longer kept in memory, and several redundant loops have been removed.

```
In [ ]:
```

```
workspace()
using Nettle
import Base.*

R = falses(128);
R[1] = true;
R[2] = true;
R[3] = true;
R[8] = true;

tag_length = UInt64(128)

function read_file_data_bits(filename)
    s = stat(filename)
    filesize = s.size
    data = falses(filesize*8);
    open(filename) do f
        read!(f, data)
    end
    return data
end;

function MSB(X::BitArray, s::UInt64)
    X[1 : s]
end;

function MSB!(X::BitArray, s::UInt64)
    deleteat!(X, collect(s + 1 : X.len))
    X
end;

function LSB(X::BitArray, s::UInt64)
    X[end - s + 1 : end]
end;

function LSB!(X::BitArray, s::UInt64)
    deleteat!(X, collect(1:X.len - s))
    X
end;

function RS!(X::BitArray, s::UInt64)
    deleteat!(X, collect(length(X) - s + 1 : length(X)))
    prepend!(X, falses(s))
end;

function RS(X::BitArray, s::UInt64)
    RS!(copy(X), s)
end;

function LS!(X::BitArray, s::UInt64)
    deleteat!(X, collect(1:s))
    append!(X, falses(s))
end;

function LS(X::BitArray, s::UInt64)
    LS!(copy(X), s)
end;
```

```

function int(X::BitArray)
    if(length(X) > 64)
        throw(ArgumentError("BitArray to Int: Too many bits"))
    end
    reinterpret(UInt64, reverse(X).chunks)[1]
end;

function binary(bytes::Array{UInt8,1})
    n = length(bytes)
    m = ceil(UInt64, n / 8) * 8
    X = falses(m*8)
    X.chunks = reinterpret(UInt64,vcat(reverse(bytes), zeros(UInt8, m - n)))
    X.len = length(bytes)*8
    reverse(X)
end

function binary(bytes::Array{UInt8,1}, len::UInt64)
    n = length(bytes)
    m = ceil(UInt64, n / 8) * 8
    X = falses(m*8)
    X.chunks = reinterpret(UInt64,vcat(reverse(bytes), zeros(UInt8, m - n)))
    X.len = len
    reverse(X)
end;

function bits2bytes(X::BitArray)
    reverse(reinterpret(UInt8, reverse(X).chunks)[1:ceil(UInt64, X.len / 8)]
end;

function binary(v::UInt64)
    t = falses(64)
    t.chunks = [v]
    reverse!(t)
    t
end;

function binary(v::UInt8)
    if(v >= UInt128(2)^8)
        throw(ArgumentError("binary: insufficient bits for value"))
    end
    LSB(binary(UInt64(v)), UInt64(8))
end;

function binary(v::UInt64, s::UInt64)
    if(v >= UInt128(2)^s)
        throw(ArgumentError("binary: insufficient bits for value"))
    end
    LSB(binary(v), s)
end;

function inc(X::BitArray, s::UInt64)
    if(s > 63)
        throw(ArgumentError("BitArray increment: Too many bits"))
    end
    right = binary((int(LSB(X, s)) + 1) % UInt64(2)^s, s);
    vcat(MSB(X, length(X) - s), right)
end;
```

```
end;

function inc!(X::BitArray, s::UInt64)
    if(s > 63)
        throw(ArgumentError("BitArray increment: Too many bits"))
    end
    X[X.len - s + 1 : end] = binary((int(LSB(X, s)) + 1) % UInt64(2)^s, s);
end;

function Base.:*(X::BitArray, Y::BitArray)
    if(X.len != 128 || Y.len != 128 || R.len != 128)
        throw(ArgumentError("BitArray multiplication: incorrect lengths"))
    end
    Z = falses(128);
    V = copy(Y);
    for i = 1:128
        if(X[i])
            Z = Z $ V
        end

        if(V[end])
            V = RS!(V, UInt64(1)) $ R
        else
            RS!(V, UInt64(1))
        end
    end
    Z
end;

function GHASH(X::BitArray, H::BitArray)
    if(X.len % 128 != 0 || H.len != 128)
        throw(ArgumentError("GHASH: incorrect lengths"));
    end
    m = round(UInt64, X.len / 128);
    Y = falses(128);
    for i = 1:m
        Y = (Y $ X[128*(i-1) + 1 : 128*i]) * H
    end
    Y
end;

function CIPH(X::BitArray, encryptor)
    binary(Nettle.encrypt(encryptor, bits2bytes(X)), UInt64(X.len))
end

function GCTR!(ICB::BitArray, X::BitArray, encryptor)
    if(ICB.len != 128)
        throw(ArgumentError("GCTR: incorrect lengths"))
    end
    if(X.len == 0)
        return BitArray[];
    end
    n = ceil(UInt64, X.len / 128)

    CB = ICB
    for i = 1:1:n-1
        X[128*(i-1) + 1 : 128*i] = X[128*(i-1) + 1 : 128*i] $ CIPH(CB, encry
        inc!(CB, UInt64(32))
    end
    X_star = LSB(X, UInt64(X.len - 128*(n-1)))
    X[end - X_star.len + 1 : end] = X_star $ MSB(CIPH(CB, encryptor), UInt64
    X
end;

function GCMAE(IV::BitArray, P::BitArray, A::BitArray, Key::BitArray)

    encryptor = Nettle.Encryptor("AES128", bits2bytes(Key))

    if(P.len > 2^39 - 256)
        throw(ArgumentError("GCMAE: P too long"))
    end
    if(P.len % 8 != 0)
        throw(ArgumentError("GCMAE: P length not multiple of 8"))
    end
    if(A.len > UInt128(2)^64 - 1)
        throw(ArgumentError("GCMAE: A too long"))
    end
    if(A.len % 8 != 0)
        throw(ArgumentError("GCMAE: A length not multiple of 8"))
    end
    if(IV.len > UInt128(2)^64 - 1)
        throw(ArgumentError("GCMAE: IV too long"))
    end
    if(IV.len < 1)
        throw(ArgumentError("GCMAE: IV too short"))
    end
    if(IV.len % 8 != 0)
        throw(ArgumentError("GCMAE: IV length not multiple of 8"))
    end

    if(!in(tag_length, UInt64[128,120,112,104,96]))
        throw(ArgumentError("GCMAE: tag_length not a valid length"))
    end


    H = CIPH(falses(128), encryptor)
    J0 = [];
    if(IV.len == 96)
        J0 = vcat(IV, falses(31), trues(1))
    else
        s = 128 * ceil(UInt64, IV.len / 128) - IV.len
        J0 = GHASH(vcat(IV, falses(s + 64), binary(UInt64(IV.len))), H)
    end
    C = GCTR!(inc(J0,UInt64(32)), P, encryptor)
    C_len = C.len;
    u = 128 * ceil(UInt64, C_len / 128) - C_len
    v = 128 * ceil(UInt64, A.len / 128) - A.len
    prepend!(C, falses(v))
    prepend!(C, A)
    append!(C, falses(u))
    append!(C, binary(UInt64(A.len)))
    append!(C, binary(UInt64(C_len)))
    S = GHASH(C, H)
    T = MSB!(GCTR!(J0,S, encryptor), tag_length)
    deleteat!(C, collect(1:A.len + v))
    deleteat!(C, collect(C_len + 1 : C.len))
    C, T
end

function GCMAD(IV::BitArray, C::BitArray, A::BitArray, T::BitArray, Key::Bit

    encryptor = Nettle.Encryptor("AES128", bits2bytes(Key))

    if(C.len > 2^39 - 256)
        throw(ArgumentError("GCMAD: C too long"))
    end
    if(C.len % 8 != 0)
        throw(ArgumentError("GCMAD: C length not multiple of 8"))
    end
    if(A.len > UInt128(2)^64 - 1)
        throw(ArgumentError("GCMAD: A too long"))
    end
    if(A.len % 8 != 0)
        throw(ArgumentError("GCMAD: A length not multiple of 8"))
    end
    if(IV.len > UInt128(2)^64 - 1)
        throw(ArgumentError("GCMAD: IV too long"))
    end
    if(IV.len < 1)
        throw(ArgumentError("GCMAD: IV too short"))
    end
    if(IV.len % 8 != 0)
        throw(ArgumentError("GCMAD: IV length not multiple of 8"))
    end
    if(T.len != tag_length)
        throw(ArgumentError("GCMAD: T not of tag_length"))
    end
    if(!in(tag_length, UInt64[128,120,112,104,96]))
        throw(ArgumentError("GCMAD: tag_length not a valid length"))
    end

    H = CIPH(falses(128), encryptor)
    J0 = [];
    if(IV.len == 96)
        J0 = vcat(IV, falses(31), trues(1))
    else
        s = 128 * ceil(UInt64, IV.len / 128) - IV.len
        J0 = GHASH(vcat(IV, falses(s + 64), binary(UInt64(IV.len), UInt64(64
    end
    C_len = C.len
    u = 128 * ceil(UInt64, C_len / 128) - C_len
    v = 128 * ceil(UInt64, A.len / 128) - A.len
    prepend!(C, falses(v))
    prepend!(C, A)
    append!(C, falses(u))
    append!(C, binary(UInt64(A.len)))
    append!(C, binary(UInt64(C_len)))
    S = GHASH(C, H)
    deleteat!(C, collect(1:A.len + v))
    deleteat!(C, collect(C_len + 1 : C.len))
    P = GCTR!(inc(J0,UInt64(32)), C, encryptor)
    T_prime = MSB!(GCTR!(J0, S, encryptor), tag_length)
    if(T != T_prime)
```

```
        throw(ArgumentError("GCMAD: invalid decryption"))
        return
    end
    P
end;
```

Now I test this new version to ensure it still works.

In [7]:
```
# generate a random encryption key
key = bitrand(128)

# generate a random initialization vector
IV = bitrand(64)

# generate 10KB of random plaintext data
plaintext = bitrand(80000)

# authenticated_data is any additional plaintext data we want to send and don't n
# In this case, we want to send the Initialization Vector as well
authenticated_data = copy(IV)

# encrypt it
ciphertext, tag = GCMAE(IV,plaintext,authenticated_data, key)

# decrypt it
IV = copy(authenticated_data)
deciphertext = GCMAD(IV, ciphertext, authenticated_data, tag, key)

# ensure the decrypted version is the same as the original
deciphertext == plaintext
```

Out[7]:
```
true
```

And run the benchmarking again.

In [ ]:
```
using BenchmarkTools
using JLD

# generate a random encryption key
key = bitrand(128)

# generate a random initialization vector
IV = bitrand(64)

# make sure the IV is authenticated as well
authenticated_data = copy(IV)

version_2_benchmarks = Dict(
    "filesizes" => [],
    "mean_times" => [],
    "median_times" => [],
    "max_times" => [],
    "min_times" => [],
    "allocs" => [],
    "memories" => []
)

# benchmark sizes ranging from 100 bytes to 10000 bytes, and every 200-byte inter
for i = 100:200:10000
    plaintext = bitrand(8*i);

    benchmark_results = @benchmark begin
        ciphertext, tag = GCMAE(IV,plaintext,authenticated_data, key)
        deciphertext = GCMAD(IV, ciphertext, authenticated_data, tag, key)
    end;
    push!(version_2_benchmarks["filesizes"], 8*i);
    push!(version_2_benchmarks["mean_times"], mean(benchmark_results.times)/10000
    push!(version_2_benchmarks["median_times"], median(benchmark_results.times)/1
    push!(version_2_benchmarks["max_times"], maximum(benchmark_results.times)/100
    push!(version_2_benchmarks["min_times"], minimum(benchmark_results.times)/100
    push!(version_2_benchmarks["allocs"], benchmark_results.allocs);
    push!(version_2_benchmarks["memories"], benchmark_results.memory);
end;

save("version_2_benchmarks.jld", "version_2_benchmarks", version_2_benchmarks);
```

9) Step 3: Implement AES block cipher natively in Julia to remove need for external packages

The third step was to implement the 128-bit AES block cipher in Julia itself, so that Nettle is no longer required and there are no dependencies. This was done by implementing the AES cipher directly from the specification document (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf). The Galois Counter Mode code for this version is identical to that in step 2 above, except with the CIPH function replaced as below.

In [9]:

```
subbytes_table = UInt8[
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0,
];

rcon_base = UInt32[
    0x8d000000, 0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000, 0x200
    0x2f000000, 0x5e000000, 0xbc000000, 0x63000000, 0xc6000000, 0x97000000, 0x350
    0x72000000, 0xe4000000, 0xd3000000, 0xbd000000, 0x61000000, 0xc2000000, 0x9f0
    0x74000000, 0xe8000000, 0xcb000000, 0x8d000000, 0x01000000, 0x02000000, 0x040
    0xab000000, 0x4d000000, 0x9a000000, 0x2f000000, 0x5e000000, 0xbc000000, 0x630
    0xc5000000, 0x91000000, 0x39000000, 0x72000000, 0xe4000000, 0xd3000000, 0xbd0
    0x83000000, 0x1d000000, 0x3a000000, 0x74000000, 0xe8000000, 0xcb000000, 0x8d0
    0x36000000, 0x6c000000, 0xd8000000, 0xab000000, 0x4d000000, 0x9a000000, 0x2f0
    0x7d000000, 0xfa000000, 0xef000000, 0xc5000000, 0x91000000, 0x39000000, 0x720
    0x33000000, 0x66000000, 0xcc000000, 0x83000000, 0x1d000000, 0x3a000000, 0x740
    0x40000000, 0x80000000, 0x1b000000, 0x36000000, 0x6c000000, 0xd8000000, 0xab0
    0x6a000000, 0xd4000000, 0xb3000000, 0x7d000000, 0xfa000000, 0xef000000, 0xc50
    0x25000000, 0x4a000000, 0x94000000, 0x33000000, 0x66000000, 0xcc000000, 0x830
    0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000, 0x1b000000, 0x360
    0xc6000000, 0x97000000, 0x35000000, 0x6a000000, 0xd4000000, 0xb3000000, 0x7d0
    0x61000000, 0xc2000000, 0x9f000000, 0x25000000, 0x4a000000, 0x94000000, 0x330
];
RCON = [];
for i = 1:length(rcon_base)
    push!(RCON, binary(UInt64(rcon_base[i]), UInt64(32)));
end;

function SubBytes!(X::BitArray)
    if(length(X) % 8 != 0)
        throw(ArgumentError("Subbytes: not a multiple of 8 bits"))
    end
    num_bytes = floor(UInt32, length(X)/8)
    for i = 0 : num_bytes - 1
        X[i*8 + 1 : (i+1)*8] = binary(subbytes_table[int(X[i*8 + 1 : (i+1)*8]) +
    end
    return X
end;
function SubBytes(X::BitArray)
    return SubBytes!(copy(X))
end;

function SubWord!(X::BitArray)
```

```
    if(length(X) != 32)
        throw(ArgumentError("SubWord: not 32 bits"))
    end
    return SubBytes!(X)
end;
function SubWord(X::BitArray)
    if(length(X) != 32)
        throw(ArgumentError("SubWord: not 32 bits"))
    end
    return SubBytes(X)
end;

function RotWord!(X::BitArray)
    if(length(X) != 32)
        throw(ArgumentError("RotWord: not 32 bits"))
    end
    return rol!(X, 8)
end;
function RotWord(X::BitArray)
    return RotWord!(copy(X))
end;

function KeyExpansion(key::BitArray, Nk::Int64, Nr::Int64, Nb::Int64)
    i = 0
    w = falses(Nb * (Nr + 1) * 32)
    w[1:length(key)] = key;
    for i = Nk : (Nb * (Nr + 1)) - 1
        temp = w[(i-1)*32 + 1 : i*32]
        if(i % Nk == 0)
            temp = SubWord(RotWord(temp)) $ RCON[round(Int32, i/Nk) + 1]
        elseif(Nk > 6 && i % Nk == 4)
            temp = SubWord(temp)
        end
        w[i*32 + 1 : (i+1)*32] = temp $ w[(i-Nk)*32 + 1 : (i-Nk+1)*32];
    end
    w
end;

function AddRoundKey!(input::BitArray, roundkey::BitArray)
    if(length(roundkey) != 128)
        throw(ArgumentError("AddRoundKey: key of incorrect length"))
    end
    if(length(input) != 128)
        throw(ArgumentError("AddRoundKey: input of incorrect length"))
    end

    for c = 0:3
        range = c*32 + 1 : c*32 + 32
        input[range] = input[range] $ roundkey[range]
    end
    input
end;

function ShiftRows!(input::BitArray)
    tmp = input[9:16]
    input[9:16] = input[41:48]
    input[41:48] = input[73:80]
    input[73:80] = input[105:112]
    input[105:112] = tmp

    tmp = input[17:24]
    input[17:24] = input[81:88]
    input[81:88] = tmp
    tmp = input[49:56]
    input[49:56] = input[113:120]
    input[113:120] = tmp

    tmp = input[25:32]
    input[25:32] = input[121:128]
    input[121:128] = input[89:96]
    input[89:96] = input[57:64]
    input[57:64] = tmp

    return input
end

function MixColumns!(input::BitArray)
    bytes = bits2bytes(input)

    b = zeros(UInt8,4)
    for c = 0:3
        a = bytes[c*4 + 1 : c*4 + 4]
        for r = 1:4
            h = 0;
            if(a[r] >  0x7f)
                h = 0xff
            end
            b[r] = (a[r] << 1) $ (0x1B & h);
        end
        range = c*32 + 1 : c*32 + 8
        input[c*32 + 1 : c*32 + 8] = binary(b[1] $ a[4] $ a[3] $ b[2] $ a[2])
        input[c*32 + 9 : c*32 + 16] = binary(b[2] $ a[1] $ a[4] $ b[3] $ a[3])
        input[c*32 + 17 : c*32 + 24] = binary(b[3] $ a[2] $ a[1] $ b[4] $ a[4])
        input[c*32 + 25 : c*32 + 32] = binary(b[4] $ a[3] $ a[2] $ b[1] $ a[1])
    end

    return input
end

function CIPH(plaintext::BitArray, key::BitArray)
    Nk = round(Int64, length(key)/32);
    Nr = 0;
    if(Nk == 4)
        Nr = 10
    elseif(Nk == 6)
        Nr = 12
    elseif(Nk == 8)
        Nr = 14
    else
        throw(ArgumentError("AES: key of incorrect length"))
    end
    Nb = 4

    state = copy(plaintext)
    if(length(plaintext) != 128)
        throw(ArgumentError("AES: plaintext of incorrect length"))
    end

    key_schedule = KeyExpansion(key, Nk, Nr, Nb)

    AddRoundKey!(state, key_schedule[1:32*4])

    for round = 1 : Nr-1
        SubBytes!(state)
        ShiftRows!(state)
        MixColumns!(state)
        AddRoundKey!(state, key_schedule[round*(Nb * 32) + 1 : (round + 1)*(Nb *
    end

    SubBytes!(state)
    ShiftRows!(state)
    AddRoundKey!(state, key_schedule[Nr*(Nb * 32) + 1 : (Nr+1)*(Nb * 32)])

    return state
end;
```

The first thing to do now is ensure that this new CIPH function returns the same value as Nettle's version that I was using previously.

In [10]:
```
# generate a random encryption key
key = bitrand(128)

# generate 128-bit block of random plaintext data
plaintext = bitrand(128)

# validate that my encryption is the same as Nettle.jl's
my_encrypted = CIPH(plaintext, key)
nettle_encrypted = binary(Nettle.encrypt(Nettle.Encryptor("AES128", bits2bytes(ke
my_encrypted == nettle_encrypted
```

Out[10]:
```
true
```

I'd also like to test to ensure that the authentication works properly as well. This should throw an error saying that the decryption is invalid.

In [12]:
```
# generate a random encryption key
key = bitrand(128)

# generate a random initialization vector
IV = bitrand(64)

# generate 10KB of random plaintext data
plaintext = bitrand(80000)

# authenticated_data is any additional plaintext data we want to send and don't n
# In this case, we want to send the Initialization Vector as well
authenticated_data = copy(IV)

# encrypt it
ciphertext, tag = GCMAE(IV,plaintext,authenticated_data, key)

# flip a bit to simulate someone modifying the data (XOR a bit with "true")
ciphertext[10] = ciphertext[10] $ true

# decrypt it
IV = copy(authenticated_data)
deciphertext = GCMAD(IV, ciphertext, authenticated_data, tag, key)
```

```
LoadError: ArgumentError: GCMAD: invalid decryption
while loading In[12], in expression starting on line 22

 in GCMAD(::BitArray{1}, ::BitArray{1}, ::BitArray{1}, ::BitArray{1}, ::BitArra
y{1}) at ./In[6]:288
```

Now this is a fully self-contained library with no dependencies! I'll benchmark it again to see how that affected it.

In [11]:
```
using BenchmarkTools
using JLD

# generate a random encryption key
key = bitrand(128)

# generate a random initialization vector
IV = bitrand(64)

# make sure the IV is authenticated as well
authenticated_data = copy(IV)

version_3_benchmarks = Dict(
    "filesizes" => [],
    "mean_times" => [],
    "median_times" => [],
    "max_times" => [],
    "min_times" => [],
    "allocs" => [],
    "memories" => []
)

# benchmark sizes ranging from 100 bytes to 10000 bytes, and every 200-byte inter
for i = 100:200:10000
    plaintext = bitrand(8*i);

    benchmark_results = @benchmark begin
        ciphertext, tag = GCMAE(IV,plaintext,authenticated_data, key)
        deciphertext = GCMAD(IV, ciphertext, authenticated_data, tag, key)
    end;
    push!(version_3_benchmarks["filesizes"], 8*i);
    push!(version_3_benchmarks["mean_times"], mean(benchmark_results.times)/10000
    push!(version_3_benchmarks["median_times"], median(benchmark_results.times)/1
    push!(version_3_benchmarks["max_times"], maximum(benchmark_results.times)/100
    push!(version_3_benchmarks["min_times"], minimum(benchmark_results.times)/100
    push!(version_3_benchmarks["allocs"], benchmark_results.allocs);
    push!(version_3_benchmarks["memories"], benchmark_results.memory);
end;

save("version_3_benchmarks.jld", "version_3_benchmarks", version_3_benchmarks);
```

10) Step 4: Parallelization

Unfortunately, I was unable to reach this stage of the project. However, Galois Counter Mode is fully parallelizable in both the encryption and decryption directions, so I would expect significant performance improvements here.

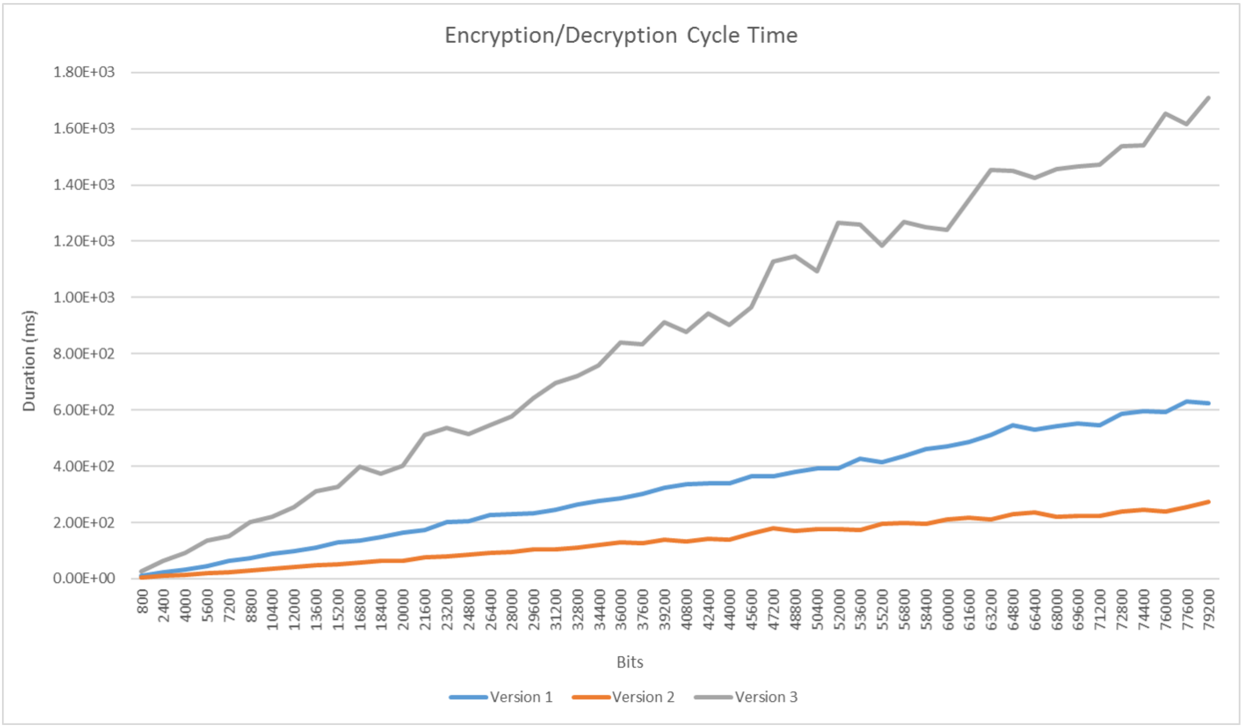
11) Performance Analysis

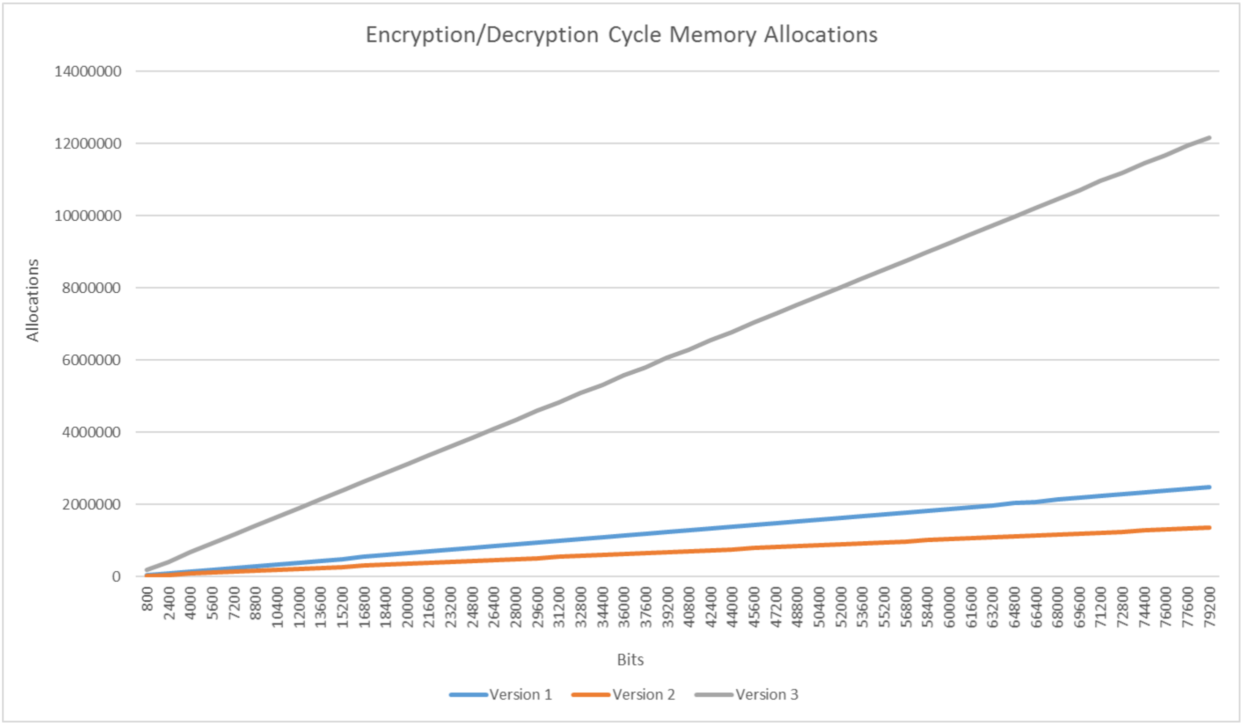
Below are several graphs showing the performance of the three code versions, including computation time, number of memory allocations, and total memory usage. Several points to note:

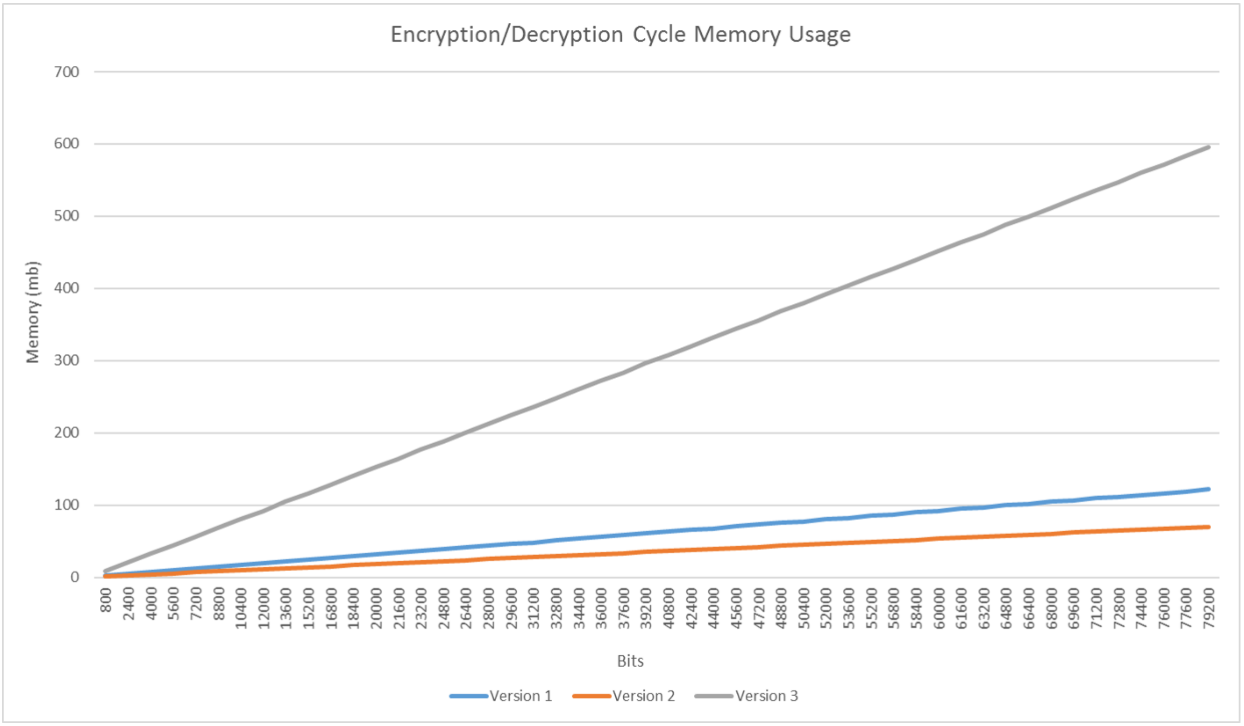
- Version 2 (serial optimization) provided a ~56% reduction in computation time, ~45% reduction in memory allocations, and ~42% reduction in total memory usage compared to version 1 (initial, non-optimized code)
- Version 3 (native AES block cipher) provided a ~626% increase in computation time, ~899% increase in memory allocations, and ~859% increase in total memory usage compared to version 2 (serial optimization)

All versions were fully functional. Below are some graphs showing the performance of the three versions. Note that the values in these graphs are from the benchmarking above, but I was unable to use any of Julia's plotting software to display them due to unknown bugs. This code sample (directly from the Julia plotting page (http://julialang.org/downloads/plotting.html)) crashes my kernel every time I run it.

```
# WARNING: running this may crash your kernel. To try your luck, change this to
a code cell and run.
workspace()
using PyPlot
x = linspace(0,2*pi,1000); y = sin(3*x + 4*cos(2*x))
plot(x, y, color="red", linewidth=2.0, linestyle="--")
```







Unsurprisingly, Nettle's AES block cipher has much better performance than the native Julia version that I put together. I believe this is probably because

12) Difficulties

In completing this project, there were several roadblocks and difficulties to overcome:

- AES block cipher is defined at the byte-level, while the Galois Counter Mode is defined at the bit-level (and does operations on chunks of bits smaller than 1 byte). Converting back and forth is very slow and confusing when trying to use BitArrays.

- Julia conversions to/from BitArrays use little-endian, while the Galois Counter Mode specification uses big-endian. Flipping back and forth is very slow and confusing.
- Julia's BitArrays have very poor support. There is no simple way to convert a 64-bit unsigned integer to a BitArray, although one would think this would be a fairly intuitive thing to do.
- It is very difficult to debug your code when it's producing ciphertext, because you have no idea what it's supposed to look like.

13) Conclusion

I'm going to call this project a success, because I managed to actually make Galois Counter Mode work in an entirely self-contained, Julia-only package. It's still much slower than libraries in other languages, and uses an outrageous amount of memory, but it works! Future work would be as follows:

- do everything in byte-based code (eliminate BitArrays entirely) to simplify the code and drastically improve performance
- much further serial optimization to do based on techniques that have been developed for encryption libraries in other languages
- parallelize everything