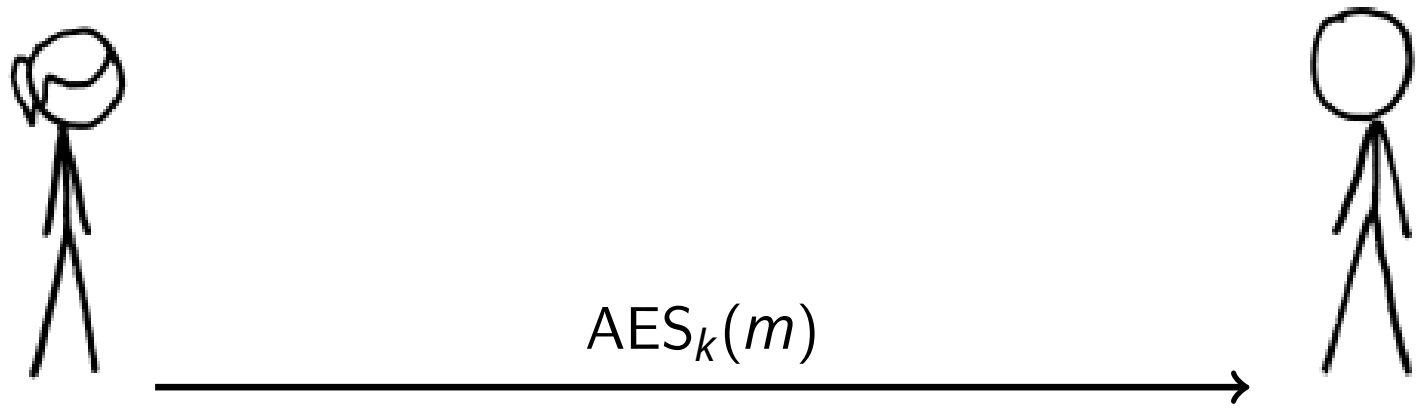
Random number generation failures from Netscape to DUHK

Nadia Heninger

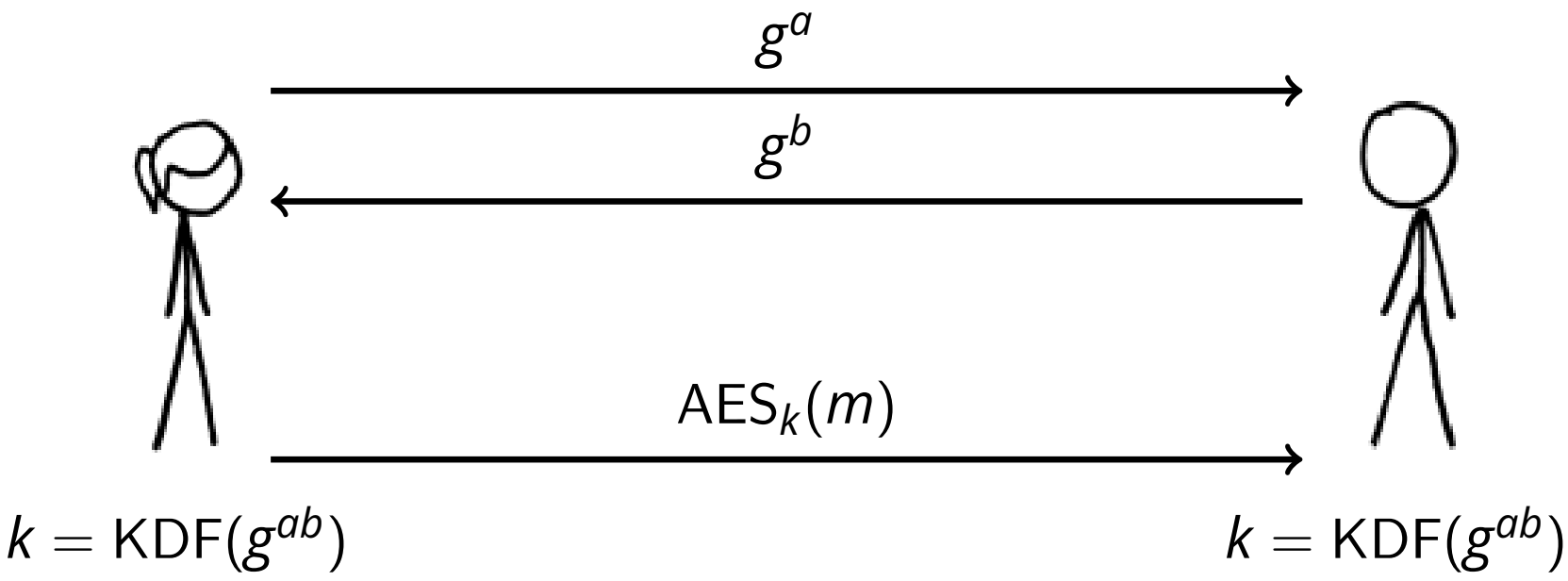
University of Pennsylvania

June 18, 2018

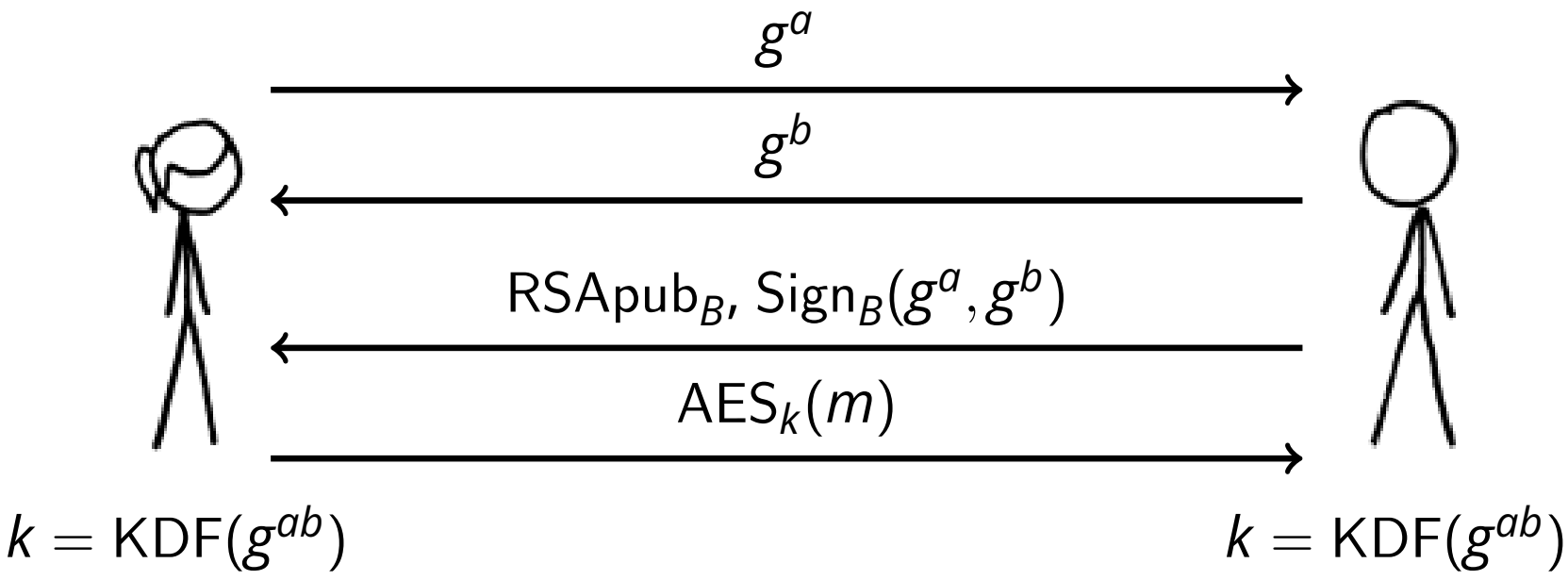
A cartoon cryptographic communication protocol



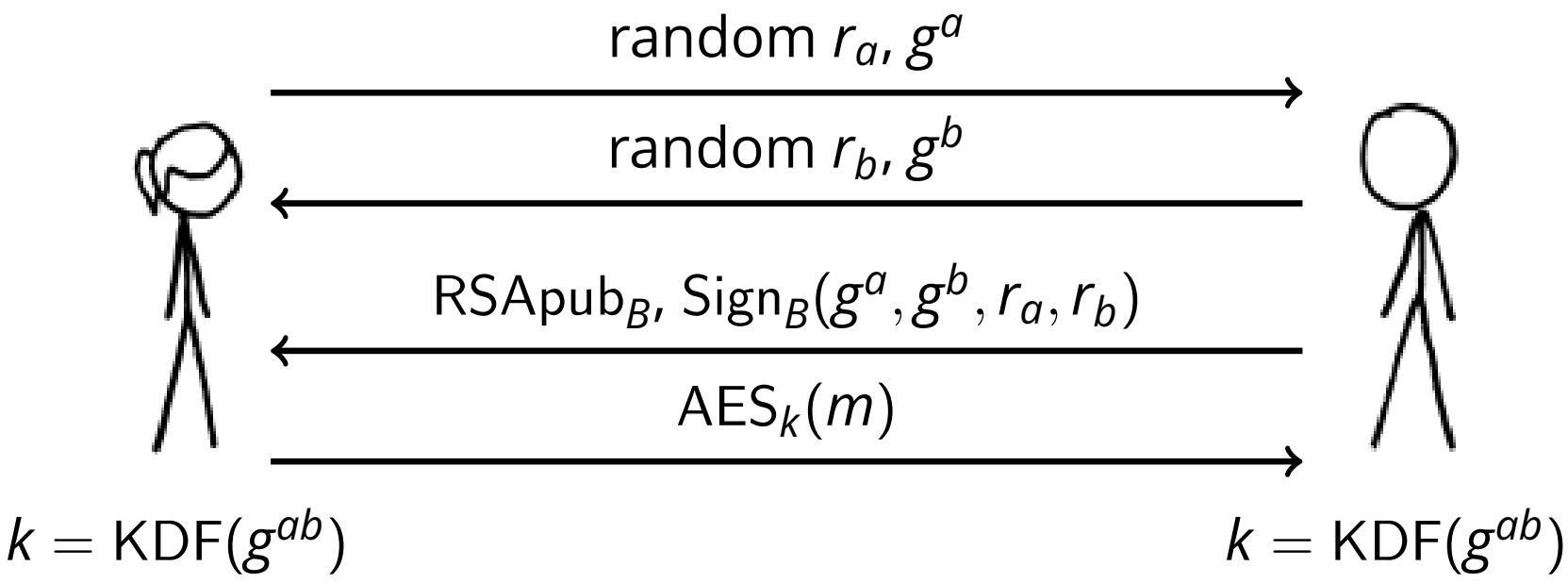
A cartoon cryptographic communication protocol



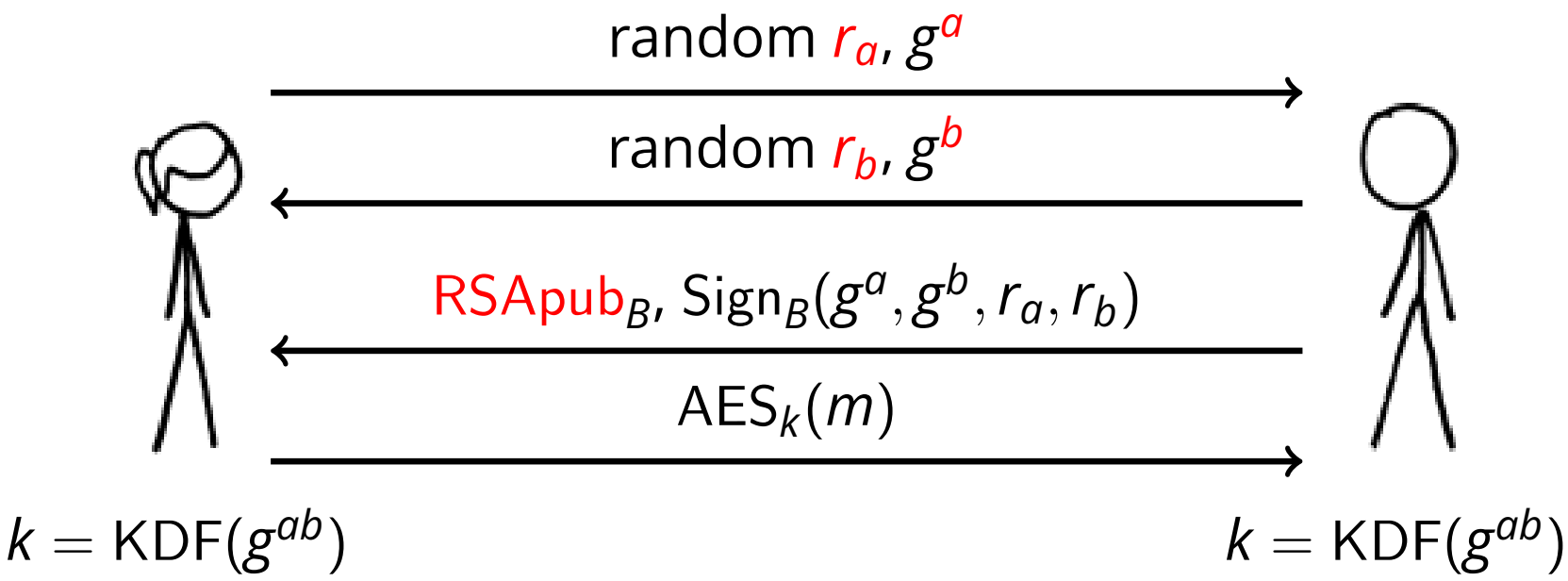
A cartoon cryptographic communication protocol



A cartoon cryptographic communication protocol



A cartoon cryptographic communication protocol



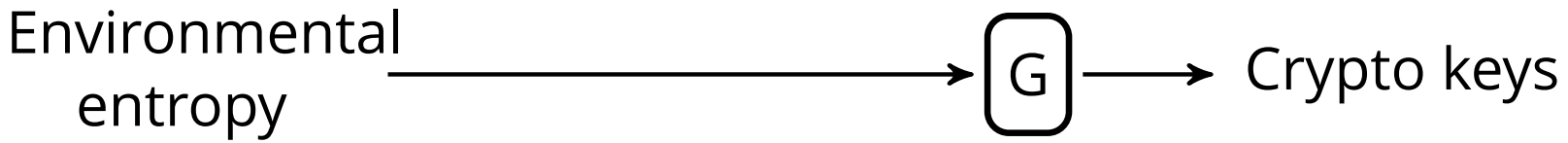
“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.”

–John von Neumann

Cryptographic pseudorandomness in theory

Definition

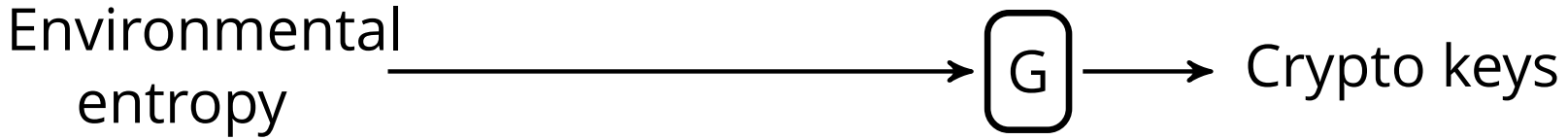
A *pseudorandom generator* is a polynomial-time deterministic function G mapping n-bit strings into $\ell(n)$-bit strings for $\ell(n) \geq n$ whose output distribution $G(U_n)$ is computationally indistinguishable from the uniform distribution $U_{\ell(n)}$.



Cryptographic pseudorandomness in theory

Definition

A *pseudorandom generator* is a polynomial-time deterministic function G mapping n-bit strings into $\ell(n)$-bit strings for $\ell(n) \geq n$ whose output distribution $G(U_n)$ is computationally indistinguishable from the uniform distribution $U_{\ell(n)}$.

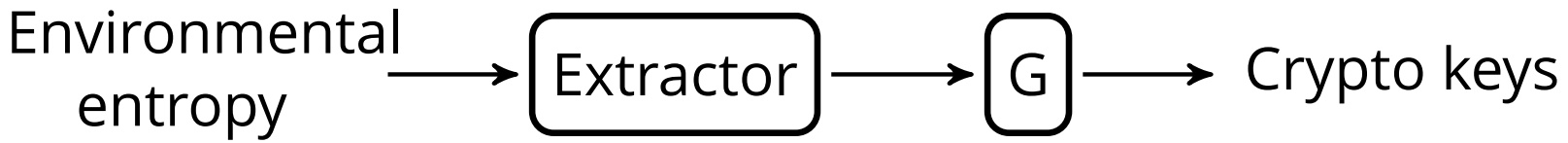


Problem: Environmental entropy not uniformly distributed.

Cryptographic pseudorandomness in theory

Definition

A *pseudorandom generator* is a polynomial-time deterministic function G mapping n-bit strings into $\ell(n)$-bit strings for $\ell(n) \geq n$ whose output distribution $G(U_n)$ is computationally indistinguishable from the uniform distribution $U_{\ell(n)}$.



NIST SP800-90A

"Random Number Generation using Deterministic Random Bit Generators"

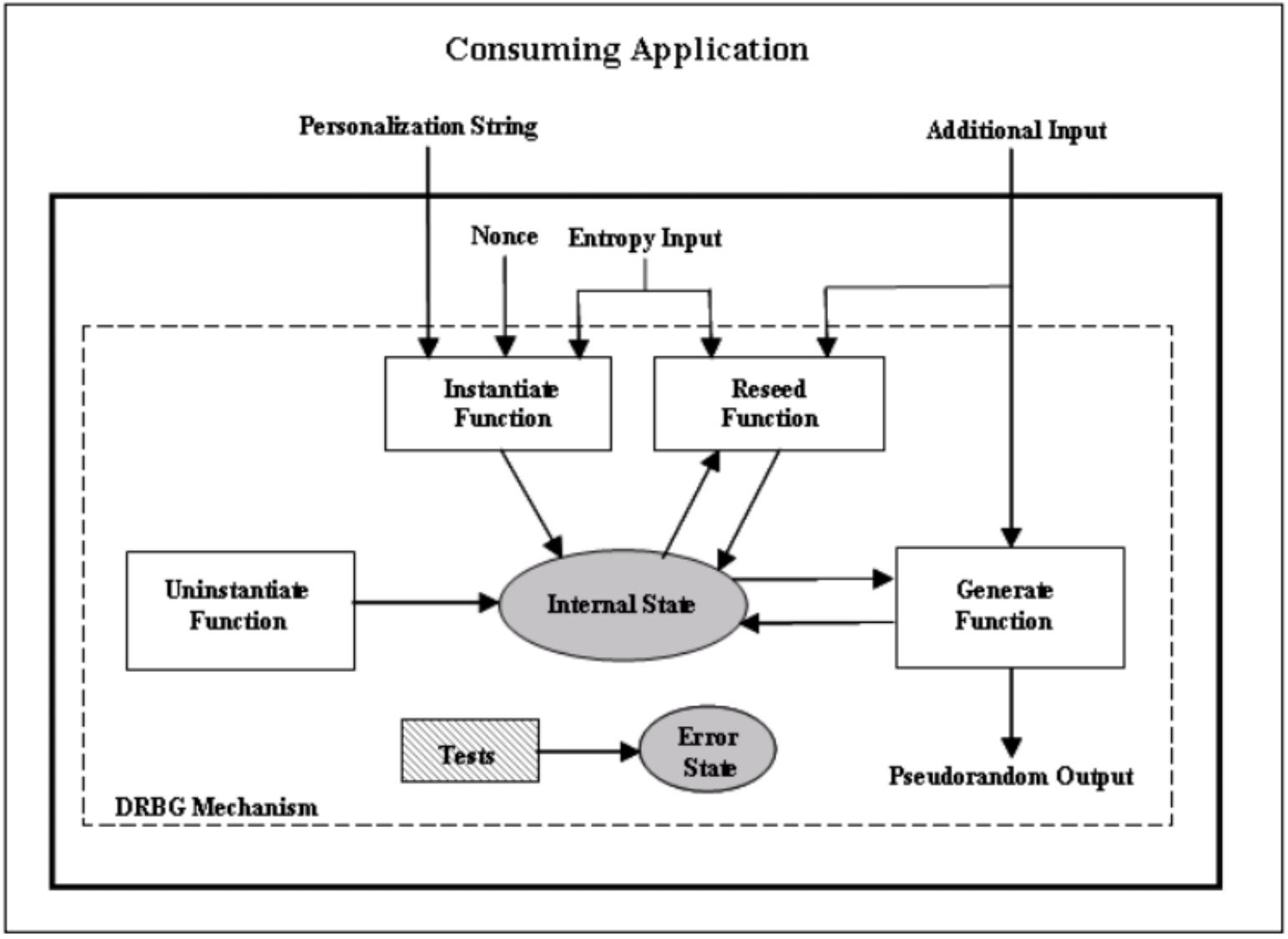


Figure 1: DRBG Functional Model

Practical Considerations with RNGs

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

Practical Considerations with RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness.

Practical Considerations with RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness.

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

Practical Considerations with RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness.

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

Practical Considerations with RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness.

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

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

Practical Considerations with RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness.

- **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 with RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness.

- **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:** How often do you reseed?

Practical Considerations with RNGs

- **Problem:** Inputs might not be random.
  Solution: Test for randomness.

- **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:** How often do you reseed?
  Possible solutions:
  1. On every new input.
  2. After k inputs accumulated in input pools.
  3. After ℓ blocks of outputs requested.

Practical considerations with RNGs

. . . that don’t make sense in theory.

- **Problem:** User might not seed PRNG.

Practical considerations with RNGs

. . . that don’t make sense in theory.

- **Problem:** User might not seed PRNG.
  Solution: Seed the PRNG.

Practical considerations with RNGs

. . . that don't make sense in theory.

- **Problem:** User might not seed PRNG.
  Solution: Seed the PRNG.
- **Problem:** User might request output before seeding RNG.

Practical considerations with RNGs

. . . that don't make sense in theory.

- **Problem:** User might not seed PRNG.
  Solution: Seed the PRNG.

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

Practical considerations with RNGs

. . . that don't make sense in theory.

- **Problem:** User might not seed PRNG.
  Solution: Seed the PRNG.

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

- **Problem:** RNG is seeded with low entropy inputs.

Practical considerations with RNGs

. . . that don't make sense in theory.

- **Problem:** User might not seed PRNG.
  Solution: Seed the PRNG.

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

- **Problem:** RNG is seeded with low entropy inputs.
  Solution: Seed with high entropy inputs.

Practical considerations with RNGs

. . . that don't make sense in theory.

- **Problem:** User might not seed PRNG.
  Solution: Seed the PRNG.

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

- **Problem:** RNG is seeded with low entropy inputs.
  Solution: Seed with high entropy inputs.

- **Problem:** User might use flawed or backdoored PRNG design.

Practical considerations with RNGs

. . . that don't make sense in theory.

- **Problem:** User might not seed PRNG.
  Solution: Seed the PRNG.

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

- **Problem:** RNG is seeded with low entropy inputs.
  Solution: Seed with high entropy inputs.

- **Problem:** User might use flawed or backdoored PRNG design.
  Solution: Don't use vulnerable designs.

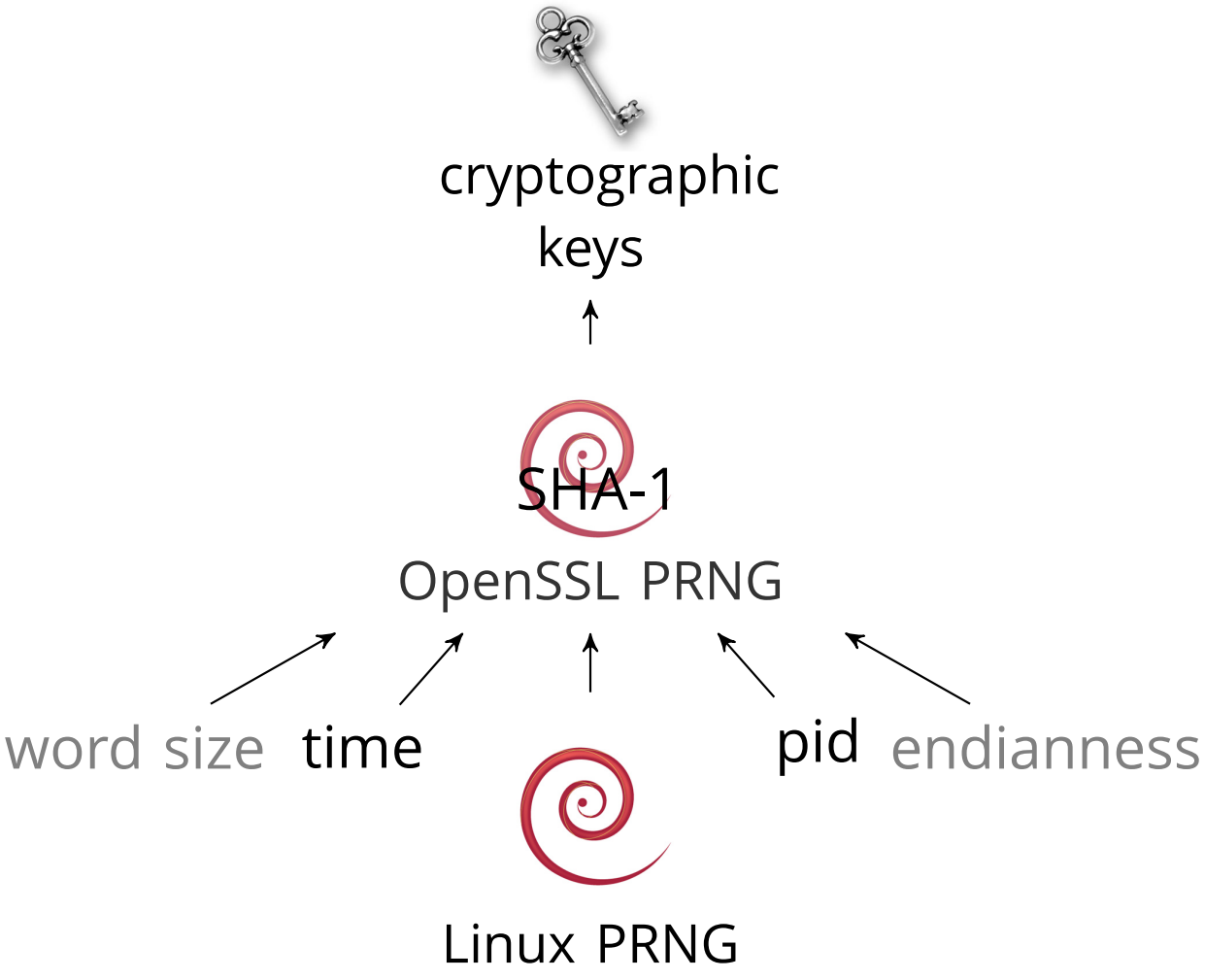
Disaster 1: Debian OpenSSL

Luciano Bello, 2008

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

Problem: User might not seed PRNG.
Solution: Seed the PRNG.

OpenSSL PRNG



```
/* 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
```

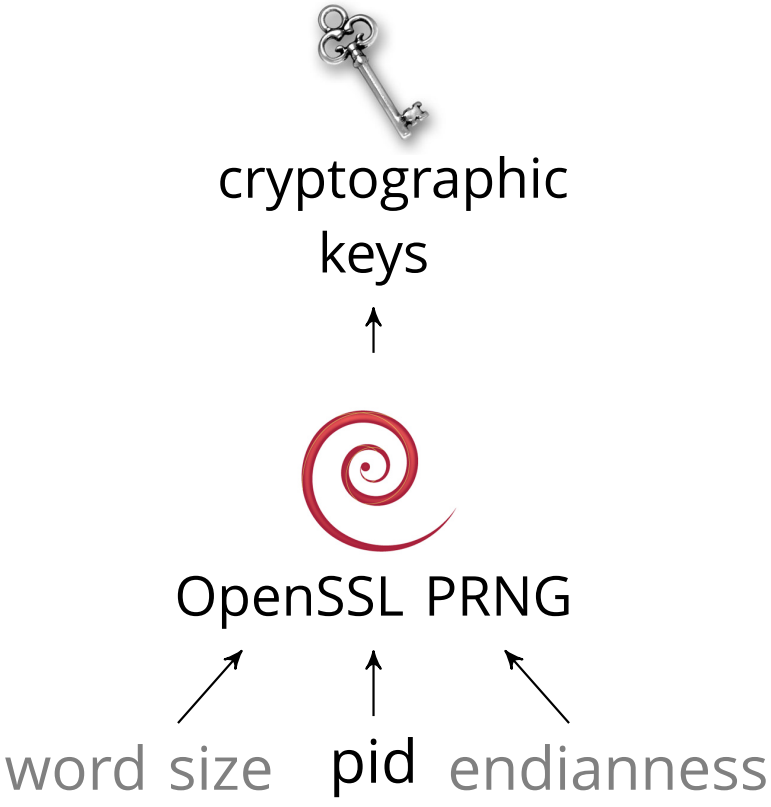
...

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



Estimated $>$ 1% of HTTPS hosts affected at disclosure time.

Disaster 2: Linux boot-time entropy hole

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

Public Keys Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter *Crypto 2012*

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

Problem: User might request output before seeding RNG.
Solution: Make sure RNG is seeded before providing output.

Linux OS RNG

`/dev/random`

"high-quality" pseudorandomness

blocks if insufficient entropy

`/dev/urandom`

pseudorandomness

never blocks

"As a general rule, `/dev/urandom` should be used for everything except long-lived GPG/SSL/SSH keys."—`man random`

RNG designs vs. real-world users

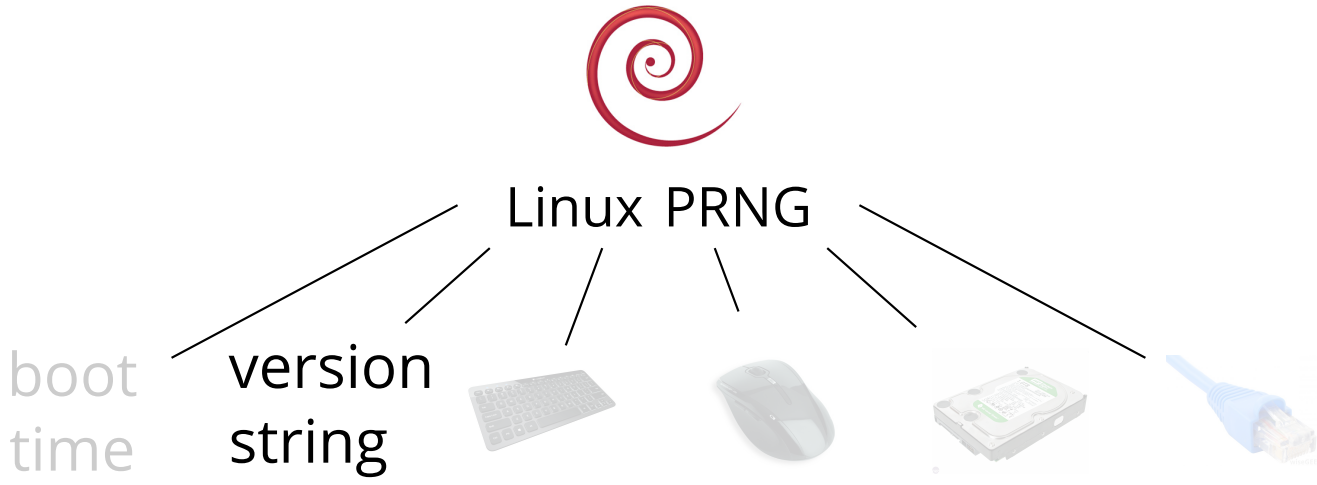
> *\/dev/urandom can indeed run out of entropy if it is called repeatedly.*
>
> – Random person on Bitcoin forum

> */dev/random is too severe. It's basically designed to be an information-theoretic random source, which means you could use its output as a one-time pad even if your adversary were time-travelling deities with countless universes full of quantum computers at their disposal.*
>
> – Random person on Hacker News

```
 /* 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"
```

Widespread RNG failures on low resource devices

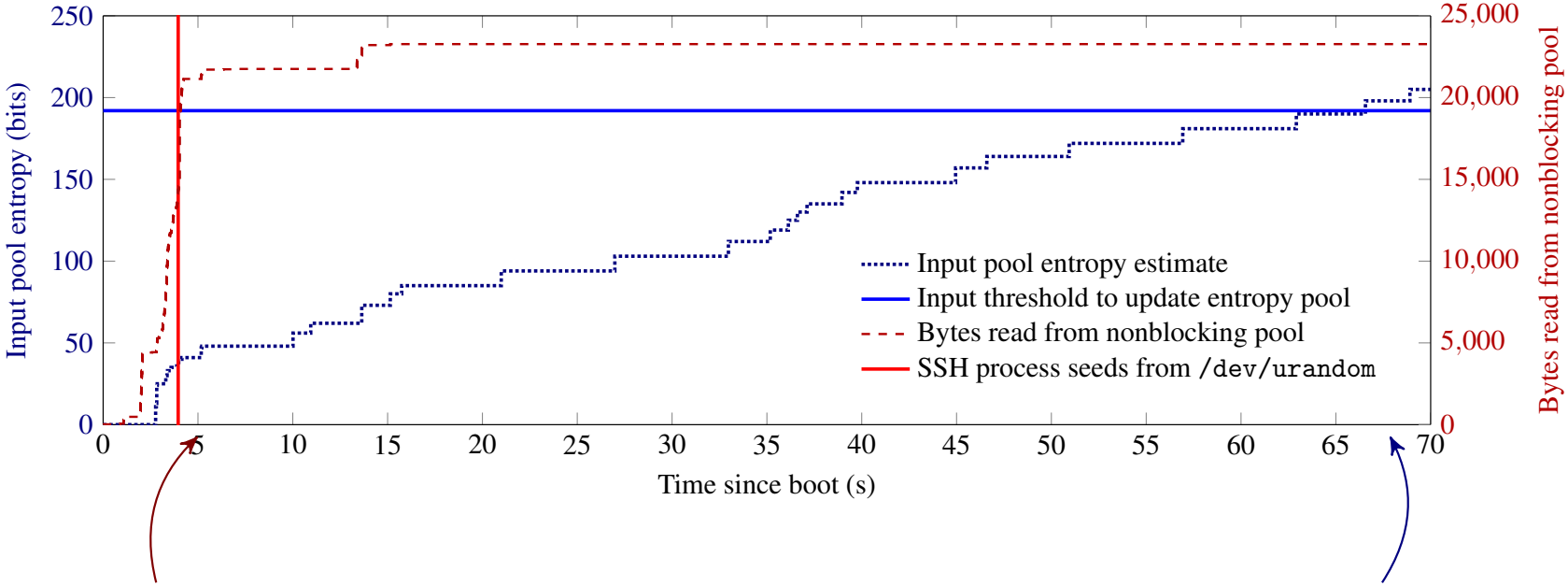
Problem # 1: Devices lack many default entropy inputs.



Linux boot-time entropy hole

Problem #2: PRNG waited to mix entropy inputs into output pool for fear of active attacks.

Ubuntu Server 10.04 on simulated low resource device



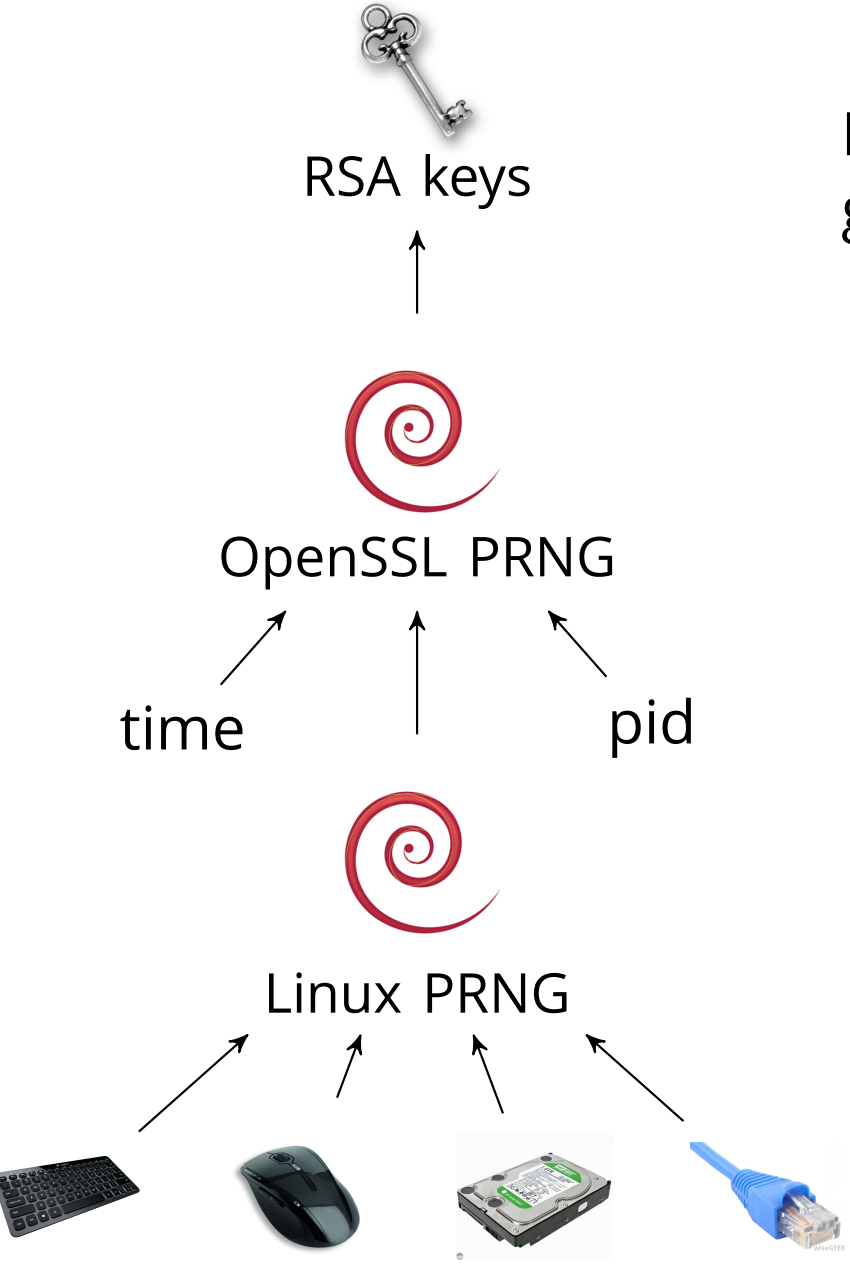
SSH process starts

entropy pool updated

Patched since July 2012.

Linux `getrandom()` (introduced in 2014) has correct interface: blocks if not seeded, always provides output if seeded.

Cascading OpenSSL and 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.
- Headless or embedded devices often lack these entropy sources.

Result: Widespread weak cryptographic keys.

In 2012, computed private keys for:

- 64,000 HTTPS servers (0.5%).
- 107,000 SSH servers (1%).
- 2 PGP users (and a few hundred invalid keys).

Result: Widespread weak cryptographic keys.

In 2012, computed private keys for:

- 64,000 HTTPS servers (0.5%).
- 107,000 SSH servers (1%).
- 2 PGP users (and a few hundred invalid keys).

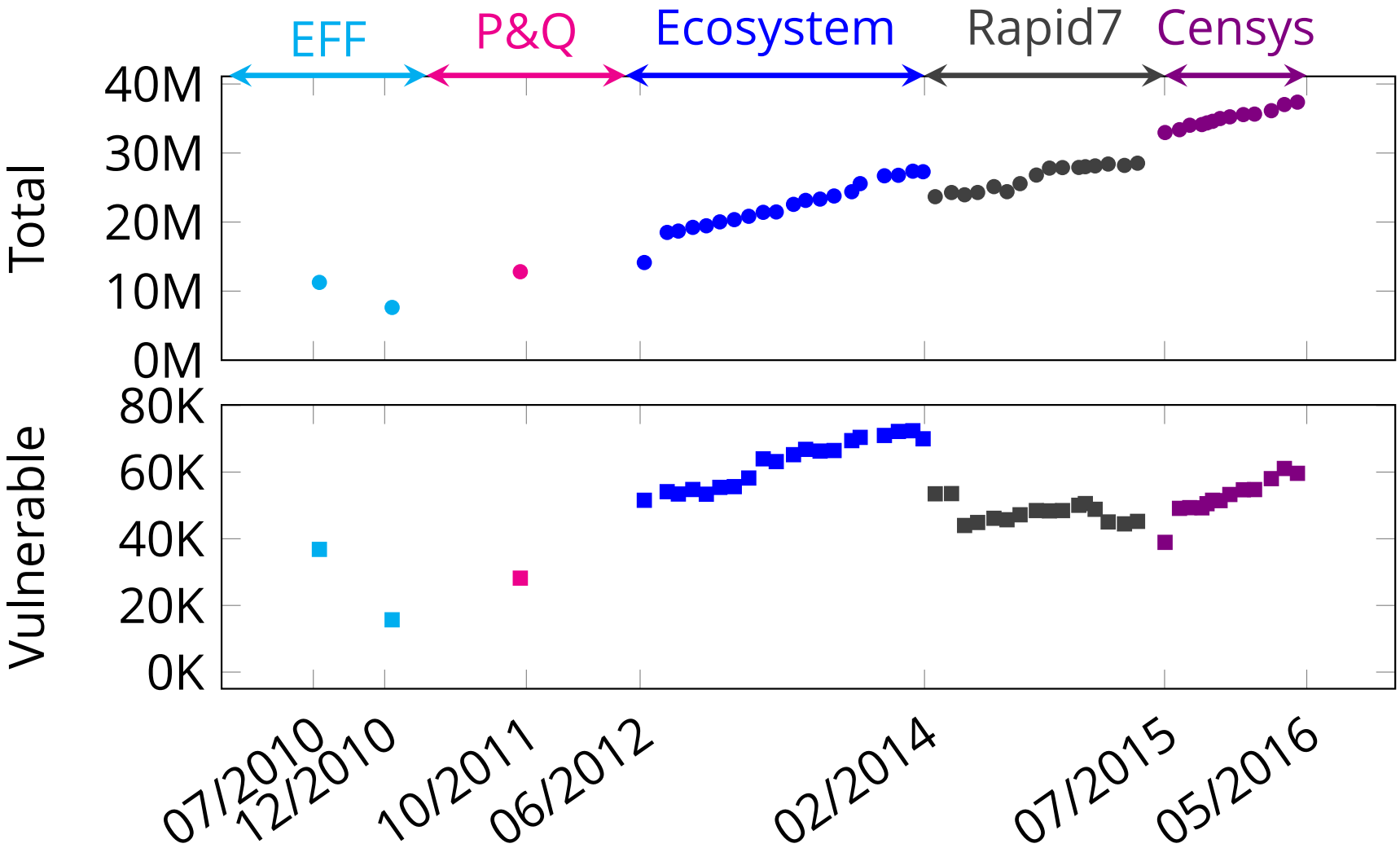
What has happened since?

- 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]

Follow-up study: Six years of weak keys

Question: Do vendors actually fix flaws after vulnerability disclosure?

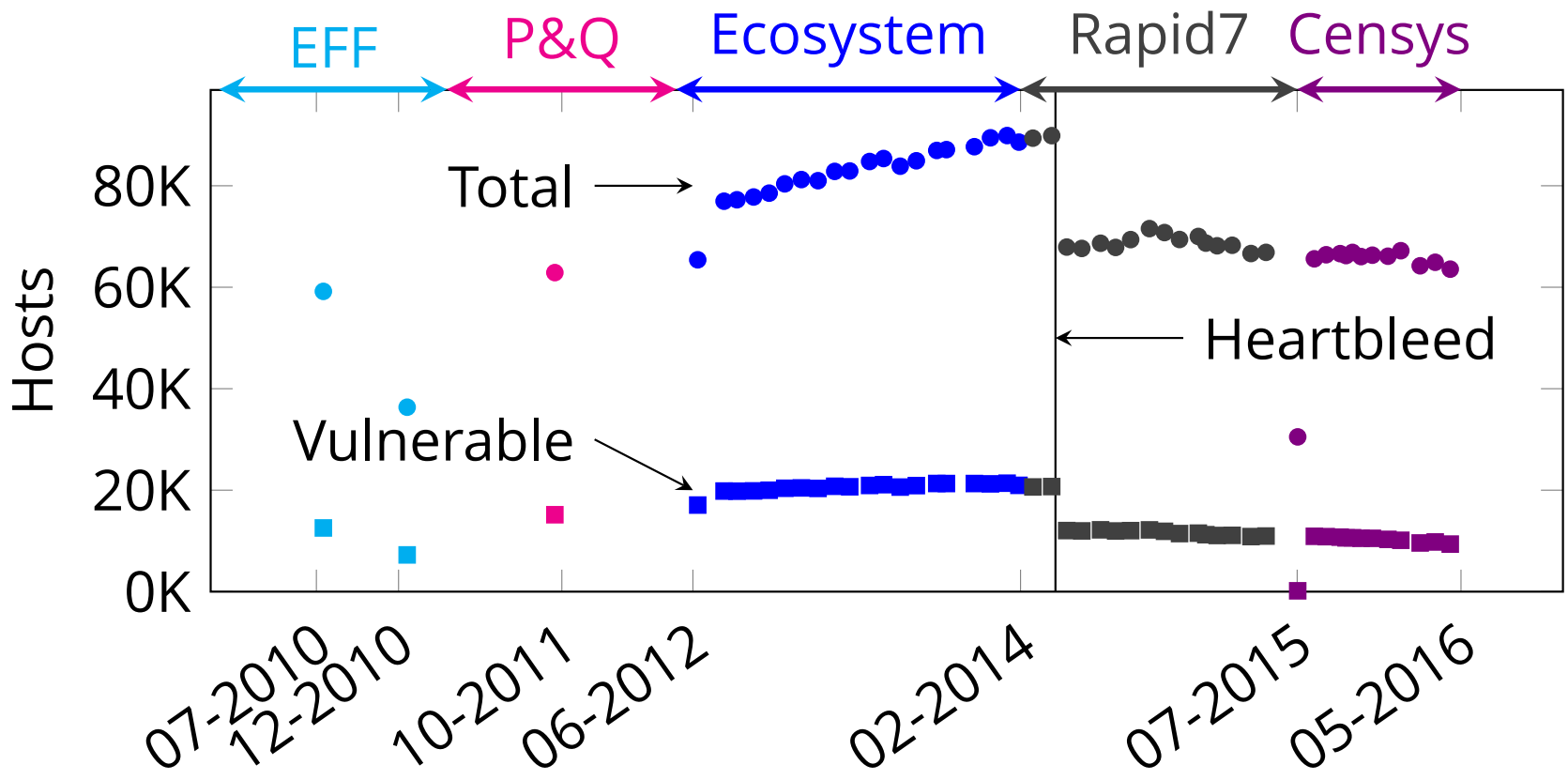
- 65 million distinct HTTPS certificates : 2.2% vulnerable
- 1.5 billion HTTPS host records : 0.19% vulnerable



Juniper

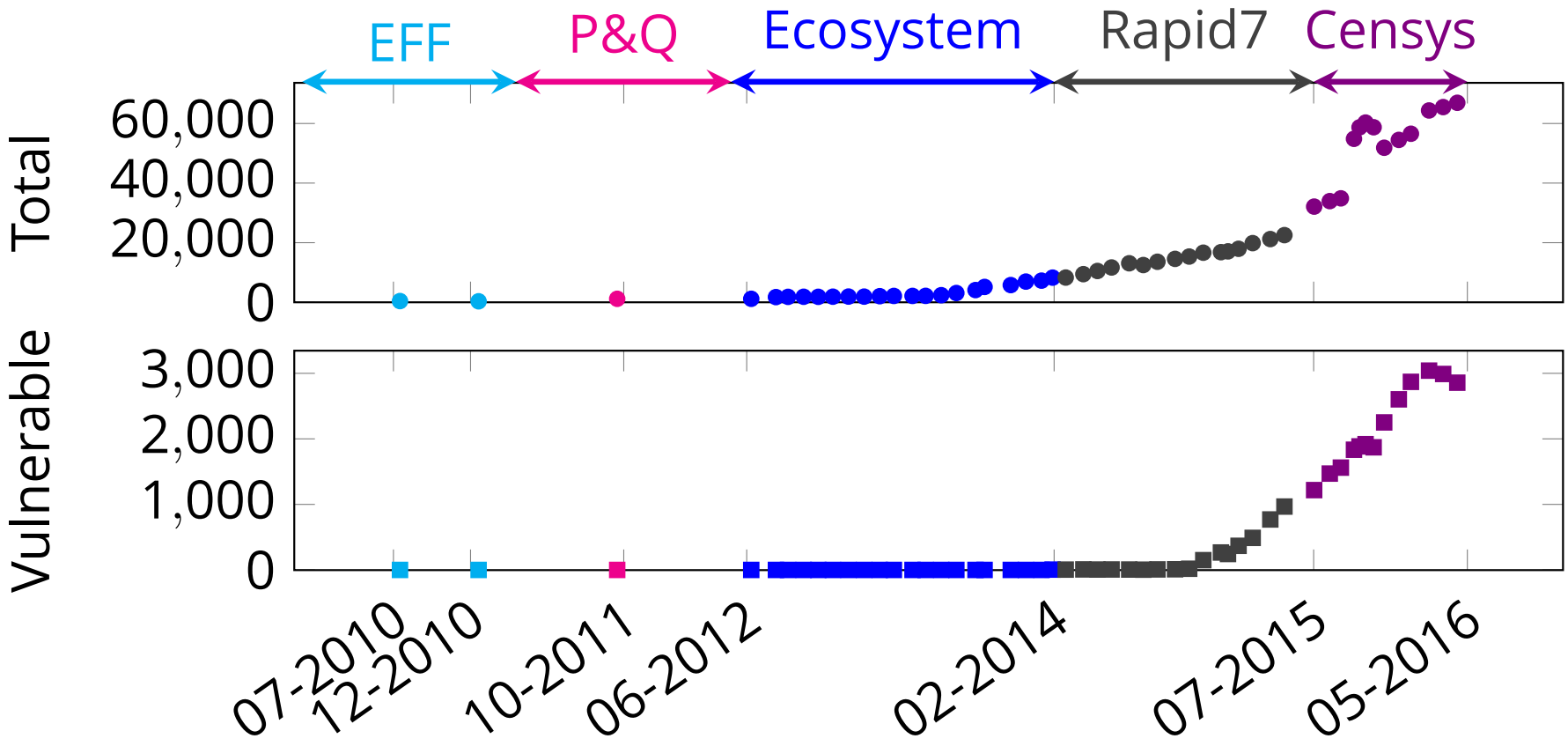
SRX Series Service Gateways, LN1000 Mobile Secure Router

- Security advisories in April, July 2012
- Majority of factored keys in 2012
- 30,000 Juniper-fingerprinted hosts went offline after Heartbleed



Huawei

- Introduced vulnerability in 2014
- Security advisory published Aug 2016



Disaster 3: Netscape SSL RNG [Goldberg Wagner 1996]

Problem: RNG is seeded with low entropy inputs.

```
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();
```

Disaster 4: ANSI X9.31 and the DUHK attack

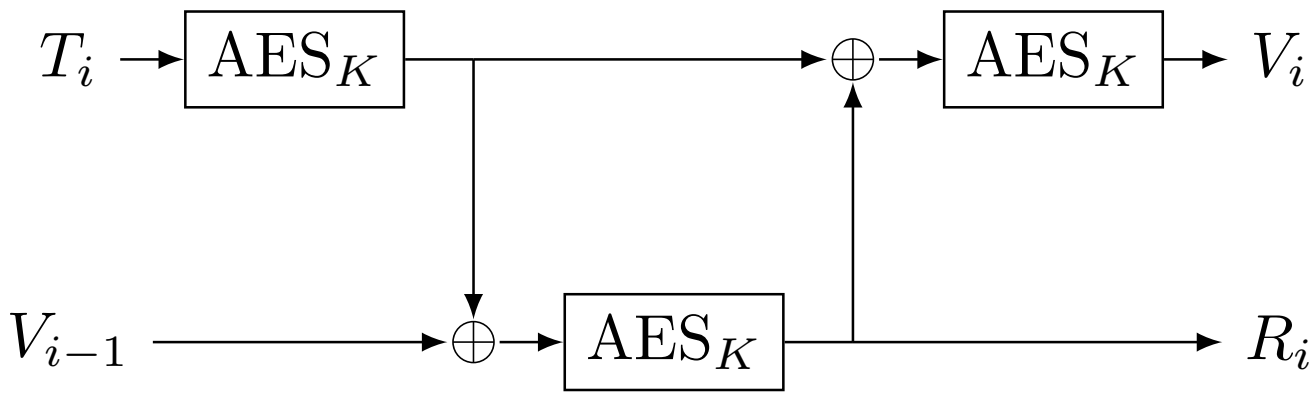
Practical state recovery attacks against legacy RNG implementations Shaanan Cohney, Matthew D. Green, Nadia Heninger. 2017.

Problem: RNG is seeded with low entropy inputs.
Solution: Seed with high entropy inputs.

Problem: User might use flawed PRNG design.
Solution: Don't use vulnerable designs.

The ANSI X9.31 PRNG

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



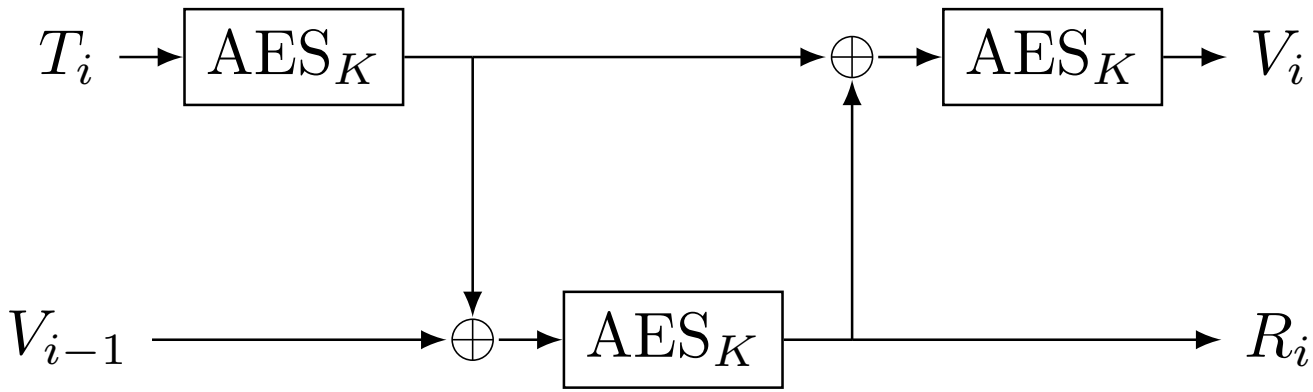
ANSI X9.31 PRNG History

- 1985: DES-based PRNG 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 from output by brute forcing timestamp.



NIST ANSI X9.31 RNG standardization failure

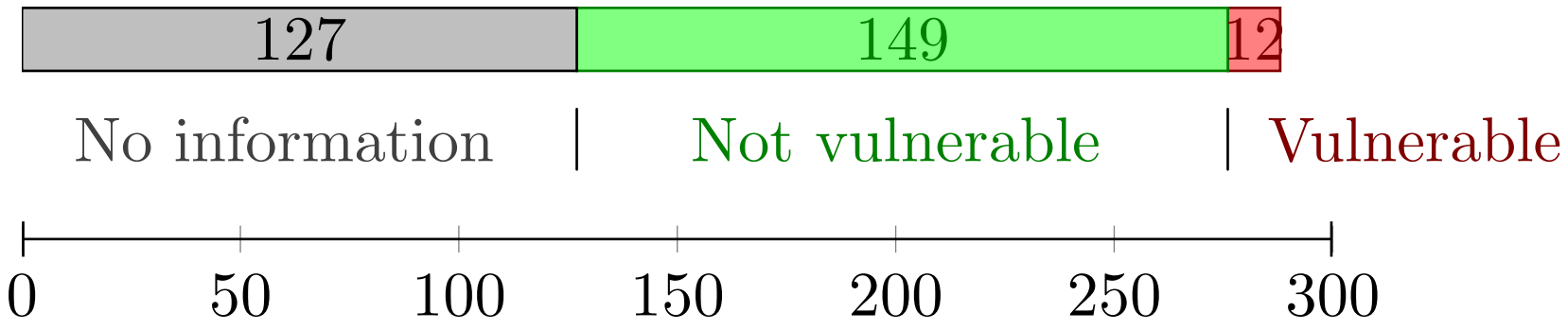
"For AES 128-bit key, let *K be a 128 bit key."

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

- Standard did *not* specify key should not be hard-coded.

Using FIPS 140 to find broken implementations

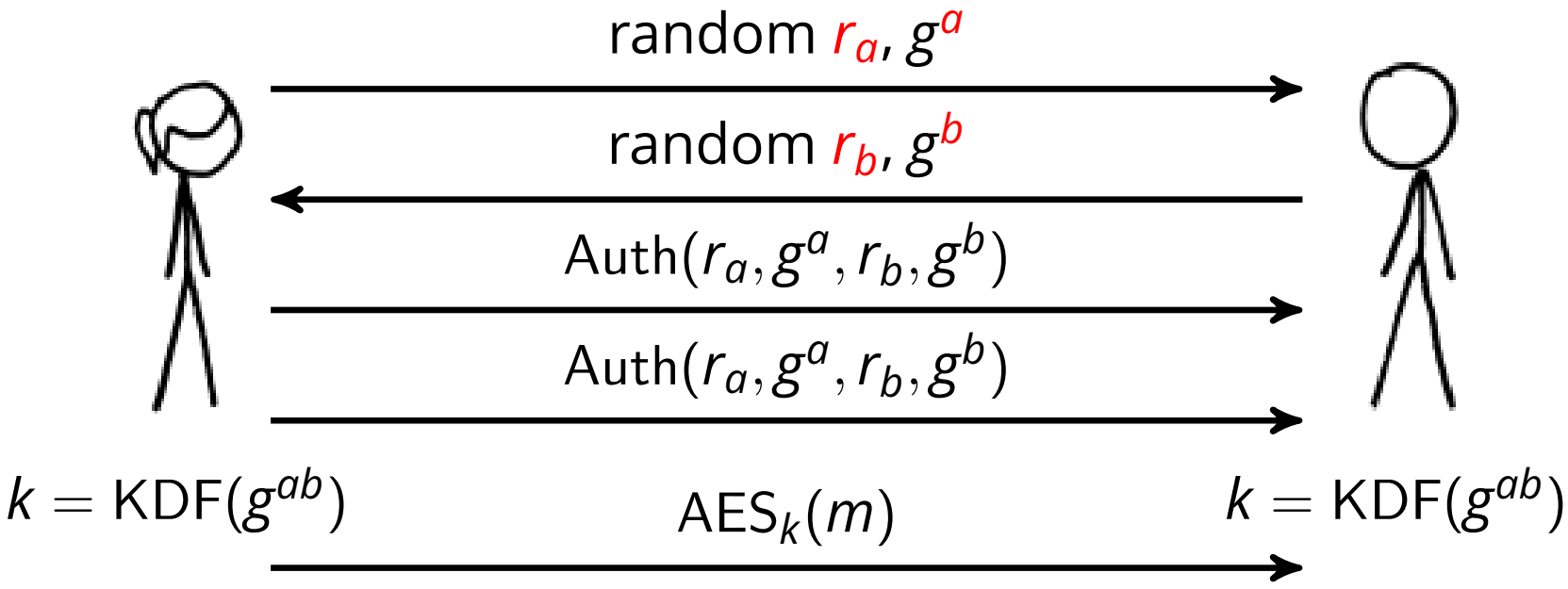
- FIPS 140 requires vendors to document key generation and storage policies in detail.
- We searched FIPS security policies to find documented hard-coded X9.31 keys.



"Compiled into binary" "statically stored in the code" "Hard Coded" "generated external to the module" "Stored in flash" "Static key, Stored in the firmware" "Entered in factory" "loaded at factory" "Static" "Embedded in FLASH" "Injected During Manufacture" "Hard-coded in the module"

Passive RNG state recovery in the IPsec protocol

Targeting Fortigate VPNs



- Need raw PRNG outputs for state recovery attack.
- Idea: Use the random nonces.
- After state recovered, then recover secret exponents.

Passive decryption for Fortigate IPsec VPNs

- FortiOS v4 hard-coded NIST test vector key
- 2^{25} work brute-forcing timestamps for state recovery
- Performed internet-wide scans and successfully recovered private keys against hosts in the wild.
- ANSI X9.31 RNG no longer included in FortiOS v5; FortiOS v4 patched since November 2016

Disaster 5: Dual EC DRBG

On the Practical Exploitability of Dual EC in TLS Implementations Checkoway, Fredrikson, Niederhagen, Everspaugh, Green, Lange, Ristenpart, Bernstein, Maskiewicz, Shacham. Usenix Security 2014.

Problem: User might use backdoored PRNG design.
Solution: Don't use vulnerable designs.

Dual EC DRBG

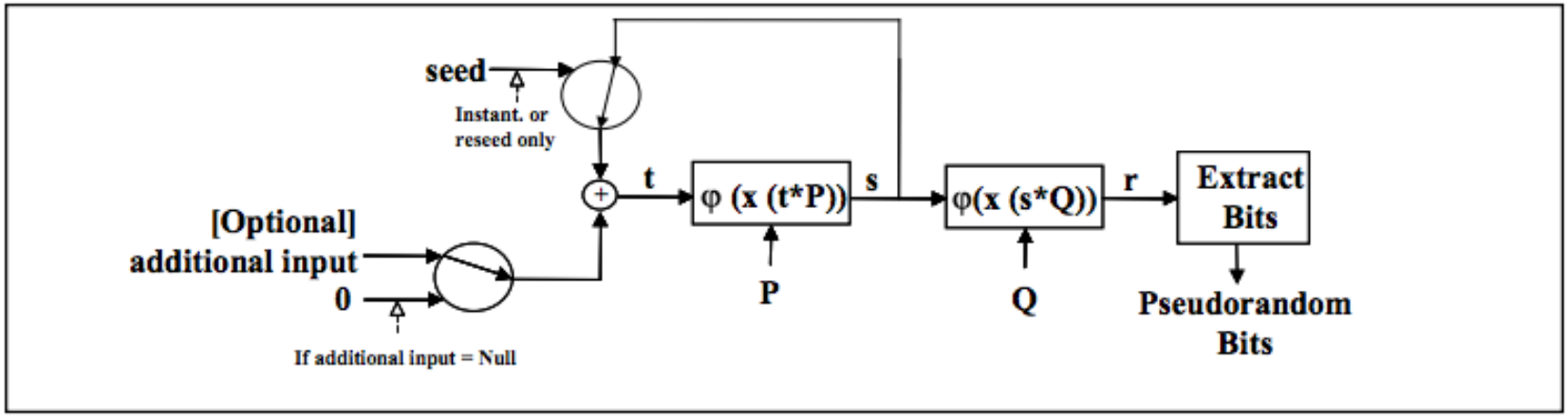


Figure 13: Dual_EC_DRBG

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

Dual EC DRBG History

- Early 2000s: NSA designed and pushed to standardize
- 2004: Published as part of ANSI X9.82 part 3 draft
- 2005: Standardized in NIST SP 800-90 draft
- 2007: Shumow, 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

Shumow and Ferguson 2007

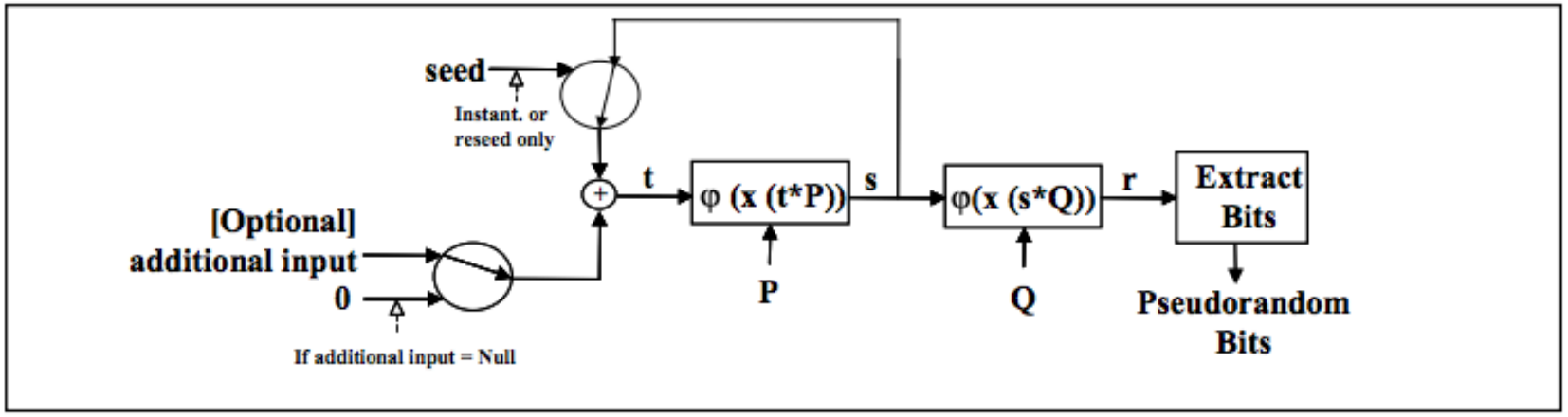


Figure 13: Dual_EC_DRBG

1. Assume attacker controls standard and constructs points with known relationship $P = dQ$.
2. Attacker gets 30 bytes of x-coordinate of sQ.
3. Attacker brute forces 2^{16} MSBs to generate candidates for sQ.
4. For each candidate sQ attacker compares $dsQ = sP$ to next output.

September 2013: NSA Bullrun in NY Times

- (TS//SI//REL TO USA, FVEY) Insert vulnerabilities into commercial encryption systems, IT systems, networks, and endpoint communications devices used by targets.
- (TS//SI//REL TO USA, FVEY) Collect target network data and metadata via cooperative network carriers and/or increased control over core networks.
- (TS//SI//REL TO USA, FVEY) Leverage commercial capabilities to remotely deliver or receive information to and from target endpoints.
- (TS//SI//REL TO USA, FVEY) Exploit foreign trusted computing platforms and technologies.
- (TS//SI//REL TO USA, FVEY) Influence policies, standards and specification for commercial public key technologies.
- (TS//SI//REL TO USA, FVEY) Make specific and aggressive investments to facilitate the development of a robust exploitation capability against Next-Generation Wireless (NGW) communications.

Dual EC Attack Complexity in TLS Implementations

Checkoway et al. 2014

Table 1: Summary of our results for Dual EC using NIST P-256.

Library	Default PRNG	Cache Output	Ext. Random	Bytes per Session	Adin Entropy	Attack Complexity	Time (minutes)
BSAFE-C v1.1	✓	✓	✓†	31–60	—	$30 \cdot 2^{15}(C_v + C_f)$	0.04
BSAFE-Java v1.1	✓		✓†	28	—	$2^{31}(C_v + 5C_f)$	63.96
SChannel I‡				28	—	$2^{31}(C_v + 4C_f)$	62.97
SChannel II‡				30	—	$2^{33}(C_v + C_f) + 2^{17}(5C_f)$	182.64
OpenSSL-fixed I*				32	20	$2^{15}(C_v + 3C_f) + 2^{20}(2C_f)$	0.02
OpenSSL-fixed III**				32	$35 + k$	$2^{15}(C_v + 3C_f) + 2^{35+k}(2C_f)$	$2^k \cdot 83.32$

* Assuming process ID and counter known. ** Assuming 15 bits of entropy in process ID, maximum counter of 2^k. See Section 4.3.
† With a library–compile-time flag. ‡ Versions tested: Windows 7 64-bit Service Pack 1 and Windows Server 2010 R2.

Disaster 6: The Juniper Dual EC Incident

A Systematic Analysis of the Juniper Dual EC Incident
Checkoway, Maskiewicz, Garman, Fried, Cohney, Green, Heninger, Weinmann, Rescorla, Shacham. CCS 2016.

Problem: User might use backdoored PRNG design.
Solution: Don’t use vulnerable designs.

the grugq
@thegrugq

Follow

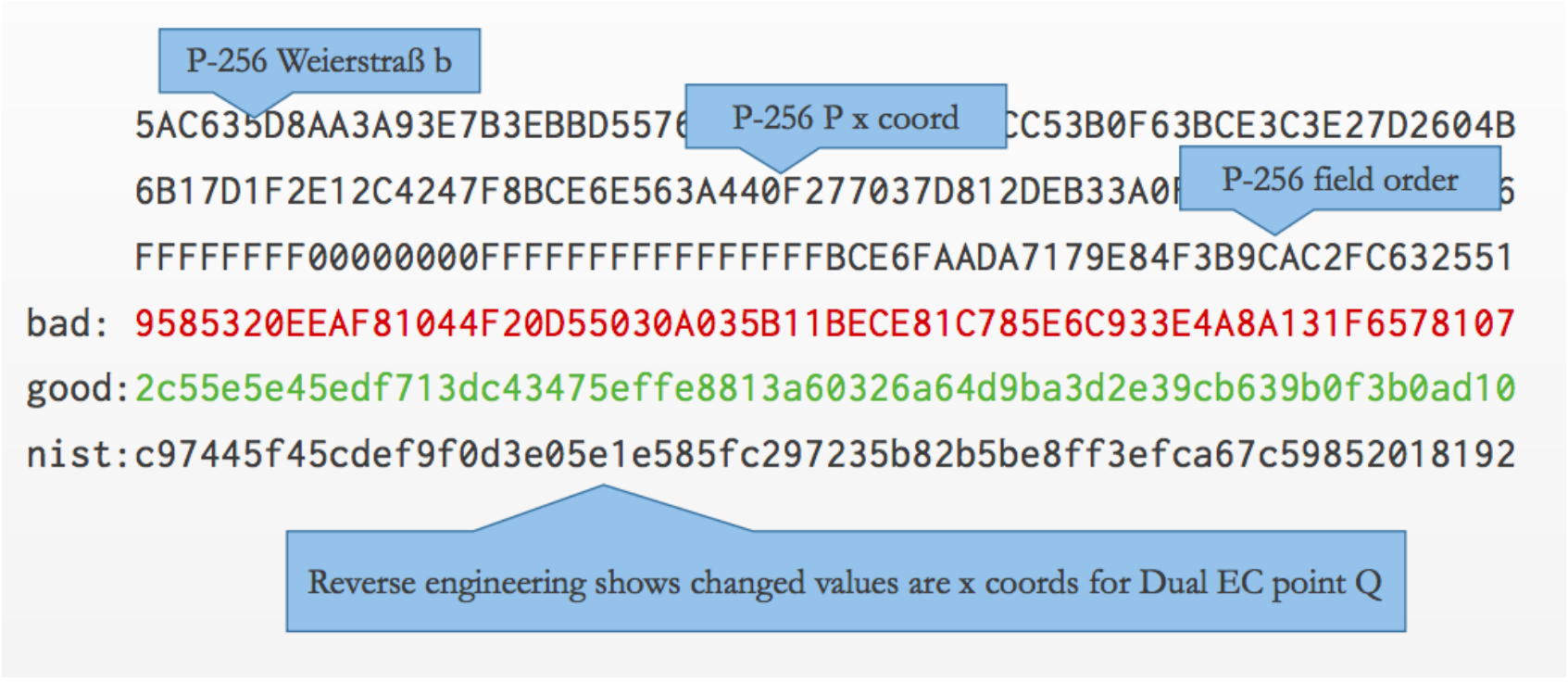
Woah! Juniper discovers a backdoor to decrypt VPN traffic (and remote admin) has been inserted into their OS source

Important Announcement about ScreenOS®

IMPORTANT JUNIPER SECURITY ANNOUNCEMENT CUSTOMER UPDATE: DECEMBER 20, 2015 Administrative Access (CVE-2015-7755) only affects ScreenOS 6.3.0r17 through

forums.juniper.net

Diff of VPN code change



Juniper cascaded Dual EC with ANSI X9.31

- ScreenOS only FIPS validated for ANSI X9.31, not Dual EC
- Juniper used non-default points for Dual EC

The following product families do utilize Dual_EC_DRBG, but do not use the pre-defined points cited by NIST:

1. ScreenOS*

* ScreenOS does make use of the Dual_EC_DRBG standard, but is designed to not use Dual_EC_DRBG as its primary random number generator. ScreenOS uses it in a way that should not be vulnerable to the possible issue that has been brought to light. Instead of using the NIST recommended curve points it uses self-generated basis points and then takes the output as an input to FIPS/ANSI X.9.31 PRNG, which is the random number generator used in ScreenOS cryptographic operations.

ScreenOS RNG “cascade” outputs raw Dual EC

```
void prng_generate(void) {
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;
}
```

ScreenOS RNG “cascade” outputs raw Dual EC

```
void prng_generate(void) {
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())      // always reseed with Dual EC
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)    // generate Dual EC output
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;
}
```

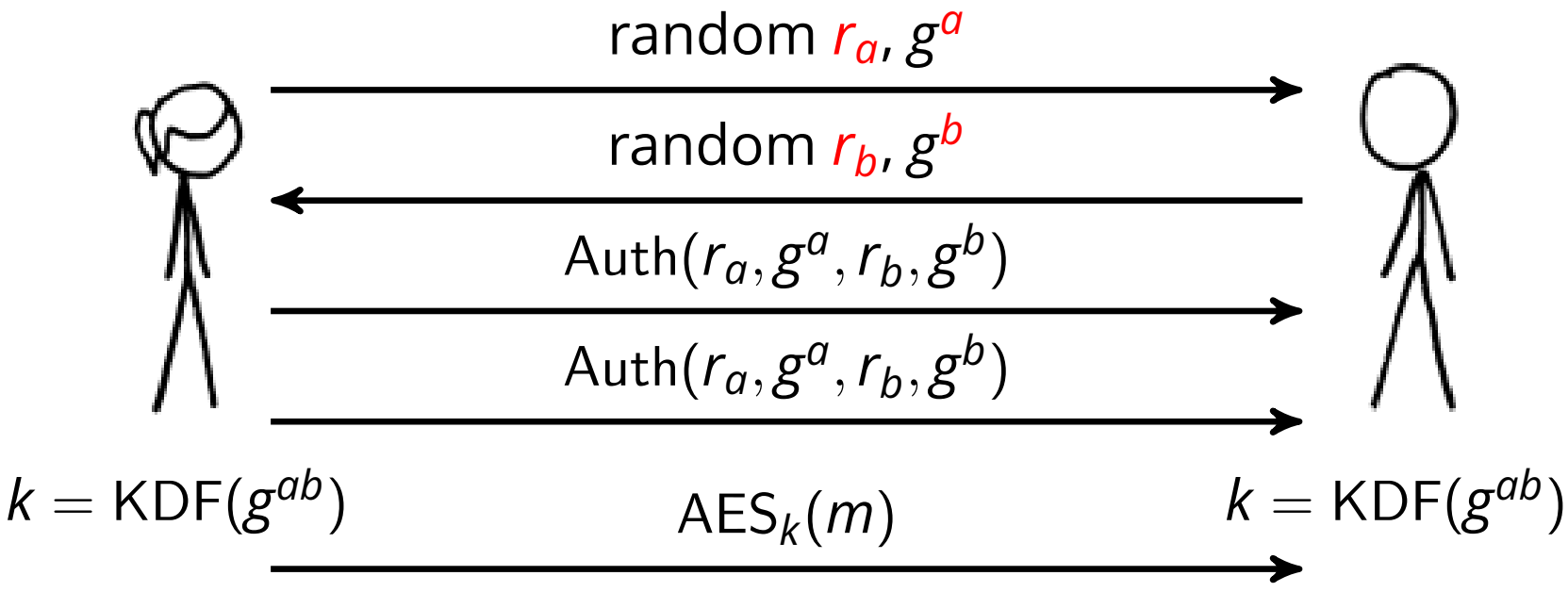
ScreenOS RNG “cascade” outputs raw Dual EC

```
void prng_generate(void) {
  prng_output_index = 0;     // global variable
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) { // never runs
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;   // set to 32
}
```

ScreenOS RNG “cascade” outputs raw Dual EC

```
void prng_generate(void) {
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    memcpy(&prng_temporary[prng_output_index], prng_block, 8); // reuses buffer
  }  // output is raw Dual EC output!
}

void prng_reseed(void) {
  blocks_generated_since_reseed = 0;
  if (dualec_generate(prng_temporary, 32) != 32)    // generate Dual EC output
    error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
  memcpy(prng_seed, prng_temporary, 8);
  prng_output_index = 8;
  memcpy(prng_key, &prng_temporary[prng_output_index], 24);
  prng_output_index = 32;
}
```

Passive state recovery in ScreenOS IPsec



- Use random nonces to carry out state recovery attack.
- ScreenOS used 32-byte nonce $\implies$ efficient attack.
- After state recovered, then recover secret exponents.

ScreenOS Version History

ScreenOS 6.1.0r7

- ANSI X9.31
- Seeded by interrupts
- Reseed every 10k calls
- 20-byte IKE nonces

ScreenOS 6.2.0r0 (2008)

- Dual EC $\rightarrow$ ANSI X9.31
- Reseed bug exposes raw Dual EC
- Reseed every call
- Nonces generated before keys
- 32-byte IKE nonces

- Attacker changed constant in 6.2.0r15 (2012).
- But passive decryption enabled in earlier release.
- Juniper’s "fix" was to reinstate original Q value. After our work they removed Dual EC completely.

Discussion

- We see the same vulnerabilities over and over again.
- Gaps between theory and practice.
- Difficult to eradicate vulnerabilities from devices.
- Need better automated methods for discovering RNG bugs.
- Backdoors can be repurposed.

How to generate random numbers

- Not everything is broken! Other RNG constructions in NIST SP 800-90a are mostly fine if implemented correctly and securely!
- Intel RDRAND, RDSEED provide fast hardware RNG interfaces. And are probably not backdoored.
- Linux `getrandom()` provides a better interface than `urandom` or `random`.