PRGs and PSEUDO-RANDOM NUMBER GENERATION

Do you really like cryptography?

More cryptography courses:

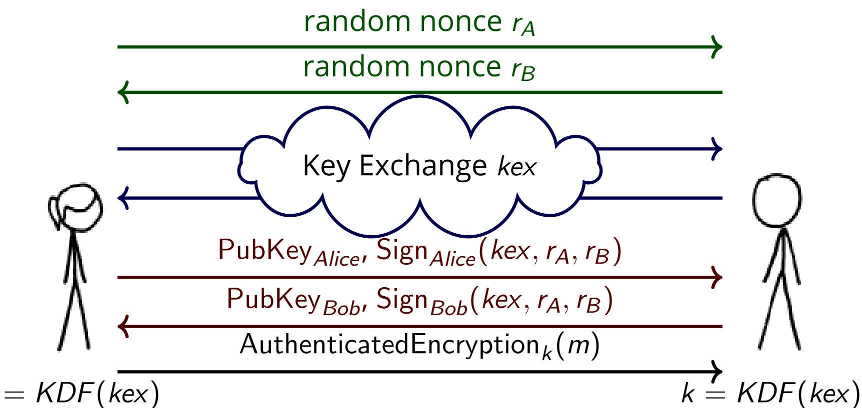
- CSE 207A: Graduate introduction to cryptography
- CSE 207B: Graduate applied cryptography
- CSE 208: Graduate advanced topics in cryptography
- CSE 206A: Lattice algorithms and applications
- Winter 2022 CSE 291-14: Number field sieve

Security courses:

- CSE 127: Undergraduate introduction to security
- CSE 227: Graduate security

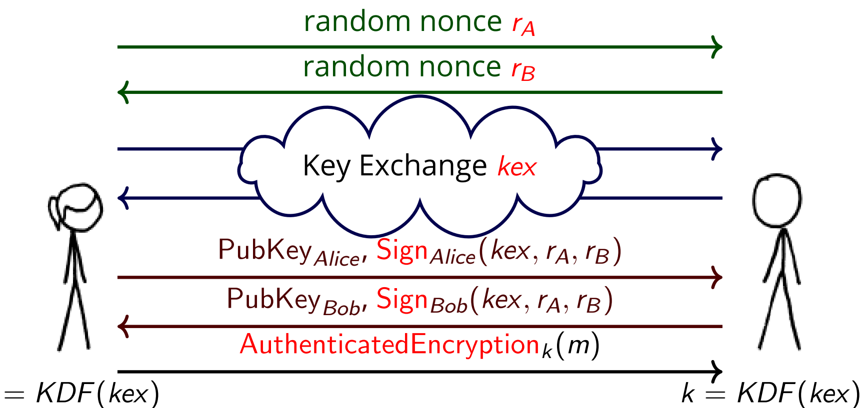
Interested in research or grad school and you did super well in CSE 107? Reach out about independent study possibilities or other ways to get involved.

Recall: Simplified cryptographic protocol diagram



1. Use symmetric encryption to encrypt messages.
2. Use a key exchange algorithm like Diffie-Hellman to agree on a shared symmetric key.
3. Use digital signatures to authenticate other party and guarantee integrity of key exchange.
4. Use random nonces to protect against replay attacks.

Random number generation in the cryptographic protocol



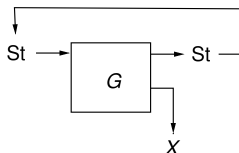
1. Use symmetric encryption to encrypt messages.
2. Use a key exchange algorithm like Diffie-Hellman to agree on a shared symmetric key.
3. Use digital signatures to authenticate other party and guarantee integrity of key exchange.
4. Use random nonces to protect against replay attacks.

“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.”

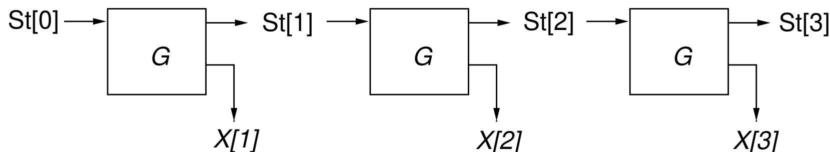
–John von Neumann

Stateful Generators

Initially, St is a random seed



Operation:



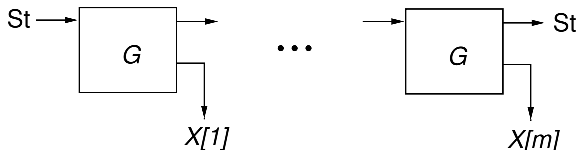
$X[1]X[2]X[3]\ldots$ is the output sequence and should be "pseudorandom".

Notation

$$(X[1]\ldots X[m], \mathrm{St}) \leftarrow G(\mathrm{St}, m)$$

means we

- Run G with starting state St for m steps
- Let $X[1]\ldots X[m]$ be the output blocks produced
- Let St be the updated state



Usage for Encryption

Alice maintains a state St_A and Bob maintains a state St_B.
Initially: $\mathrm{St}_A = \mathrm{St}_B$ is a random seed.

$\mathcal{E}(M[1] \ldots M[m])$	$\mathcal{D}(C[1] \ldots C[m])$
$(X[1] \ldots X[m], \mathrm{St}_A) \leftarrow G(\mathrm{St}_A, m)$	$(X[1] \ldots X[m], \mathrm{St}_B) \leftarrow G(\mathrm{St}_B, m)$
for $i = 1, \ldots, m$ do	for $i = 1, \ldots, m$ do
$\quad C[i] \leftarrow X[i] \oplus M[i]$	$\quad M[i] \leftarrow X[i] \oplus C[i]$

Note that the states must be synchronized!

Usage for Pseudorandom Bit Generation

G is initialized with a random seed and its outputs are then used for any purpose needing randomness, including:

- Keys
- IVs for block-cipher based encryption
- Nonces
- Simulations

Methods

- Linear Congruential Generators (LCGs)
- Linear Feedback Shift Registers (LFSRs)

These have

- Good statistical properties: #1's $\approx$ #0's; Chi-square; . . .
- But are predictable: Given some outputs can infer future ones

Predictability can be exploited to break encryption privacy via a chosen-message attack.

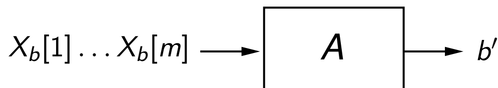
Cryptographic constructs

- (Alleged)-RC4
- SEAL (1.0, 2.0)

Theoretical Security Requirement

INDR : Indistinguishablity from random

- Pick a random seed St and let $(X_1[1] \ldots X_1[m], \mathrm{St}) \leftarrow G(\mathrm{St}, m)$
- Pick $X_0[1] \ldots X_0[m]$ at random
- Pick a challenge bit b at random



A is trying to compute b.

G is secure if no practical A has high advantage.

Formalization

Let G be a stateful generator with seed length s and output-block length n.

Game INDR_G

procedure Initialize
$\mathrm{St} \xleftarrow{\$} \{0,1\}^s; b \xleftarrow{\$} \{0,1\}$

procedure Next(m)
$(X_1[1]\ldots X_1[m], \mathrm{St}) \leftarrow G(\mathrm{St}, m)$
$X_0[1]\ldots X_0[m] \xleftarrow{\$} \{0,1\}^{nm}$
return $X_b[1]\ldots X_b[m]$

procedure Finalize(b')
return $(b = b')$

The indr advantage of adversary A is

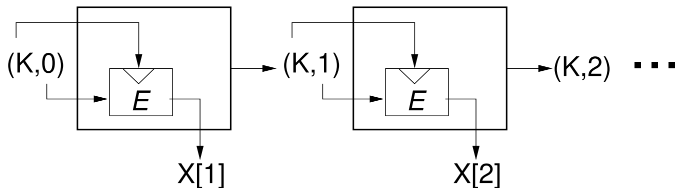
$$\mathrm{Adv}_G^{\mathrm{indr}}(A) = 2\Pr\left[\mathrm{INDR}_G^A \Rightarrow \mathsf{true}\right] - 1$$

Stream Ciphers / PRGs from Block Ciphers

Let $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a block cipher, and define:

$\mathrm{algorithm}\ G(\mathrm{St})$
$(K, i) \leftarrow \mathrm{St}$

- State has the form (K, i) where K is a key for E and i is an n-bit integer ($0 \leq i < 2^n$).
- Initial state is $(K, 0)$ where $K \xleftarrow{\$} \{0,1\}^k$.



Security of the CTR mode PRG

Fact: If E is a secure PRF, then G is an INDR secure PRG.

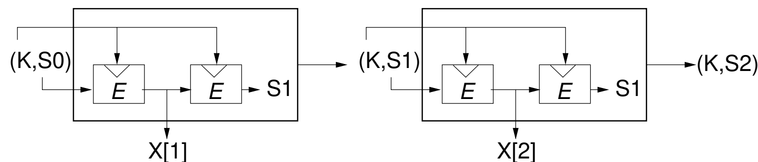
Similarly, other modes of operation of block ciphers also give rise to PRGs.

ANSI X9.17 PRG

Let $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$ be a block cipher and define

$\text{algorithm } G(\text{St})$
$(K, S) \leftarrow \text{St}$ // Parse St as (K, S)
$X \leftarrow E_K(S); S \leftarrow E_K(X); \text{return}(X, (K, S))$

- state has the form (K, S) where K is a key for E and $S \in \{0,1\}^n$
- Initial state has both K and S chosen at random.

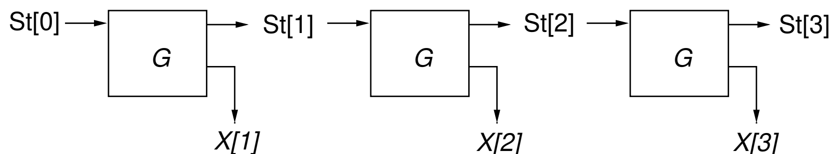


The standard uses $E =$ DES in 2-key EDE mode.
Analysis: [DHL02]

Government standards for RNGs

	FIPS 140-2	NIST SP800-90A	ISO 18031
Block cipher designs			
ANSI X9.31	Disallowed 2016		
CTR DRBG	✓	✓	✓
OFB DRBG			✓
Hash function designs			
ANSI X9.62	Disallowed 2016		
Hash DRBG	✓	✓	✓
HMAC DRBG	✓	✓	✓
Number theoretic designs			
Dual EC DRBG		Disallowed 2015	✓ (?)
Micali Schnorr DRBG			✓

Forward Security

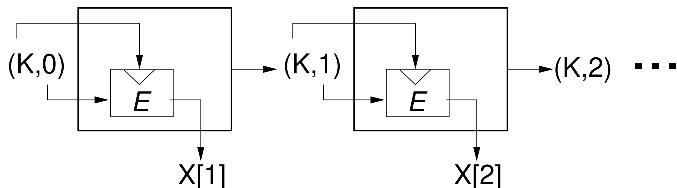


Suppose adversary obtains St_2. Then

- It can compute $X[3]X[4]\ldots$
- But can it compute $X[1]X[2]$?

Forward security requires that the answer to the latter question be "NO".
Important in the face of exposure due to malware and system compromise.

Forward Security Failures



If adversary gets $(K, 2)$ then it can compute

$$X[1] = E_K(0)\,;\ X[2] = E_K(1).$$

Similar failures for ANSI X9.17.

Generating randomness in a real implementation

Sources of “true” randomness:

A hardware/physical process that extracts randomness from the environment. Examples:

- Oscillating circuits
- Quantum phenomena
- Thermal noise
- Fine-grained instruction timing

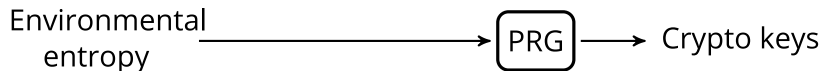
Generating randomness in a real implementation

Sources of “true” randomness:

A hardware/physical process that extracts randomness from the environment. Examples:

- Oscillating circuits
- Quantum phenomena
- Thermal noise
- Fine-grained instruction timing

These still have biases, and may be slow.

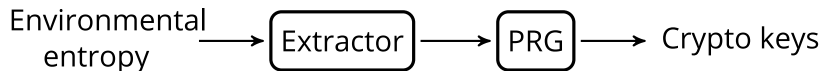


Generating randomness in a real implementation

Sources of "true" randomness:
A hardware/physical process that extracts randomness from the environment. Examples:

- Oscillating circuits
- Quantum phenomena
- Thermal noise
- Fine-grained instruction timing

These still have biases, and may be slow.



A PRG requires uniformly random inputs to be secure, so we need to use something like a hash function to obtain uniform inputs.

Practical Considerations for RNGs

- **Problem:** Inputs might not be random.

Practical Considerations for RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

Practical Considerations for RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

- **Problem:** Testing for randomness is theoretically impossible.

Practical Considerations for RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

- **Problem:** Testing for randomness is theoretically impossible.
  Solution: … do as well as you can?

Practical Considerations for RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

- **Problem:** Testing for randomness is theoretically impossible.
  Solution: … do as well as you can?

- **Problem:** Inputs might be controlled by attacker.

Practical Considerations for RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

- **Problem:** Testing for randomness is theoretically impossible.
  Solution: … do as well as you can?

- **Problem:** Inputs might be controlled by attacker.
  Solution: Seed from a variety of sources and hope attacker doesn't control everything.

Practical Considerations for RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

- **Problem:** Testing for randomness is theoretically impossible.
  Solution: … do as well as you can?

- **Problem:** Inputs might be controlled by attacker.
  Solution: Seed from a variety of sources and hope attacker doesn't control everything.

- **Problem:** User might request output before seeding.

Practical Considerations for RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness, use extractor functions, seed cryptographic PRG.

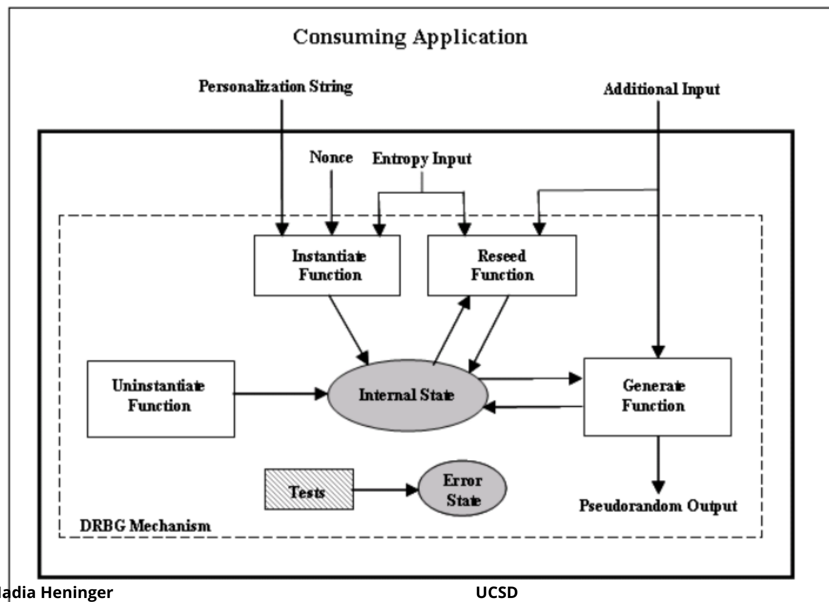
- **Problem:** Testing for randomness is theoretically impossible.
  Solution: … do as well as you can?

- **Problem:** Inputs might be controlled by attacker.
  Solution: Seed from a variety of sources and hope attacker doesn't control everything.

- **Problem:** User might request output before seeding.
  Possible solutions:
  1. Don't provide output.
  2. Provide output.
  3. Raise an error flag.

NIST SP800-90A

"Random Number Generation using Deterministic Random Bit Generators"



RNG security properties and threat model

Pseudorandomness: Output cryptographically indistinguishable from random.

Entropy Input: RNG seeded using enough entropy so adversary can't predict seed.

Prediction Resistance: Adversary who compromises state at time t can't distinguish output $t+1$ from random.

Backtracking Resistance: Adversary who compromises state at time t can't distinguish output $t-1$ from random.

Real-world threats to RNGs outside of theoretical model

- State-level adversaries interfering with the design and standardization process.
- Unclear or too-permissive algorithm specifications.
- Implementers misunderstanding algorithm specification.
- Implementation bugs.

Cryptographic failures with bad randomness

- Repeating IVs/keys with stream cipher encryption
  - XOR of ciphertext reveals information about plaintext

Cryptographic failures with bad randomness

- Repeating IVs/keys with stream cipher encryption
  - XOR of ciphertext reveals information about plaintext
- Repeating nonces with AES-GCM
  - Allows key recovery from ciphertext

Cryptographic failures with bad randomness

- Repeating IVs/keys with stream cipher encryption
  - XOR of ciphertext reveals information about plaintext
- Repeating nonces with AES-GCM
  - Allows key recovery from ciphertext
- Repeated public keys

Cryptographic failures with bad randomness

- Repeating IVs/keys with stream cipher encryption
  - XOR of ciphertext reveals information about plaintext
- Repeating nonces with AES-GCM
  - Allows key recovery from ciphertext
- Repeated public keys
- RSA keys with common factors
  - Allows secret key recovery from public keys

Cryptographic failures with bad randomness

- Repeating IVs/keys with stream cipher encryption
  - XOR of ciphertext reveals information about plaintext
- Repeating nonces with AES-GCM
  - Allows key recovery from ciphertext
- Repeated public keys
- RSA keys with common factors
  - Allows secret key recovery from public keys
- Repeated (EC)DSA signature nonces
  - Allows secret key recovery from signatures and public key

A biased historical tour of random number generation failures

Netscape SSL RNG Vulnerability [Goldberg Wagner 1996]

Underlying cause: Seeding PRNG with insufficient entropy.

```
global variable seed;

RNG_CreateContext()
    (seconds, microseconds) = time of day; /* Time elapsed since 1970 */
    pid = process ID;  ppid = parent process ID;
    a = mklcpr(microseconds);
    b = mklcpr(pid + seconds + (ppid << 12));
     seed = MD5(a, b);

mklcpr(x) /* not cryptographically significant; shown for completeness */
    return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

 RNG_GenerateRandomBytes()
     x = MD5(seed);
     seed = seed + 1;
     return x;

 global variable challenge, secret_key;

 create_key()
     RNG_CreateContext();
     ...
     challenge = RNG_GenerateRandomBytes();
     secret_key = RNG_GenerateRandomBytes();
```

The Debian OpenSSL Disaster

Luciano Bello, 2008

When Private Keys are Public: Results from the 2008 Debian OpenSSL Vulnerability Yilek, Rescorla, Shacham, Enright, Savage. (2009)

Underlying cause: Failure to seed PRNG.

OpenSSL PRNG

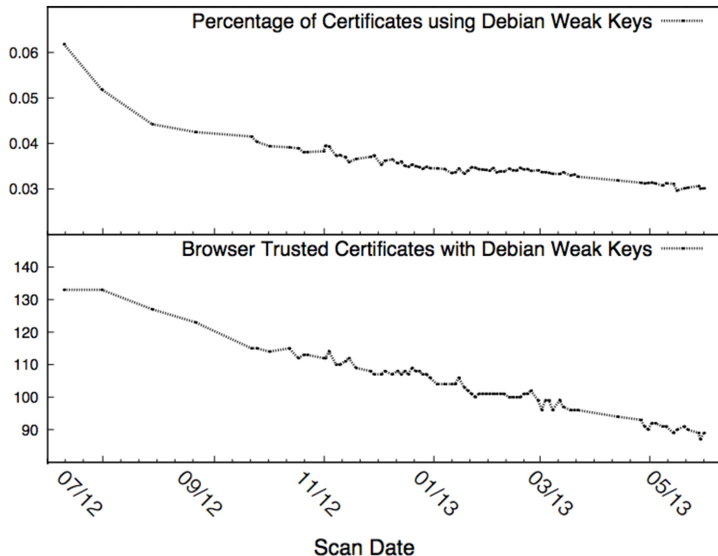
- Seed: `/dev/urandom`, pid, time()
- Update: time() (in seconds)
- Mixing function: SHA-1
- Output: SHA-1 hash of state.

```
/* state[st_idx], ..., state[(st_idx + num - 1) % STATE_SIZE]
 * are what we will use now, but other threads may use them
 * as well */

md_count[1] += (num / MD_DIGEST_LENGTH) + (num % MD_DIGEST_LENGTH > 0);

if (!do_not_lock) CRYPTO_w_unlock(CRYPTO_LOCK_RAND);

EVP_MD_CTX_init(&m);
for (i=0; i<num; i+=MD_DIGEST_LENGTH)
        {
        j=(num-i);
        j=(j > MD_DIGEST_LENGTH)?MD_DIGEST_LENGTH:j;

        MD_Init(&m);
        MD_Update(&m,local_md,MD_DIGEST_LENGTH);
        k=(st_idx+j)-STATE_SIZE;
        if (k > 0)
                {
                MD_Update(&m,&(state[st_idx]),j-k);
                MD_Update(&m,&(state[0]),k);
                }
        else
                MD_Update(&m,&(state[st_idx]),j);

        MD_Update(&m,buf,j);
        MD_Update(&m,(unsigned char *)&(md_c[0]),sizeof(md_c));
        MD_Final(&m,local_md);
        md_c[1]++;

        buf=(const char *)buf + j;

        for (k=0; k<j; k++)
                {
                /* Parallel threads may interfere with this,
                 * but always each byte of the new state is
                 * the XOR of some previous value of its
                 * and local_md (itermediate values may be lost).
```

```
List:       openssl-dev
Subject:    Random number generator, uninitialised data and valgrind.
From:       Kurt Roeckx <kurt () roeckx ! be>
Date:       2006-05-01 19:14:00

Hi,

When debbuging applications that make use of openssl using
valgrind, it can show alot of warnings about doing a conditional
jump based on an unitialised value.  Those unitialised values are
generated in the random number generator.  It's adding an
unintialiased buffer to the pool.

The code in question that has the problem are the following 2
pieces of code in crypto/rand/md_rand.c:

247:
                MD_Update(&m,buf,j);

467:
#ifndef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif

...

What I currently see as best option is to actually comment out
those 2 lines of code.  But I have no idea what effect this
really has on the RNG.  The only effect I see is that the pool
might receive less entropy.  But on the other hand, I'm not even
sure how much entropy some unitialised data has.

What do you people think about removing those 2 lines of code?


Kurt
```

Debian OpenSSL weak keys, 2006–2008

RNG output dependent on pid and machine architecture.



[Durumeric Wustrow Halderman 2013]

Widespread random number generation vulnerabilities

Mining your Ps and Qs: Widespread Weak Keys in Network Devices Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman *Usenix Security 2012*

Factoring RSA keys from certified smart cards: Coppersmith in the wild. Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren. *Asiacrypt 2013.*

Weak keys remain widespread in network devices Marcella Hastings, Joshua Fried, and Nadia Heninger *IMC 2016*

RSA and GCDs

Public Key
$(N = pq, e)$

Private Key
$(p, q, d \equiv e^{-1} \bmod (p-1)(q-1))$

RSA and GCDs

Public Key
$(N = pq, e)$

Private Key
$(p, q, d \equiv e^{-1} \bmod (p-1)(q-1))$

If two RSA moduli share a common factor,

$$N_1 = pq_1 \qquad N_2 = pq_2$$

RSA and GCDs

Public Key
$(N = pq, e)$

Private Key
$(p, q, d \equiv e^{-1} \bmod (p-1)(q-1))$

If two RSA moduli share a common factor,

$$N_1 = pq_1 \qquad N_2 = pq_2$$

$$\gcd(N_1, N_2) = p$$

You can factor both keys with GCD algorithm.

Time to factor
829-bit RSA modulus:
2700 core-years
[Boudot et al. 2020]

Time to calculate GCD
for 1024-bit RSA moduli:
15μs

Should we expect to find prime collisions in the wild?

Experiment: Compute GCD of each pair of M RSA moduli randomly chosen from P primes.

What *should* happen? **Nothing.**

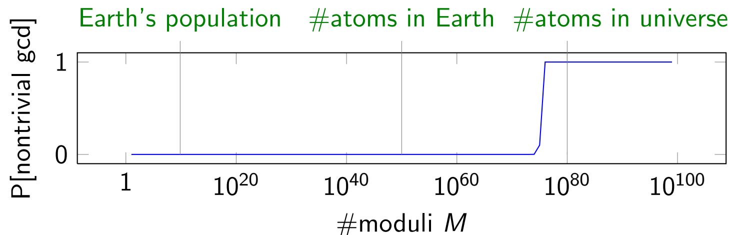
Should we expect to find prime collisions in the wild?

Experiment: Compute GCD of each pair of M RSA moduli randomly chosen from P primes.

What *should* happen? **Nothing.**

Prime Number Theorem:
$\sim 10^{150}$ 512-bit primes

Birthday bound:
$\Pr[\text{nontrivial gcd}] \approx 1 - e^{-2M^2/P}$



What happened when we GCDed RSA keys in 2012?

Computed private keys for

- 64,081 HTTPS servers (0.50%).
- 2,459 SSH servers (0.03%).
- 2 PGP users (and a few hundred invalid keys).

What happened when we GCDed RSA keys in 2012?

Computed private keys for

- 64,081 HTTPS servers (0.50%).
- 2,459 SSH servers (0.03%).
- 2 PGP users (and a few hundred invalid keys).

What has happened since?

- 103 Taiwanese citizen smart card keys [Bernstein, Chang, Cheng, Chou, Heninger, Lange, van Someren 2013]
- 90 export-grade HTTPS keys. [Albrecht, Papini, Paterson, Villanueva-Polanco 2015]
- 313,330 HTTPS, SSH, IMAPS, POP3S, SMTPS keys [Hastings Fried Heninger 2016]
- 3,337 Tor relay RSA keys. [Kadianakis, Roberts, Roberts, Winter 2017]

What happens if you look for repeated (EC)DSA nonces?

Compute private keys for

- Thousands of SSH servers
- Hundreds of HTTPs hosts
- Thousands of Bitcoin keys
- ...

Widespread RNG failures on low resource devices

GCDing RSA TLS and SSH keys accidentally uncovered multiple independent implementation problems.

Vast majority of weak keys generated by low resource devices.

- Juniper network security devices
- Cisco routers
- Fortigate firewalls
- Intel server management cards
- . . .

Identified devices from $>$ 50 manufacturers

Very different behavior for different devices. Different companies, implementations, underlying software, distributions of prime factors.

One cause: Cascading PRNG failures

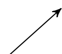
RSA keys

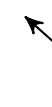
OpenSSL PRNG

time

pid

Linux PRNG





One cause: Cascading PRNG failures

RSA keys

OpenSSL PRNG

time

pid

Linux PRNG

Many devices automatically generate crypto keys on first boot.

One cause: Cascading PRNG failures

RSA keys

OpenSSL PRNG

time

pid

Linux PRNG

Many devices automatically generate crypto keys on first boot.

- Headless or embedded devices often lack these entropy sources.

One cause: Cascading PRNG failures

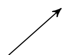
RSA keys

OpenSSL PRNG

time

pid

Linux PRNG



Many devices automatically generate crypto keys on first boot.

- The Linux PRNG had not yet been seeded when queried by OpenSSL $\Longrightarrow$ deterministic output. Patched since 2012.

- Headless or embedded devices often lack these entropy sources.

How did factorable keys arise in practice?

- Usability problems in random number generator interface.

```
 /* We'll use /dev/urandom by default, since /dev/random is too
much hassle.  If system developers aren't keeping seeds between
boots nor getting any entropy from somewhere it's their own fault.
*/
#define DROPBEAR_RANDOM_DEV "/dev/urandom"
```

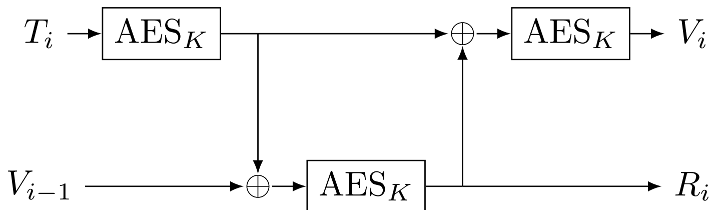
- A cascade of vulnerable software behaviors.
  - OpenSSL mixed current time in seconds into RNG state
  - This led to factorable and not merely repeated keys.

ANSI X9.31 and the DUHK attack

Practical state recovery attacks against legacy RNG implementations
Shaanan Cohney, Matthew D. Green, Nadia Heninger. CCS 2018.

The ANSI X9.31/X9.17 RNG

- Uses block cipher (AES or 3DES) as a mixing function.
- On each iteration, mixes state V_{i-1} with timestamp T_i.
- Produces output block R_i and new state V_i.



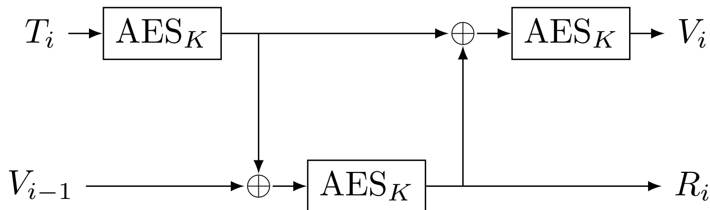
ANSI X9.31 RNG History

- 1985: DES-based RNG standardized in ANSI X9.17
- 1992: Adopted as a FIPS standard
- 1994: Included on list of approved RNGs in FIPS 140-1
- 1998: Variant using 3DES standardized in ANSI X9.31
- 1998: Kelsey et al.: state recovery if key known
- 2004: ANSI X9.31 RNG included in FIPS 186-2
- 2005: AES-based variant published by NIST and included on FIPS 140-2 approved RNGs
- 2011: FIPS deprecates ANSI X9.31 design
- 2016: ANSI X9.31 RNG removed from FIPS 140-2

X9.31 state recovery from a known key

[Kelsey, Schneier, Wagner, Hall 1998]

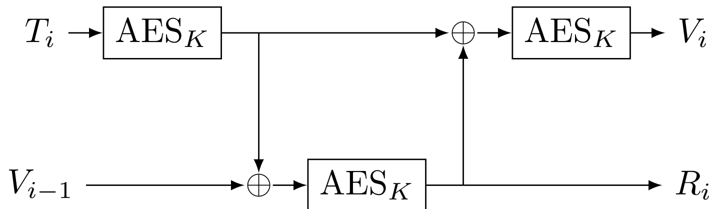
If key K used with block cipher is known, can recover state V_{i-1} from output R_i by brute forcing timestamp.



X9.31 state recovery from a known key

[Kelsey, Schneier, Wagner, Hall 1998]

If key K used with block cipher is known, can recover state V_{i-1} from output R_i by brute forcing timestamp.



Design flaws:

- Block cipher is invertible, so if key is known can run both forwards and backwards.
- Low-resolution timestamps alone do not provide much entropy.
- Fixed symmetric key used for many outputs increases attack surface.

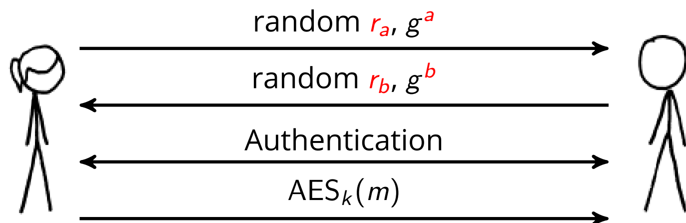
NIST ANSI X9.31 RNG standardization failure

"For AES 128-bit key, let *K be a 128 bit key."

"This *K is reserved only for the generation of pseudo random numbers."

- Standard did *not* specify key should not be hard-coded.
- Fortigate FortiOS v4 hard-coded NIST test vector key (oops)

Passive X9.31 state recovery in the IPsec protocol



- Raw RNG outputs: IKE nonce, cookie.

1. Use nonce, cookie to recover RNG state.
2. Once state recovered, increment forward to recover Diffie-Hellman secret
3. Verify DH exponent against public value on the wire.

Summary of security proofs

Proofs from [Woodage and Shumow 2019]

	Security Proof	Should you use?
Block cipher designs		
ANSI X9.31	✗	✗
CTR DRBG	✗	Maybe.
OFB DRBG	???	???
Hash function designs		
ANSI X9.62	✗	✗
Hash DRBG	✓	✓
HMAC DRBG	✓*	✓*
Number theoretic designs		
Dual EC DRBG	✓*	✗
Micali Schnorr DRBG	✗	✗

Discussion

- It is surprising that RNG security models are still in flux.
- It is surprising that NIST SP800-90A designs did not receive formal security analysis until a couple of years ago.
- Cryptographic standards should probably go through developer usability testing.
- NIST is reforming the FIPS certification process.

Thanks for a fun quarter!

Good luck on your finals!