A Survey of Random Numbers in Cryptography

Paul Adams
University of Washington
Department of Electrical Engineering
EE 595 Advanced Topics in Communication Theory
Winter 2016

***Abstract*—Random number generators (RNG) are a central element of almost any information system's security profile. As an example, they are often used to bootstrap trust between two communicating digital entities. This project set out to be a survey of RNGs in cryptographical applications, but ended up as more of a meander. While a few different topics were explored, including examples of good and bad software and hardware RNGs, in addition to methods for testing quality of RNGs, an exercise was eventually found which tied together the criticality of robust entropy sources for real-world applications. In this case, an RSA simulation was created to demonstrate the catastrophic failure arising from using inadequate random streams.**

I. Introduction

The first question we might ask is how to establish the quality of a random number generator. In order to evaluate the qualities of a random stream, we should have an understanding of the characteristics that secure cryptosystems share.

While mostly a philosophical question, it is relevant to cryptography that our understanding of randomness is *via negativa*. One might ask, does random exist as something more than simply the name we assign to uncorrelated observations? In the context of testing random numbers, software will run a suite of tests that verify the stream does not have any known bad qualities. That is, the tests demonstrate necessary but insufficient qualities of RNGs. Fundamentally, the nature of a random stream is that it's elements defy correlation and consequently there exists no exhaustive set of conditions to describe perfect randomness given a finite set of observations.

In short, we say that a random stream must be completely unpredictable. This fits well with the intuitive sense that secrets shouldn't be guessable. To this end, a common test of RNG performance is the *next bit test*, wherein it is verified that given every entry of an n - bit stream, predicting $n+1$ is equiprobable for any n.

I propose that observations meeting the following conditions are unpredictable.

- equally likely
- independent

To generate observations meeting the above conditions, we require a source of entropy. In an information theoretic sense, entropy is simply the information density of a stream. The entropy of a perfectly random stream should be equal to the length of the stream. In practice, only hardware RNGs can truly meet these conditions in the limit.

In practical cryptosystems, sourcing entropy from natural processes via hardware RNGs is too costly. For this reason the majority of cryptosystems use entropy sourced from an algorithm and implemented with software. The algorithms that generate these random numbers are inherently deterministic and fundamentally lack the required conditions proposed above. The design then attempts to appear random for as long as possible and are thus designated Pseudo-random number generators (PRNG).

Due to their formulaic nature, PRNGs must meet an additional condition to achieve reasonable unpredictability

- equally likely
- independent
- unknown (hidden) state

If the state of a pseudo-random stream is known or can be guessed, then all following numbers are immediately known. We can describe the state of a PRNG as a combination of the algorithm and some reference point in the stream. The first describes how the random samples are generated, while the second reveals where in the stream the current iteration sits. Typically, the initial value, or the seed, provides this reference. Further, since the space of cryptographically robust PRNG algorithms is relatively small, it is possible to guess the algorithm, or a close-enough variant through brute-force methods. This makes the seed value the critical component of the state.

II. RANDU

A. Description

RANDU is the classic example of an PRNG whose output is uniformly distributed but whose samples are not in-

dependent. It is defined with the following recurrence relation: Given an initial seed V_0, subsequent numbers are given as

$$V_{j+1} = 65539 \cdot V_j \mod 2^{31} \tag{1}$$

The numbers generated are uniformly distributed over $[1, ..., 2^{31} - 1]$ yet fall into a pattern when grouped into tuples of three or more. Figure 1 shows the distribution as points in a three-dimensional grid at various camera angles.

In the first, the points appear to be evenly distributed over the space. In the second, a pattern is beginning to appear, while the third shows the numbers aligning into planes.

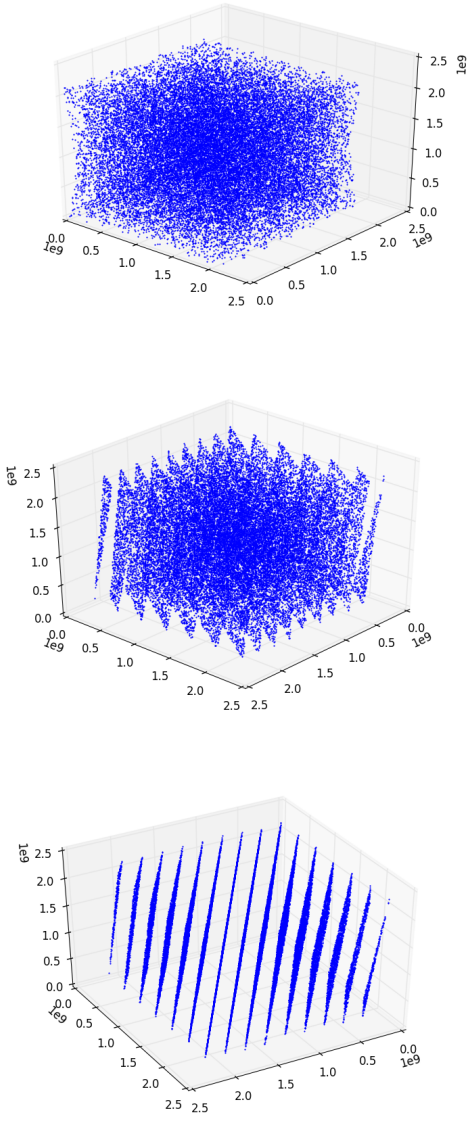

Fig. 1: RANDU Generated Numbers in 3-Tuples at Various Angles

This example of failing the spectral test holds for all dimensions greater than 2. It also points to the necessary property of PRNGs that in addition to having uniform distributions, the numbers must be free of correlations. In the case of the RANDU generator the numbers clearly fall into planes.

III. RSA Simulation with Bad PRNG

It is easy to ask why a robust random stream actually matters when there are sophisticated and vetted algorithms that seem to do the heavy lifting. After all, in the case of RSA, the random number generator provides only the initial boost before creating very large numbers that are near impossible to factor.

In order to test the effects of an unreliable random stream, I designed an RSA protocol simulation programmed in the julia language [1] to mimic a practical application.

A. Scenario

To tie the problem to a realistic setting, the following scenario is given.

A small online investment startup has contracted with an affordable, and new, security firm to provide secure, randomized passwords for its customers. While the passwords themselves are generated with a decent PRNG, the security firm struggles with the startup's request to use RSA instead of their preferred elliptic curve based algorithms. Due to scheduling pressures, they opt to re-appropriate an open-source protocol purporting to implement a secure RSA protocol. While the security firm attempts to do due diligence on the implementation, they overlook testing the quality of the built-in random number generator.

Additionally, an adversary, Eve, has discovered a vulnerability in the startup's network implementation that allows her to intercept and cache all outgoing messages from the startup. Caching the RSA encrypted passwords along with their respective moduli allows her to perform offline processing to test the strength of the encryption used by the investment startup.

B. Forward Problem

The program has two primary steps, which I will to refer to as the forward problem, wherein a large set of RSA channels is generated, and the inverse problem, wherein offline processing is used to break the system.

For context, in an RSA scheme the encryption and decryption functions are given, respectively, as

$$c = E_k(m, e) = m^e \mod n \quad (2)$$

$$m = D_k(c, d) = c^d \mod n \quad (3)$$

where m is the message, e is the public key, n is the modulus, d is the private key, and c is the cipher.

Simulation of the generation and transmission of secure passwords to customers program contains a function to generate a large number of RSA ciphers, composed of ten character random passwords, and then cache the details to a file for post-processing. Each message creates a new channel, generating a set of public/private keys along with the modulus. The figure below illustrates one cycle of the forward problem. Additionally, for each message, the modulus and cipher are observed by Eve and cached.

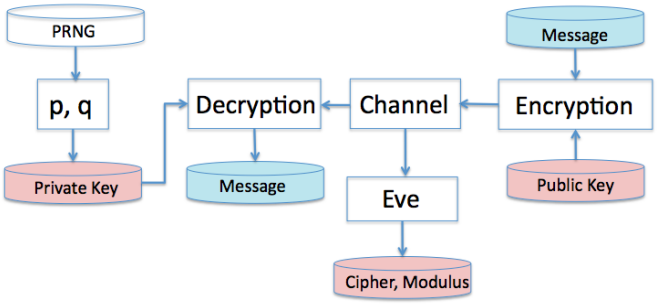

Fig. 2: Forward Problem

C. PRNG Vulnerability

From Fig. 2, it can be seen that the private key is a function of the randomly generated primes p and q. Specifically,

$$n = pq \quad \text{modulus} \quad (4)$$

$$\phi = (p-1)(q-1) \quad \text{totient} \quad (5)$$

$$d = e^{-1} \mod \phi \quad \text{private key} \quad (6)$$

Herein lies the vulnerability introduced by a bad PRNG. If by chance two separate moduli share a common factor, the private key for both channels can easily be deduced, as discussed in more detail in the following section.

Ordinarily, under 1028-bit strength, a collision of prime factors would occur with probability nearly zero. However, this assumes a cryptographically secure PRNG. With a bad random stream feeding p's and q's, the probability of collision is increased significantly. This effect was surveyed in detail in the seminal research of Heninger *et al.* [2]. The researchers harvested a large collection of public keys from RSA and DSA for TLS and SSH hosts and then sought for non-trivial moduli. In the case of RSA for TLS hosts, the private keys of 0.50 % were recovered due to random streams with insufficient entropy.

D. Inverse Problem

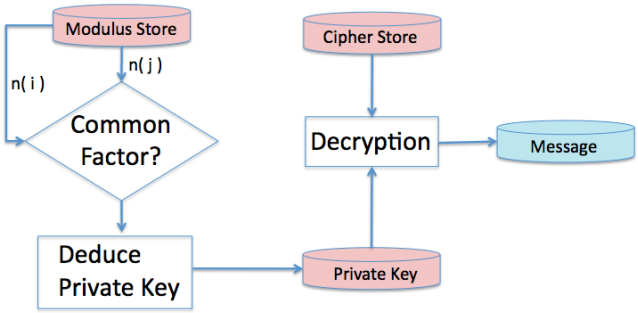

Fig. 3: Inverse Problem

Suppose Eve has found two independent moduli, n_i and n_j and has further found that they share a common factor by testing,

$$\text{GCD}(n_1, n_2) \neq 1 \quad (7)$$

As a side note, if she has many moduli, millions or more to test, it is not computationally infeasible to compute the pairwise greatest common denominator due to Euclid's extended algorithm. Security of RSA rests in the difficulty of factoring large numbers composed of primes. That is, given n, it is difficult to find p and q. However, given two n's, it is easy to compute their GCD.

For the pair above, it is straightforward to posit $p = \text{GCD}(n_i, n_j)$ and further, $q_i = n_i/p$ and $q_j = n_j/p$. Because the moduli share a common factor, it is trivial to find their respective p, q pair. From these, the private key can be found using (5) and (6) and the cipher decrypted using (3). Put simply, if a pair of common factors is found for two moduli, the private keys for both channels are immediately compromised.

E. Numerical Example of Compromise

Suppose 10 million channels are generated, each containing a private, public key pair, and a modulus. Over the RSA channel, a message is encrypted, transmitted, received, and decrypted with the private key. This process is mostly captured in Figure 2 and repeated here for context.

Further suppose an adversary harvests the modulus and cipher associated with the 10 million RSA channels and begins to search the space for moduli with common prime factors. For this example, let the message be "6qNu0qfmsM",

a randomly generated password and let the modulus bit depth be 31 bits. Making use of the function written for this purpose, we can display the output of the Julia prompt to show the corresponding encrypted elements

```
cipher = send_rsa("6qNu0qfmsM", pubkey, n)
   10-element Array{BigInt,1}:
        347800891852005865
         42670716049858146
        579594145838870300
        275167124031112302
        450351544165447506
         42670716049858146
        214153226707085291
        341372119076152163
        144771398172410886
         62087393764008425
```

Now, let the i^{th}, j^{th} pair of moduli tested be,

$$n_i = 776945765374771193 \tag{8}$$

$$n_j = 682044630882172907 \tag{9}$$

Taking the GCD of the above pair yields

$$q = 839262571 \tag{10}$$

From here, we can easily compute the corresponding prime factors of both moduli as

$$p_i = \frac{776945765374771193}{839262571} = 812671157 \tag{11}$$

$$p_j = \frac{682044630882172907}{839262571} = 925747943 \tag{12}$$

Supposing we are only interested in the message corresponding to the i^{th} cipher, knowing the public key and using (3), we now compute its private key as,

$$d = 1073741827^{-1} \mod 812671156 * 839262570 \tag{13}$$

$$d = 505876514661585763 \tag{14}$$

Given the private key, the cipher, and the modulus we can again make use of modular exponentiation and compute the first element of the cipher,

$$m[1] = c[1]^d \mod n \tag{15}$$

$$54 = 347800891852005865^{505876514661585763} \mod n \tag{16}$$

Using an ASCII lookup, the number 54 corresponds to the numeric digit "6", the first element of our secret password. Utilizing another routine from my purpose-built crypto library, the remaining elements are computed as

```
julia> msg = recv_rsa(cipher, d, n)
10-element Array{Char,1}:
 '6'
 'q'
 'N'
 'u'
 '0'
 'q'
 'f'
 'm'
 's'
 'M'
```

Which matches the original message of "6qNu0qfmsM". Thus it can be seen that RSA fails catastrophically when two moduli are found with shared factors. It is emphasized that this event would be virtually nonexistent given 1028-bit moduli and high entropy sources of prime factors. In this case, the failure is due solely to a bad PRNG generating duplicate primes.

F. Simulation Parameters and Implementation Notes

Given computing and time constraints, I limited the level of encryption 32 bits or less and used the RANDU PRNG. I used the outline of [3] for generating the channel keys, with the requirement that p, q be co-prime with the public key.

Additionally, an implementation of Euclid's extended algorithm was required for computing modular multiplicative inverses, as well as an implementation of modular exponentiation. For the latter, I used an adaptation of pseudocode attributed to Bruce Schneier in the Wikipedia article for modular exponentiation [4].

Without a fast implementation of modular exponentiation the simulation of RSA would not have been feasible. I started with a naive computation, leveraging Julia's interface to Gnu MultiPrecision Library's `BigInt` data type. While using the `BigInt` type was necessary, given the size of the numbers, arbitrary precision arithmetic carried more computational overhead than was feasible to scale. Schneier's algorithm exploits bit-shifting to achieve computations similar to (1) in a few dozen iterations.

An additional computational constraint had to do with the shared-factor search. Initially, I wrote a routine to search the entire key space pairwise. Even though computing the GCD of arbitrarily large numbers is very fast, the space is

large. If N moduli are generated, the unique, pairwise space is given as

$$\sum_{i=1}^{N} i \tag{17}$$

For $N = 1000000$ this amounts to 500000500000 GCD operations. Heninger, *et al.*, achieve greater efficiency using a special algorithm, but that level of effort seemed outside the scope of this paper. Therefore, for the purpose of the simulation, I varied the level of encryption bits until a sample set on the order of 1-10 million moduli could be identified where the number of unique p, q pairs was less than the total number of pairs. This would imply shared factors and allow me to back out the decryption on a per case basis rather than testing every one. That is, given this is a simulation and I have p, q in hand, it is much easier to check for duplicates than it is to compute the pairwise GCD over the entire set.

After the forward and inverse routines were programmed and then tuned for efficient use, I found that generating 1-10 million RSA channels took around 30 minutes on my 7 year-old Macbook Pro. Solving the inverse problem was much quicker for the reasons outlined in the above paragraph. This sort of time scale allowed for multiple iterations with different parameters.

IV. Discussion of Results

Number of Bits	Unique Primes	Total Primes	Compromised
28	1979328	24499998	91.92%
29	3801933	24499998	84.48%
30	7316030	24499998	70.14%
31	14096925	24499998	42.46%
32	24499998	24499998	0.00%
32	52551186	199999998	73.72%

Fig. 4: RSA Sim Results

Using the RANDU PRNG, collisions of primes occurred greater than half the time for most cases. I ran the simulation using various bit depths, 28-32 as seen in figure 4. Clearly the number of compromised keys decreased as the number of bits in the modulus increased. For each independent RSA channel there are two moduli and four large primes, therefore each channel presents two opportunities for failure. These numbers are represented in the columns labeled, "Unique Primes" and "Total Primes". Given a PRNG with sufficient entropy, all the primes should be unique.

Due to an indexing bug the total number of primes generated was less than the intended value of 25 million, however the percentage of compromised channels is still accurate. It can be seen that with a space of approximately 25 million primes, 32 bits of modulus depth there are no collisions. However, increasing the space to nearly 200 million primes caused it to fail spectacularly.

After running the tests that generated the above results, I attempted to find a collision with a more robust PRNG. I chose the Mersenne twister algorithm, which is not considered cryptographically secure, but is nevertheless a robust PRNG. With a space of 200 million primes there was not one collision. This is not surprising and underlines the characteristic of PRNGs that when one fails, it is not subtle. A lesson similar is that quality of PRNGs is not measured in degrees but decibels. Typically a bad PRNG fails very quickly, whereas one that is robust, but not quite the caliber needed for cryptography takes several orders of magnitude more effort to expose its weakness.

V. Conclusion

My somewhat random walk through randomness in cryptography included additional stops at the NIST Randomness Beacon, which makes full entropy bits available on the web using quantum observations, and the TestU01 c library of PRNG tests. I initially had intended to study and characterize traits of a spectrum of software and hardware RNGs. This led naturally to wondering just how insufficient randomness can affect practical security applications. The research by Heninger *et al* conducted the first harvest and analysis of a long known weak spot in RSA and provided the surprising results that even in 2012, when much of the theory of software sourced generators and OS-sourced entropy is well-developed and understood, a non-trivial fraction of internet hosts use their random numbers in a way that enables RSA to fail.

By simulating an RSA implementation and testing the effect of insufficiently random primes, I was able to tie the theory to a practical application and gain a much deeper appreciation for proper implementation of cryptosystems. Even otherwise robust and secure systems like RSA can fail spectacularly when carelessly implemented.

References

[1] http://julialang.org

[2] N. Heninger, Z. Durumeric,E. Wustrow,J. Halderman, *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices*

[3] A. Kak *Lecture 12: Public-Key Cryptography and the RSA Algorithm* https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture12.pdf

[4] B. Schneier https://en.wikipedia.org/wiki/Modular_exponentiation

VI. Code Listings

A. RSA Simulation

```
#!/usr/local/bin/julia
using DataFrames
include("my_crypto.jl");

# Params
pubkey = BigInt(1073741827);
n_bit_modulus = 30;
n_msg_str = 10;
resim_msgs = true;
do_fast = true;
do_crack = true;
N =  10000^2; #number of sim'd messages
rng_type = set_rng_type("randu");
#= cache_file = ".$(rng_type)_$(N)_msgs_$(n_bit_modulus)_bits.csv"; =#

function message_loop(N, n_bit_modulus)
    global cache_file
    cf = open(cache_file, "w");
    m = sqrt(N);
    for i = 1:N
        msg = randstring(n_msg_str);
        if i % m == 0
            print("block $(Int(i/m)) -> ");
        end
        pubkey, n, d, p, q = new_channel(n_bit_modulus);

        # Cache to file
        if do_fast
            write(cf, "$p;$q\n");
        else
            #= C = send_rsa(msg, pubkey, n); =#
            write(cf, "$msg;$p;$q;$n\n");
        end
    end
    close(cf);
end

function pairwise_crack(mods)
    cnt = 0;
    correct = 0;
    for j = i+1:length(mods)
        p = gcd(mods[i], mods[j]);
        if p != 1
            cnt += 2; # 2 private keys comprimised per collision
            println("Found collision $cnt, Fraction: $(cnt/i)")
            # Retrieve private key
            P1 = crack(n1, p, ciph1);
            P2 = crack(n2, p, ciph2);
            correct += P1 == df[j, :msg];
            correct += P2 == df[i, :msg];
            # Print results
```

```
            println("\tOriginal Msg: $(df[j, :msg]), Cracked: $P1")
            println("\tOriginal Msg: $(df[i, :msg]), Cracked: $P2")
            println("\t% Correct: $(correct/cnt)\n")
        end
    end
end

function fast_check()
    if do_fast
        df = readtable(cache_file, separator=';',
                names=[:p, :q]);
    else
        df = readtable(cache_file, separator=';',
                names=[:msg, :p, :q, :modulus]);
    end
    p = df[:, :p];
    q = df[:, :q];
    s = [p;q];
    k = length(unique(s));
    r = length(s);
    println("s: $r, unq: $r")
    if length(unique(s)) < length(s)
        println("check it")
    end
    rf = open("bigresults.txt", "a+");
    write(rf, "$k;$r\n");
    close(rf);
end

function crack(n, p, cipher)
    q = div(n, p);
    phi = (p-1)*(q-1);
    _, d, _ = ext_euclid(pubkey, phi);
    P = join(recv_rsa(cipher, d, n));
    return P
end

function test_channel()
    msg = "6qNu0qfmsM";
    pubkey, n, privkey, p, q = new_channel(n_bit_modulus);
    C = send_rsa(msg, pubkey, n);
    println("$C, $n, $privkey, $p, $q")
    p = recv_rsa(C, privkey, n);
    println("$p, $msg")
    if join(p) == msg
        println("Channel looks good")
    end
end

function test_cracker()
    msg1 = randstring(n_msg_str);
    msg2 = randstring(n_msg_str);
    p1 = gen_rand_prime(n_bit_modulus - 1);
    p2 = gen_rand_prime(n_bit_modulus - 1);
    q = gen_rand_prime(n_bit_modulus - 1);
    n1 = p1*q;
```

```
    n2 = p2*q;
    ciph1 = send_rsa(msg1, pubkey, n1);
    ciph2 = send_rsa(msg2, pubkey, n2);
    p = gcd(n1, n2);
    if p != 1
        println("Collision found: q: $q, p2: $p2")
        P1 = crack(n1, p, ciph1);
        P2 = crack(n2, p, ciph2);
        # Print results
        println("\tOriginal Msg1: $msg1, Original Msg2: $msg2")
        println("\tCracked      : $(P1), Cracked:       $(P2)")
    end
end

function param_sweep()
    rng_type = set_rng_type("randu");
    global cache_file

    for n_bit_modulus = 32:32
        cache_file = ".$(rng_type)_$(N)_msgs_$(n_bit_modulus)_bits.csv";
        message_loop(N, n_bit_modulus);
        fast_check();
    end
end

function main()
    test_channel();
    test_cracker();
    # Generate rsa msgs
    if resim_msgs
        message_loop(N);
    end

    # Try to crack
    if do_crack
        fast_check();
    end
end
```

B. My Crypto Lib

```
global v_prev = 13; #init state of randu
global rng_type

function set_rng_type(this)
    global rng_type = this;
    return rng_type
end

function gen_rand_prime(n)
    global v_prev;
    global rng_type;

    v_cur = 0;
    done = false;
```

```
    while !done
        if rng_type == "mersenne"
            v_cur = mersenne_twister();

        elseif rng_type == "randu"
            v_cur = mod(65539*v_prev, 2^n);

        end
        # repeat if number not prime
        v_prev = v_cur;
        if isprime(v_cur)
            done = true;
        end
    end
    return v_cur
end

function modular_pow(base, expon, modulus)
    # fast method by Bruce Schneier adapted from
    # https://en.wikipedia.org/wiki/Modular_exponentiation
    result = BigInt(1);

    base = BigInt(mod(base, modulus));
    expon = BigInt(expon);
    modulus = BigInt(modulus);

    while expon > 0
        #= println("result: $result, expon: $expon"); =#
        if mod(expon, 2) == 1
            result = mod((result * base), modulus);
        end

        expon = expon >> 1;
        base = mod((base * base), modulus);
    end
    return result
end

function new_channel(n)
    n_bit_modulus = n;
    pubkey = BigInt(1073741827);
    p, q = 1, 1;

    # generate random p/q prime pair
    done = false;
    while !done
        p = gen_rand_prime(n_bit_modulus-1);
        q = gen_rand_prime(n_bit_modulus-1);
        # Verify numbers coprime with pubkey
        if mod(p, pubkey) != 1 && mod(q, pubkey) != 1
            done = true;
        end
    end

    # compute modulus
    n = BigInt(p*q);
```

```
    phi = (p - 1)*(q - 1);

    # compute secret key, d
    _, privkey, _ = ext_euclid(pubkey, phi);
    return (pubkey, n, privkey, p, q)
end

function send_rsa(msg, pubkey, n)
    msg_num = [Int(M[1]) for M in msg];
    cipher = [modular_pow(x, pubkey, n) for x in msg_num];
    return cipher
end

function recv_rsa(c, prikey, n)
    plain = [modular_pow(y, prikey, n) for y in c];
    plaintext = [Char(p) for p in plain];
    return plaintext
end

function ext_euclid(a, b)
    a0, b0, = a, b;
    t0, t = 0, 1
    s0, s = 1, 0
    q = div(a, b)
    r = a - q*b
    while r > 0
        tmp = t0 - q*t;
        t0, t = t, tmp;
        tmp = s0 - q*s;
        s0, s = s, tmp;
        a, b = b, r;
        q = div(a, b);
        r = a - q*b;
    end
    r = b;
    if s < 0
        s = mod(s, b0);
    end
    if t < 0
        t = mod(t, a0);
    end
    return (r, s, t)
end

function totient(n)
    P = 1;
    for p in keys(factor(n))
        if isprime(p)
            P *= (1 - 1/p);
        end
    end
    return round(Int64, P*n)
end

mt = MersenneTwister(29);
function mersenne_twister()
```

```
    # default mersenne twister wrapper
    x = parse(dec(rand(mt, UInt32, 1)[1]));
    return x
end

function abc2num(a)
    num = [parse(Int, s) for s in a];
    num -= parse(Int, 'a');
    return num[1]
end

function num2abc(n)
    a_offset = 97;
    abc = Char(n + a_offset);
    return abc[1]
end
```

C. Rabbit

```
#!/usr/local/bin/julia

# Packages
using JLD
using PyPlot

# Functions
function run_rabbit_run(test, n)
    print("Run rabbit for $test and n = $n...");
    binname = join(["data/", test, ".bin"]);
    rabname = join(["data/", test, ".rab"]);
    prog = pwd() * "/run_rabbit";
    cmd = `$prog $binname $n`;
    run(pipeline(cmd, stdout=rabname));
    println("Success.")
end

function parse_rabbit_parse(test)
    print("Parue rabbit for $test...");
    rabname = join(["data/", test, ".rab"]);
    x = Dict();

    # colon, spaces, word, spaces, *****
    name_reg = Regex("(\\w+)\\s*test:\n");
    re = Regex("p-value of test\\s+:\\s+(.+)\n");

    # retrieve rabbit results
    f = open(rabname, "r");
    name = "";
    # parse p-values
    for line in readlines(f)
        if ismatch(name_reg, line)
            m = match(name_reg, line);
            name = m.captures[1];
            try
                name = name[collect(search(name, "_"))[1] + 1:end];
```

```
            catch
                continue;
            end
        elseif ismatch(re, line)
            m = match(re, line);
            score = m.captures[1];
            score = replace(score, "*", "");
            score = replace(score, " ", "");
            score = replace(score, "eps", "$(eps())");
            score = eval(parse(score));
            if haskey(x, name)
                name = name*"2";
            end
            println("algo: $test; test: $name; p-value:  $(score);");
            x[name] = score;
        end
    end
    close(f);
    println("Success.");
    return x
end

# Main
d = Dict();
d["randu"] = Dict();
d["matV5"] = Dict();
d["mersenne"] = Dict();

d["randu"]["n"] = 10.^[3, 4, 5, 6];
d["matV5"]["n"] = 10.^[6, 7, 8, 9, 10, 11];
d["mersenne"]["n"] = 10.^[8, 9, 10, 11, 12];

#= testnames = ["randu"]; =#
do_plot = false;

# for each rng output, run Rabbit for various n, parse results and cache
for test in keys(d)
    for (i, N) in enumerate(d[test]["n"])
        println(repeat("*", 60)*"\n");
        run_rabbit_run(test, N);
        d[test]["score"] = parse_rabbit_parse(test);
        println(repeat("*", 60)*"\n");
    end
end

if do_plot
    figure(j)
    x = collect(values(D[test][i]));
    plot(x, "-*")
    xticks(0:length(x) - 1, collect(keys(D[test][i])))
    xticks(0:length(x) - 1, collect(keys(D[test][i])), rotation=90)
    savefig(test*".png");
end
```