

# CSE 207B: Applied Cryptography

**Nadia Heninger**

UCSD

Fall 2025 Lecture 15

# Announcements

1. HW 7 due in 1 week.

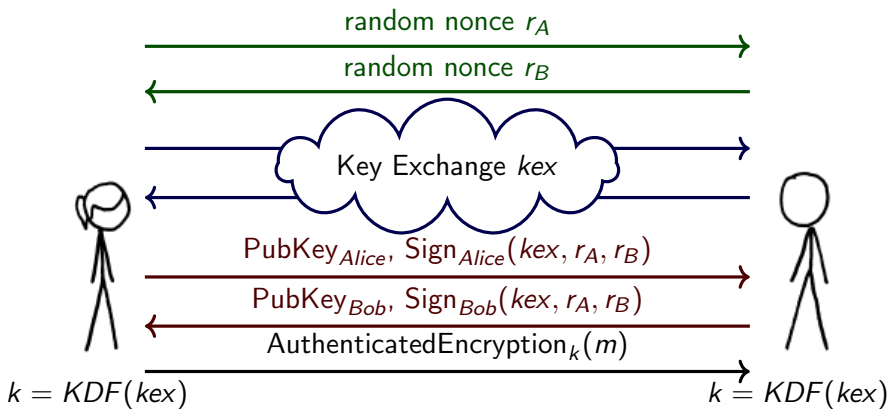
**Last time:**

- Authenticated key exchange and TLS

**This time:**

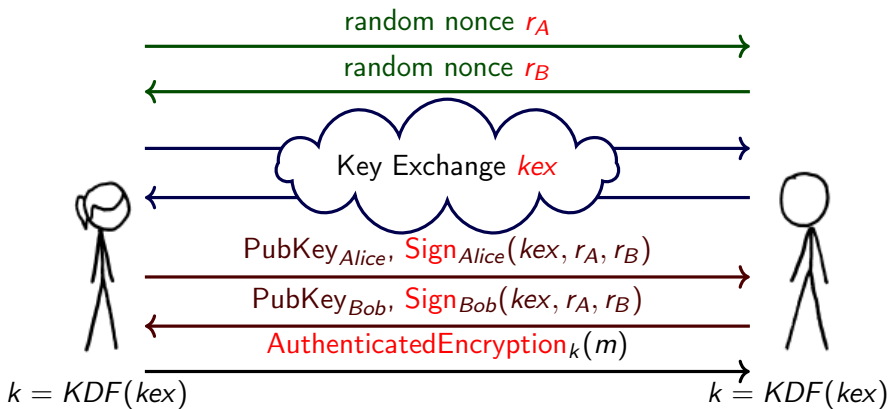
- Random number generation

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

# Generating randomness for cryptography

## 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

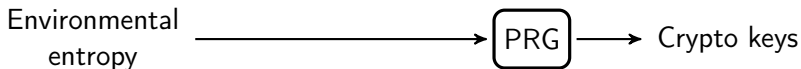
# Generating randomness for cryptography

## 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.



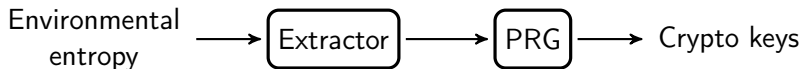
# Generating randomness for cryptography

## 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”

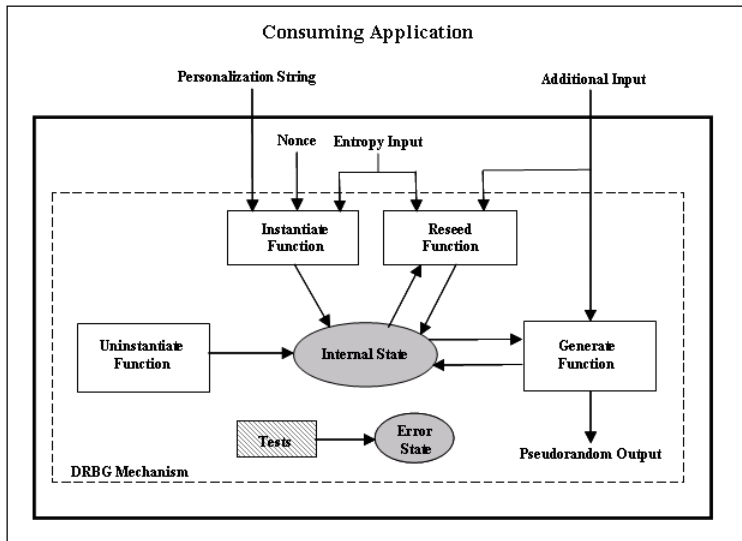


Figure 1: DRBG Functional Model

# 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.

# Government standards for RNGs

	FIPS 140-2	NIST SP800-90A	ISO 18031
Block cipher designs			
<b>ANSI X9.31</b>	Disallowed 2016		
<b>CTR DRBG</b>	✓	✓	✓
<b>OFB DRBG</b>			✓
Hash function designs			
<b>ANSI X9.62</b>	Disallowed 2016		
<b>Hash DRBG</b>	✓	✓	✓
<b>HMAC DRBG</b>	✓	✓	✓
Number theoretic designs			
<b>Dual EC DRBG</b>		Disallowed 2015	Removed 2014
<b>Micali Schnorr DRBG</b>			✓

# 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

# 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 debugging 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 uninitialised 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:
#ifdef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif

...
```

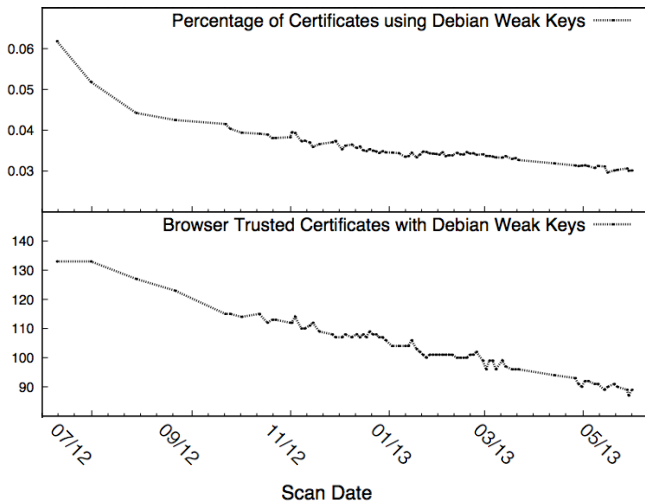
What I currently see as best option is to actually comment out those 2 lines of code. But I have no idea what effect this really has on the RNG. The only effect I see is that the pool might receive less entropy. But on the other hand, I'm not even sure how much entropy some unitialised data has.

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*

# 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.

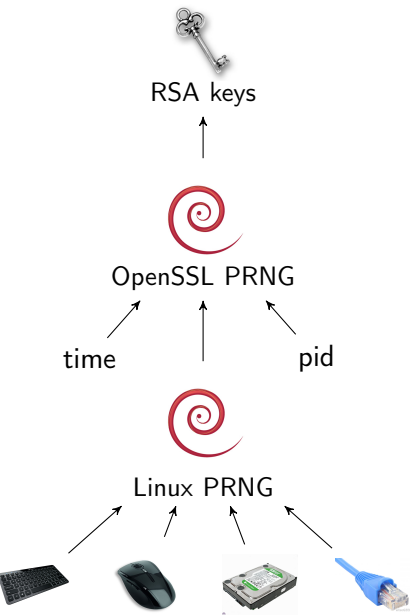


- Juniper network security devices
- Cisco routers
- Fortigate firewalls
- Intel server management cards
- ...

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



# 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  $\implies$  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.

## Generating vulnerable RSA keys in software

- Insufficiently random seeds for pseudorandom number generator  $\implies$  we should see repeated keys.

```
prng.seed()  
p = prng.random_prime()  
q = prng.random_prime()  
N = p*q
```

- We do:
  - $> 60\%$  of hosts share keys
  - At least  $0.3\%$  due to bad randomness.
- Repeated keys may be a sign that implementation is vulnerable to a targeted attack.

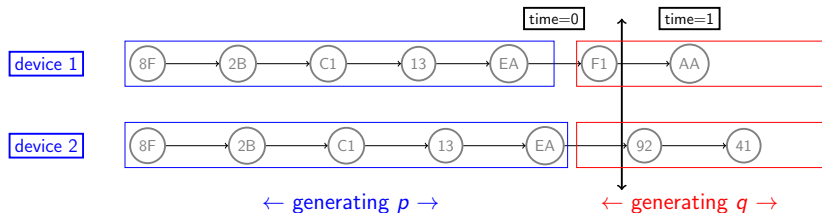
But why do we see factorable keys?

# Generating factorable RSA keys in software

```
prng.seed()  
p = prng.random_prime()  
prng.add_randomness()  
q = prng.random_prime()  
N = p*q
```

OpenSSL adds time in seconds

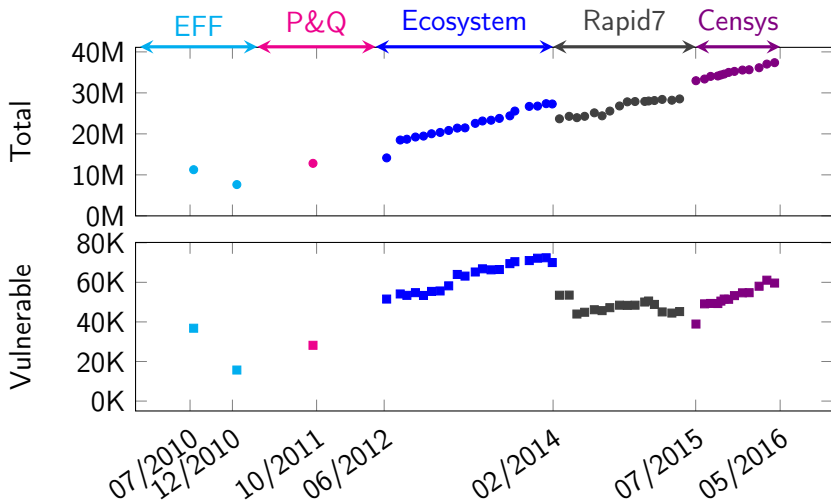
Insufficient randomness can lead to factorable keys.



Experimentally verified OpenSSL generates factorable keys in this situation.

# Follow-up study: Six years of factoring keys

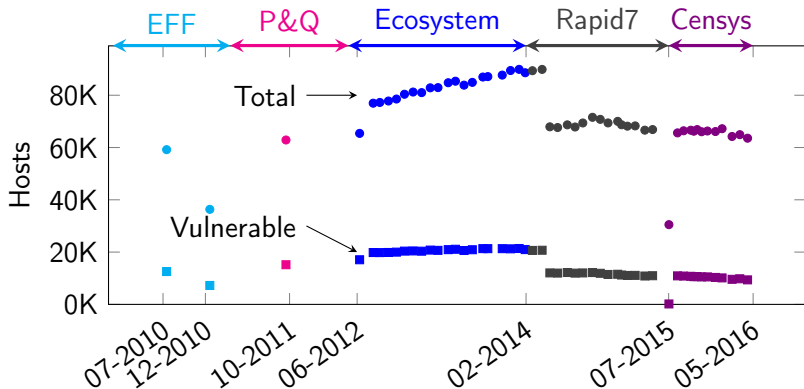
- 51 million distinct HTTPS RSA moduli : 0.43% vulnerable
- 65 million distinct HTTPS certificates : 2.2% vulnerable
- 1.5 billion HTTPS host records : 0.19% vulnerable



# Juniper

SRX Series Service Gateways (SRX100, SRX110, SRX210, SRX220, SRX240, SRX550, SRX650), LN1000 Mobile Secure Router

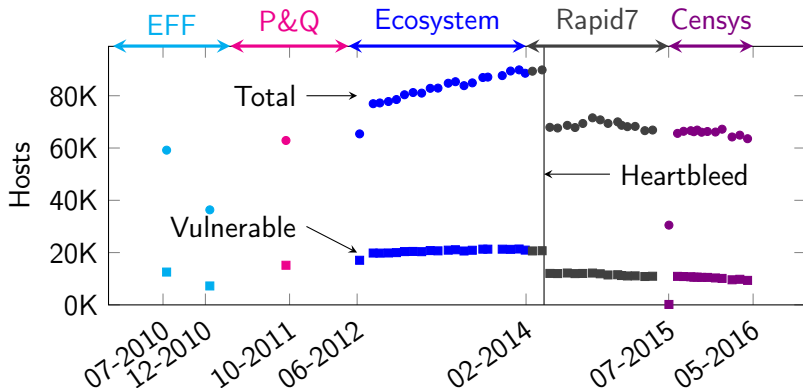
- Public security bulletin in April 2012, out-of-cycle security notice in July 2012
- Majority of factored keys in 2012 were Juniper hosts
- Weird behavior in April 2014



# Juniper

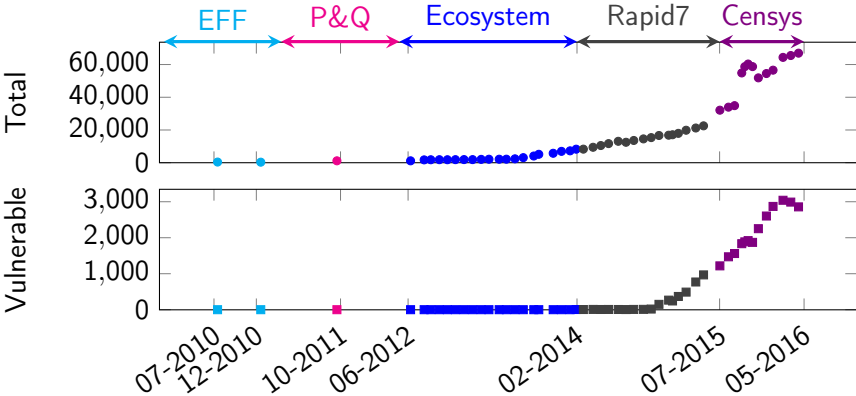
SRX Series Service Gateways (SRX100, SRX110, SRX210, SRX220, SRX240, SRX550, SRX650), LN1000 Mobile Secure Router

- 30,000 Juniper-fingerprinted hosts (9000 vulnerable) came offline after Heartbleed
- IPs do not reappear in later scans: TLS disabled, scans blocked, devices offline?



# Huawei

- Introduced vulnerability in 2014
- Security advisory published Aug 2016



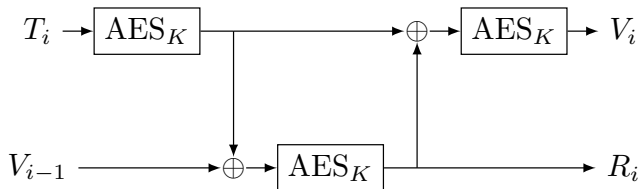
# ANSI X9.31 and the DUHK attack



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

## The ANSI X9.31 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  $V_i$ .



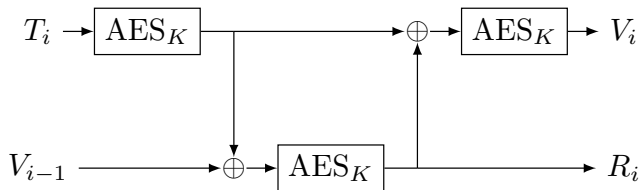
## ANSI X9.31 RNG History

- 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]

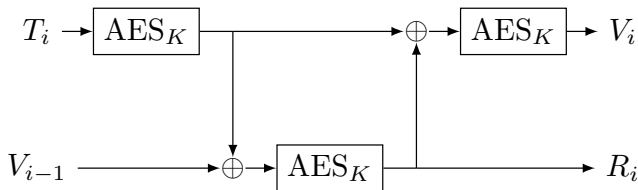
If key  $K$  used with block cipher is known, can recover state  $V_{i-1}$  from output  $R_i$  by brute forcing timestamp.



## 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.

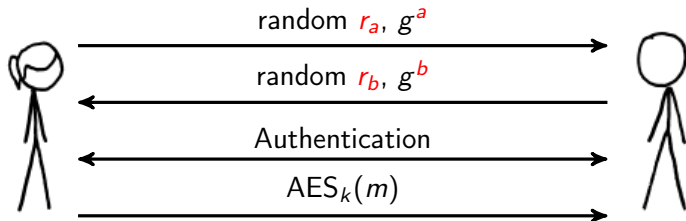
# NIST ANSI X9.31 RNG standardization failure

"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

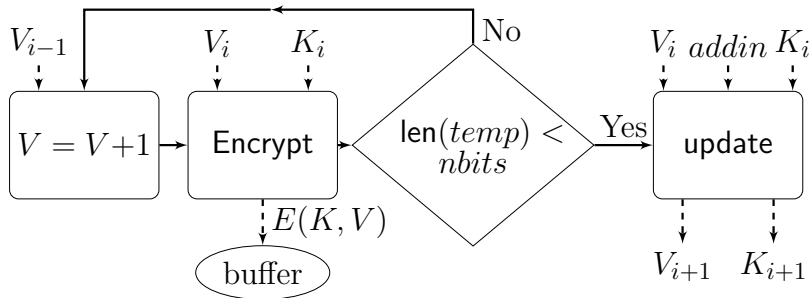


- Raw RNG outputs: IKE nonce, cookie.
1. Use nonce, cookie to recover RNG state.
  2. Once state recovered, increment forward to recover Diffie-Hellman secret
  3. Verify DH exponent against public value on the wire.

CTR DRBG



# CTR DRBG



- Output: AES encryption of incrementing counter.
- State key, counter values refreshed periodically
- Optional additional entropy can be mixed in on update

# Cryptanalysis of CTR DRBG

[Bernstein 2017], [Woodage and Shumow 2019]

Design flaws:

- Block cipher is invertible, so if attacker learns key can move both backwards and forwards.
- In some situations, key is reused to generate many outputs, increasing attack surface (e.g. against side-channel attacks).
- Standard does not require additional entropy to be added.

# Cryptanalysis of CTR DRBG

[Bernstein 2017], [Woodage and Shumow 2019]

Design flaws:

- Block cipher is invertible, so if attacker learns key can move both backwards and forwards.
- In some situations, key is reused to generate many outputs, increasing attack surface (e.g. against side-channel attacks).
- Standard does not require additional entropy to be added.

Theoretical attack when large amounts of data is buffered:

1. Attacker compromises state key  $K_t$  using a side-channel attack at time  $t$
2. Attacker decrypts output  $r_t$  to compute the state  $V_t$
3. Attacker winds generator forward using update function
4. Attacker then predicts output

# Hash DRBG and HMAC DRBG

[Woodage and Shumow 2019]

## HMAC DRBG:

- Use HMAC as mixing function
- Lacks a proof of security.

## Hash DRBG:

- Uses a hash function as a mixing function.
- Has a proof of security.

## Dual EC DRBG (again)

*On the Practical Exploitability of Dual EC in TLS Implementations*

Checkoway, Fredrikson, Niederhagen, Everspaugh, Green, Lange, Ristenpart, Bernstein, Maskiewicz, Shacham. Usenix Security 2014.

# Dual EC DRBG

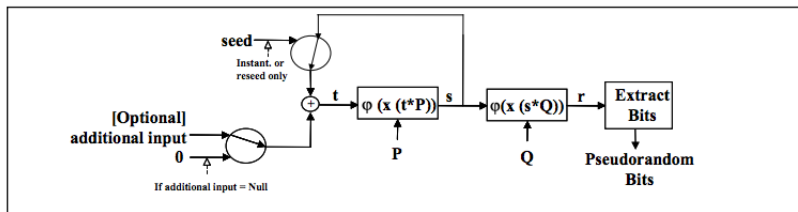


Figure 13: Dual\_EC\_DRBG

- Parameters: Pre-specified elliptic curve points  $P$  and  $Q$ .
- Seed: 32-byte integer  $s$
- State:  $x$ -coordinate  $x(sP)$ .
- Update: Set  $s = x(tP)$ , where  $t = s \oplus$  optional additional input.
- Output: 30 bytes of  $x(sQ)$ .

# Dual EC DRBG History

- Early 2000s: Created by the NSA and pushed towards standardization
- 2004: Published as part of ANSI X9.82 part 3 draft
- 2004: RSA makes Dual EC the default PRNG in BSAFE
- 2005: Standardized in NIST SP 800-90 draft
- 2007: Shumow and Ferguson demonstrate theoretical backdoor
- 2013: Snowden documents lead to renewed interest in Dual EC
- 2014: Practical attacks on TLS using Dual EC demonstrated
- 2015: NIST removes Dual EC from list of approved PRNGs

# Dual EC DRBG Trapdoor

[Shumow Ferguson 2007]

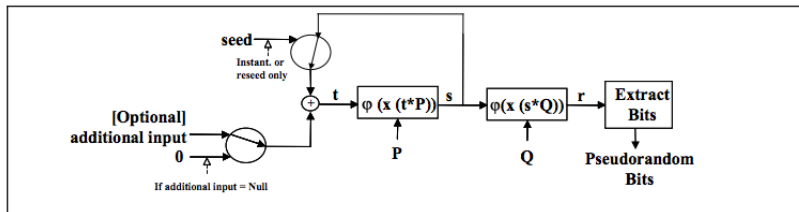
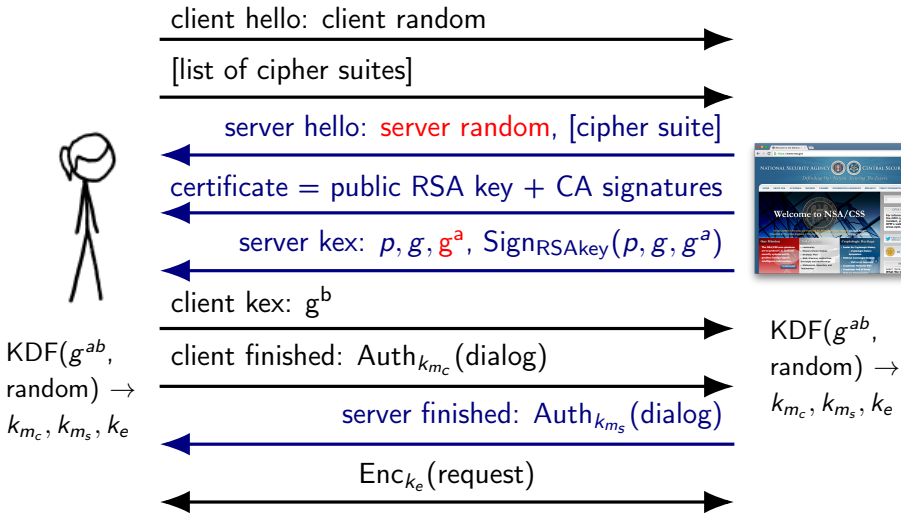


Figure 13: Dual\_EC\_DRBG

1. Attacker controls standard and constructs points with known relationship  $P = dQ$ .
2. Attacker gets 30 bytes of output. Attacker brute forces  $2^{16}$  bits to get  $2^{15}$  candidate points for  $sQ$ .
3. For each candidate  $sQ$  attacker computes  $dsQ = sP$ .
4. Matching value reveals correct state.
5. Attacker can now predict future outputs.

# Exploiting Dual EC trapdoor in TLS 1.2

- Server random is raw Dual EC output.
- Attacker mounts state recovery attack on server random, winds forward for DH secret.



# 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.
  - Linux kernel discussion is ongoing! <https://lore.kernel.org/lkml/Ym1MGx6+uigkGiZ0@zx2c4.com/>
- It is surprising that NIST SP800-90A designs did not receive formal security analysis until years after standardization.
- Cryptographic standards should probably go through developer usability testing.
- NIST is reforming the FIPS certification process.