2017

VLC RSA Encryption

1205363

NICHOLAS TROUTMAN

Overview

In broad strokes, my goal was to encrypt and decrypt visible light communication (VLC) on purple BeagleBones. By piggybacking off the existing repositories from Shenrong (and Milad) I could focus on the actual encryption/decryption in userspace, and then optimize the performance and test the results of encrypyed and unencrypted statistics.

RSA

I was requested to encrypt the traffic in RSA, since it is both a simple and highly popular commercial encryption standard. Modern RSA keys were at least 1024 bits long, and recently have been suggested to be at least 2048 bits for commercial use.

Recommended RSA key sizes depending on lifetime of confidential data.

Lifetime of data	RSA key size
Up to 2010	1024 bits
Up to 2030	2048 bits
Up to 2031 onwards	3072 bits

The table above is provided by Shamir & Tromer's 2003 estimate, the size of the key should be proportional to the life-span of the information.
For our purposed of a semester project, and the interests of time, I have used keys that are less than 32 bits. These keys can be cracked in a matter of minutes on an ordinary laptop, in fact I calculated all of my private keys using a wolfram alpha equation.

RSA itself is a fairly straightforward algorithm, any two prime numbers (p,q) can be used as the basis for a public (d,n) and private key (e,n) . The public key is made widely available to encrypt data, and the private key is kept secret to decrypt the data. Reverse-engineering the private key with the public data is a NP-hard problem, and if a substantially better algorithm for reverse engineering in polynomial time ever comes along, most cryptographic functions would be in immediate jeopardy. Hence, the cybersecurity community has a vested interest in NP!=P, as matter of both national and worldwide importance.

Before we create the public/private keys, we need to calculate the totient of p and q, (p-1)*(q-1) = totient. We must then find n = p*q, n will be the maximum number of information you can encrypt at once, so log(2,n) is the number of bits that can be encrypted at once, and n will be

half of the public and private key. The second public key is e, find a prime number (preferably low for computational ease) that is relatively prime with the totient , 3,7,11,17 are popular choices, this number will be publicly available so it's size isn't critical. Finally we must solve (e*d)mod(totient)=1, for d, the private key. My totients were all less than 32 bits (an important fact that will be explained later), so I could solve this equation in wolfram alpha in a matter of seconds. However a naïve brute force incrementing d until a match is found took a matter of hours in R.
To encrypt (e,n): edncryptedMessage = (originalMessage^e)mod(n).

To decrypt (d,n): originalMessage = (encryptedMessage%d)mod(n).

Computation

Now that we've laid out the mathematics of encrypting and decrypting a message, I'll explain how I implemented it in C.6

The BeagleBones did not have a uint128_t or bigint library, so I made do with unsigned long long for all my computations, (64 bits). In the future I could use a bitmask of multiple unsigned long longs to "create" a dynamically sized data structure, capable of computing larger RSA computations with bigger keys, but adding a bigint library is preferable to the legwork and inefficiency of a custom bitmask.

There is an important property of modulus calculations, that every individual arithmetic operation can be modulo'd. example: (a^3)mod(b) = ((a*a)mod(b) * a)mod(b)). Thus if our original message is less than sqrt(2^64) = 32 bits, then we can run all possible RSA computations on it. So our possible "workspace" can only work with less than 32 bits at a time, rounding down to the nearest byte, we get a frame of 3 to

We've now established that we will be encrypting 3 bytes at a time of our message, but it is not a 1 to 1 byte substitution. If our n is less than 2^(8*3)=2^(24), then we won't be able to decrypt the values that lie between n and 2^(24). We can prove this by the pigeon-hole principal, if we are working in modulo n, there can only be n distinct results, if we try to encrypt/decrypt (n+x), it will decrypt to the same value as x. If we want to keep the full range of values, n must be greater than or equal to the frame size.

The actual implementation uses:
p = 55639

q = 68767

d = 1093143631

e=7

n=3826127113, log(2,n)= 31.882

Thus we can fully encrypt 31 bits at a time, but for simplicity, we will be using 3 bytes at a time (24), and padding the rest of the data (1 byte). We could use a smaller n, one with 25 bits, and we would only have to pad a single bit, which would increase throughput and speedup performance. This smaller key would be less secure, but it is already trivial to crack, so not much of a tradeoff.

To actually compute the exponent, we use a recursive exponentiation and modulo function:

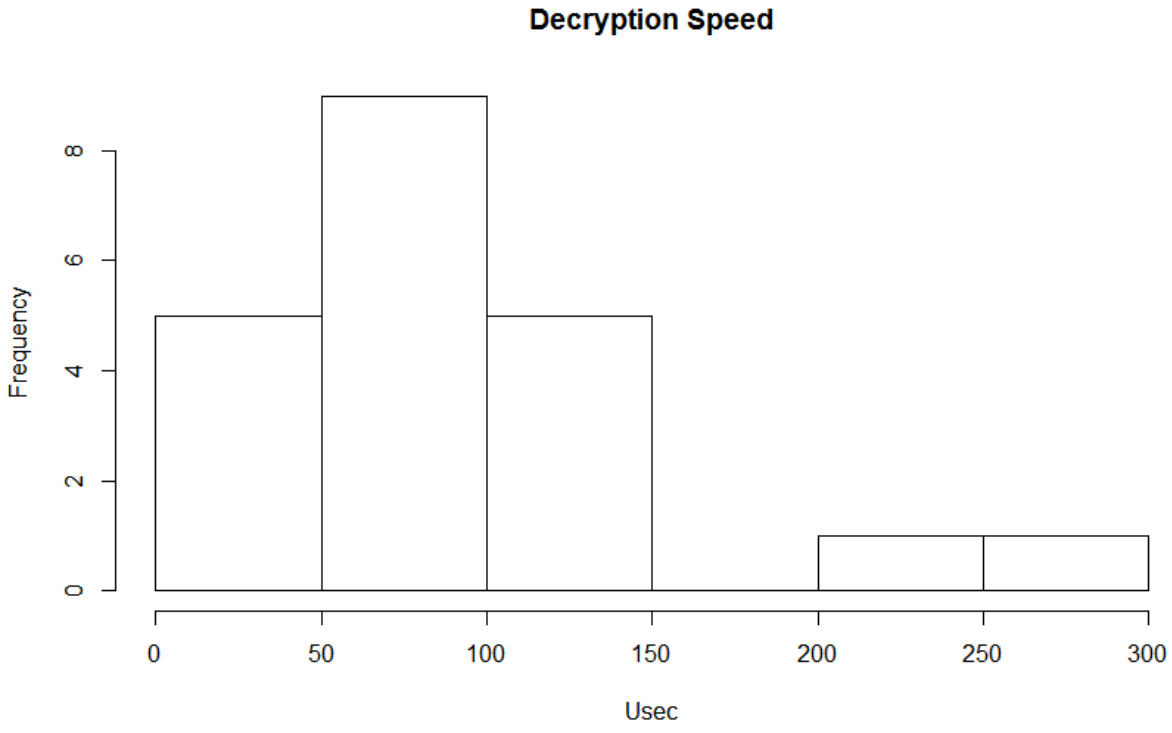
```
unsigned long long expo(unsigned long long a, unsigned long long b, unsigned long long n)
{
    printf("a = %llu",a);
    printf("\t b = %llu\t",b);
    printf("n = %llu\n",n);
    if (b==1)
        return (a%n);
    if (b==2)
        printf("Squared  = %llu mod %llu = %llu\n   ", (a)*a, n,(a*a)%n);
        printf("Squared  = %llu mod %llu = %llu\n", (a)*(a*n), n, (a%n)*(a*n)%n);
        return ((a*a)%n);

    if (b%2==0){
            return ((expo(expo(a,b/2,n),2,n))%n);
    }
    else{
        return ((a*expo(expo(a,(b-1)/2,n),2, n) )%n);
    }
}
```

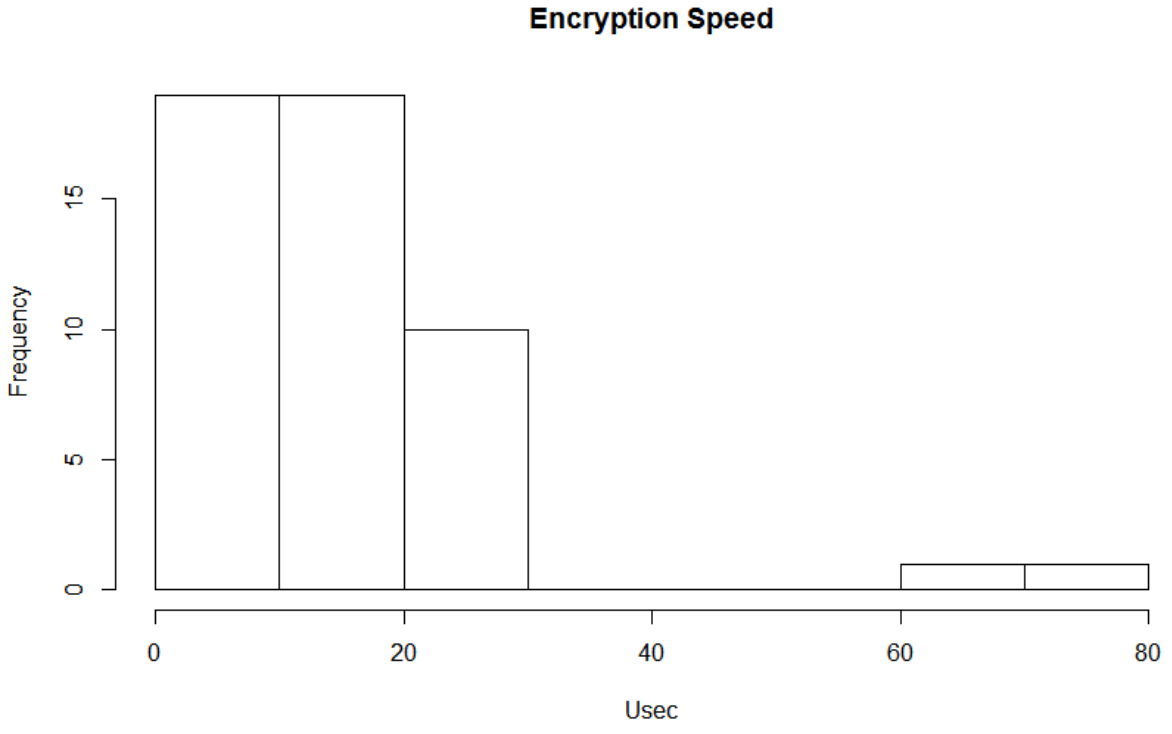
This works off the principal that a^10 is equivalent to (a^8)*(a^2) = ((a^2)^2)^2 * (a^2). This simplified version takes only 5 multiplication values, and it scales up to the extraordinarily large exponentiation needed for RSA.

Analysis

Time:
Timings in nanoseconds of encrypting functionality of a single frame.

	Mean	Std
Encrypt	14.26	13.432
Decrypt	94.3	58.639





This data is reasonable, since on average the encrypted message is larger than the unencrypted message, and d>e, so there are more algebraic calculations necessary. This can be parallelized

though, since each encrypted frame is independent, and can be sped up with less buffer space/make it more efficient.

Space: The Size of each message will increase by ceil((n-1)/3)*4 – n, because other than the header, we are expanding each byte by 4/3.
Throughput: Since the total size will be very close to 4/3*n, the throughput will become ¾ of the original value.
Error Probability: Since each byte is virtually dependant on each bit of the frame (4 bytes), it increases the severity of each individual flipped bit by a factor of 4. Thus the message cohesion is greatly reduced by this encryption, but every modern day encryption protocol increases the "connectivity" and effect of each individual bit. Otherwise the protocol would only be a series of independent Caesar ciphers, which are very vulnerable to statistical analysis of the frequency of occurance.

Output

```
root@beaglebone:/home/debian/RSA/syin2-real-time-pru-vlc-515f275f035d# ./example/send "Hello world"
11
0: H = 72
1: e = 101
2: l = 108
3: l = 108
4: o = 111
5:  = 32
6: w = 119
7: o = 111
8: r = 114
9: l = 108
10: d = 100
0: 20
1: 16
2: 11
3: 12
0: g = 103
1: ▒ = 215
2: ▒ = 136
3: ▒ = 207
4: * = 42
5: ▒ = 201
6: J = 74
7: ▒ = 185
8: G = 71
9: ▒ = 201
10: ▒ = 170
11:  = 31
12: ▒ = 199
13: r = 114
14: k = 107
15: ▒ = 176
16: ( = 40
Message sent successfully... size = 18
```

```
The node is currently listening on the light signals...
STRLEN(SINEBUF) = 13
FRAMES = 4
0: ▒ = 215
1: ▒ = 136
2: ▒ = 207
3: * = 42
4: ▒ = 201
5: J = 74
6: ▒ = 185
7: G = 71
8: ▒ = 201
9: ▒ = 170
10:  = 31
11: ▒ = 199
12: r = 114
0: 102
1: 84
2: 83
3: 84
4: 84
5: 85
6: 29
7: 30
8: 84
9: 229
10: 30
11: 30
12: 30
Encrypted: □▒*▒J▒Gr▒rk▒
Decrypted: Hello wor"n

Only 16 Characters:
Encrypted: □▒*▒J▒Gr▒
Decrypted: Hello wor"n
```

The top image is a screenshot of a successful send.c “Hello world”, notice that it breaks down the message char by char unencrypted, encrypts it while timing how many nanoseconds it took to encrypt each frame, and then prints each new encrypted char before sending.
The bottom image is a screenshot of a (mildly) successful receive.c. It prints the received buffer from the ARM, times how long it takes to decrypt the frames, and then outputs the original buffer and it’s decrypted counterpart.
Take notice that the received buffer from ARM is different than the sent message. (The received buffer is intentionally offset by 1 to get rid of the header ‘g’, so received[12] = sent[13], but sent[14] and sent[15] are not received by the receiver, and thus it is impossible to decrypt the last frame successfully. This is not an encryption problem, but a sensitivity/reliability problem with the board, as we can see, when the proper values are given, it decrypts successfully.

Optimization (O3 and O0)

-O0:

	Mean	Std
Encrypt	9.9	4.327
Decrypt	88.8	7.639

-O3:

	Mean	Std
Encrypt	10.625	20.742
Decrypt	85.25	60.84

We see that the O3 and O0 opcodes both make a significant improvement to the optimization and runtime of the encryption/decryption of a singular frame. Overall, -O0 seems to have be more usable because of it’s much smaller standard deviation.