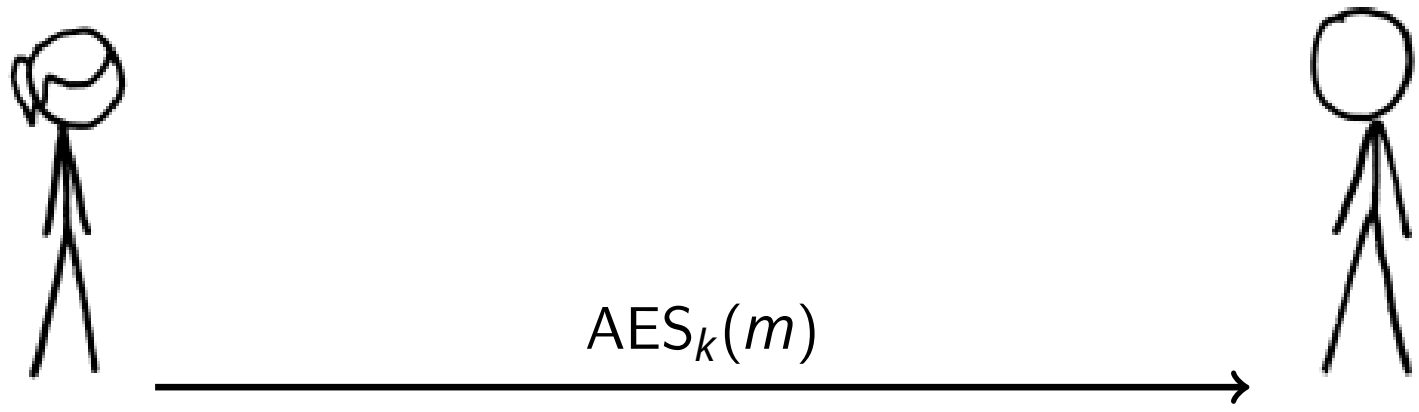
How not to generate random numbers

Nadia Heninger

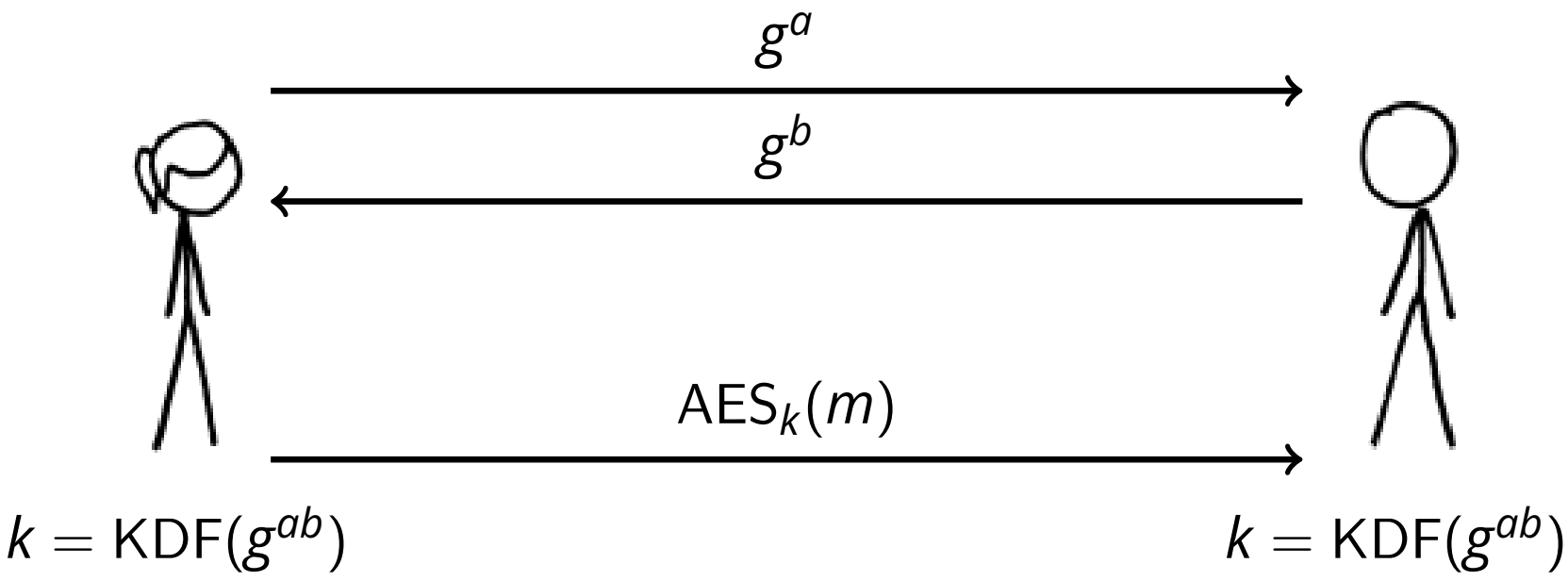
University of Pennsylvania

June 15, 2018

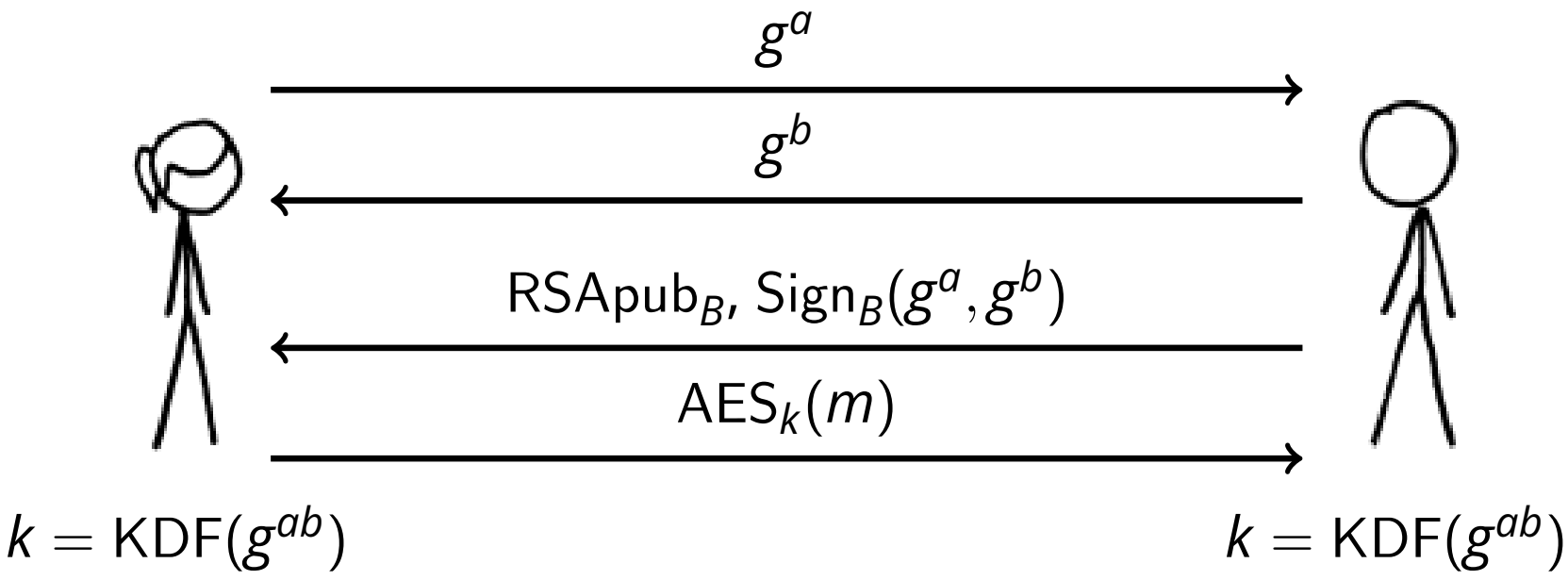
A crash course in cryptographic protocols



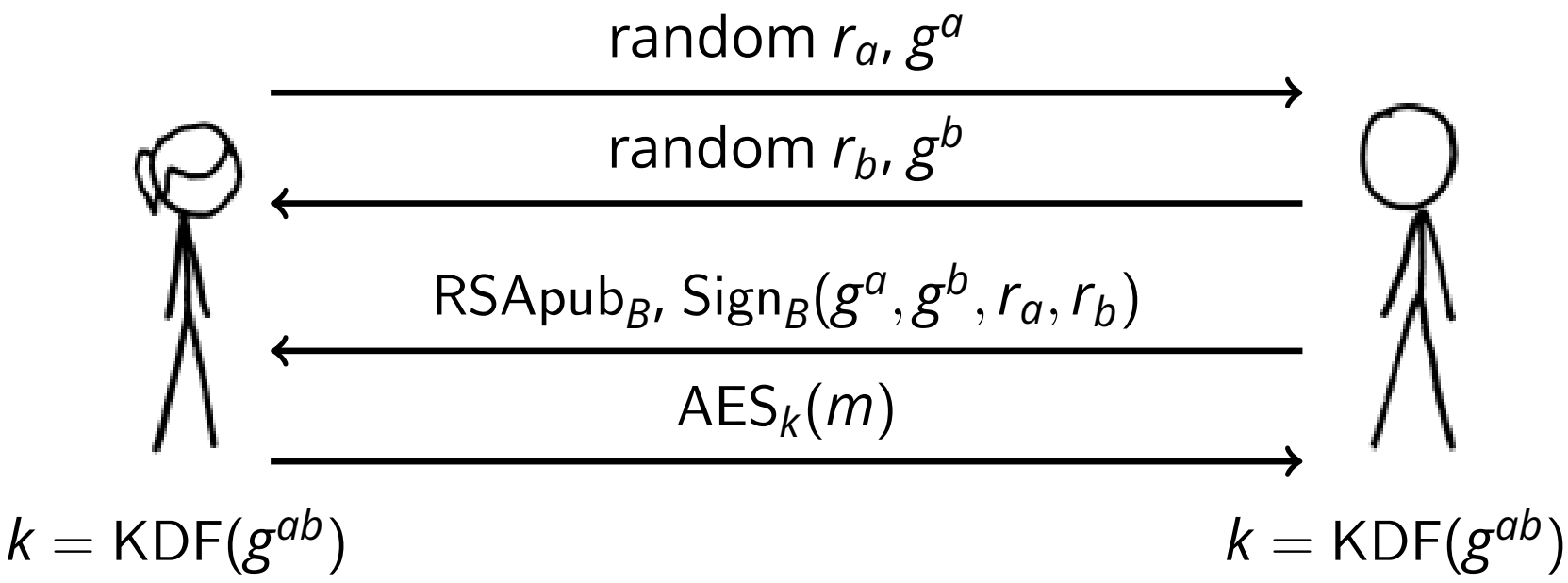
A crash course in cryptographic protocols



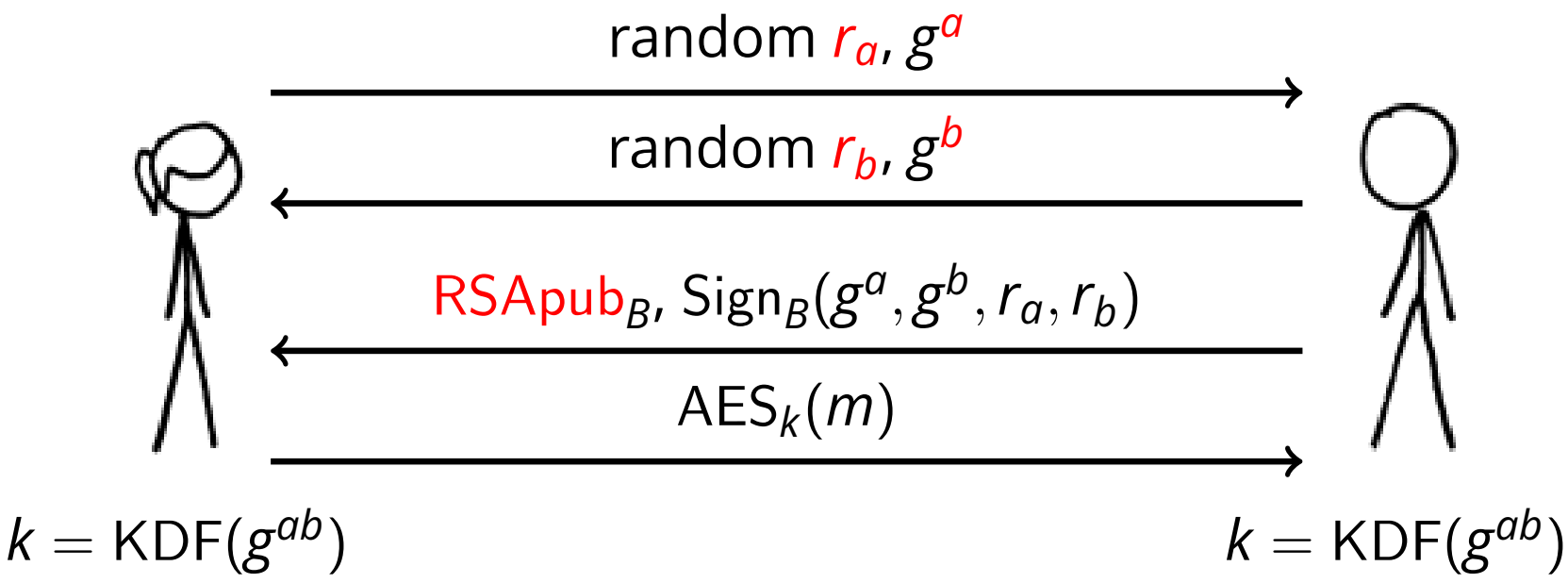
A crash course in cryptographic protocols



A crash course in cryptographic protocols



A crash course in cryptographic protocols



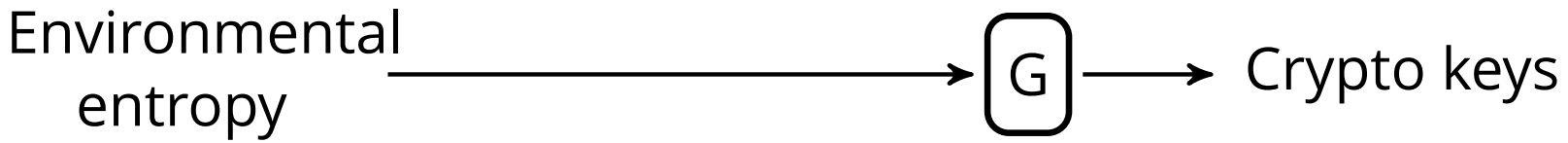
“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.”

–John von Neumann

Cryptographic pseudorandomness in theory

Definition

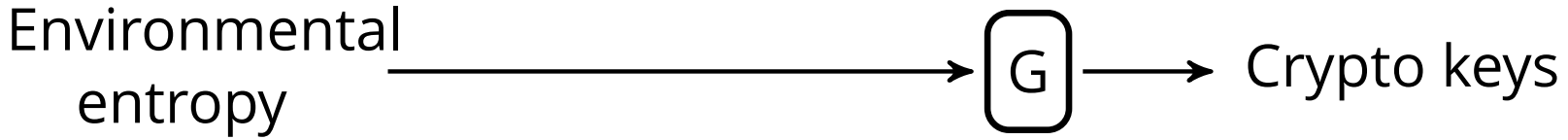
A *pseudorandom generator* is a polynomial-time deterministic function G mapping n-bit strings into $\ell(n)$-bit strings for $\ell(n) \geq n$ whose output distribution $G(U_n)$ is computationally indistinguishable from the uniform distribution $U_{\ell(n)}$.



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 $U_{\ell(n)}$.

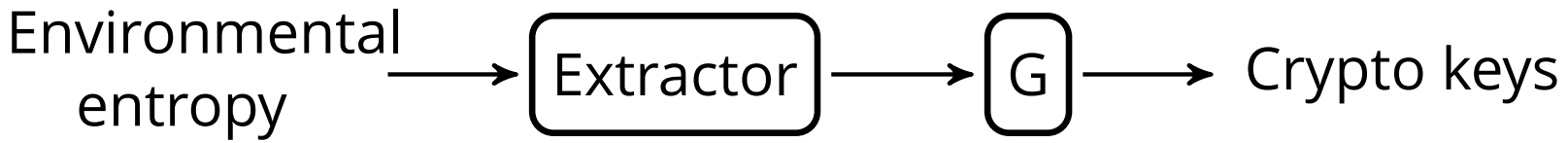


Problem: Environmental entropy not uniformly distributed.

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"

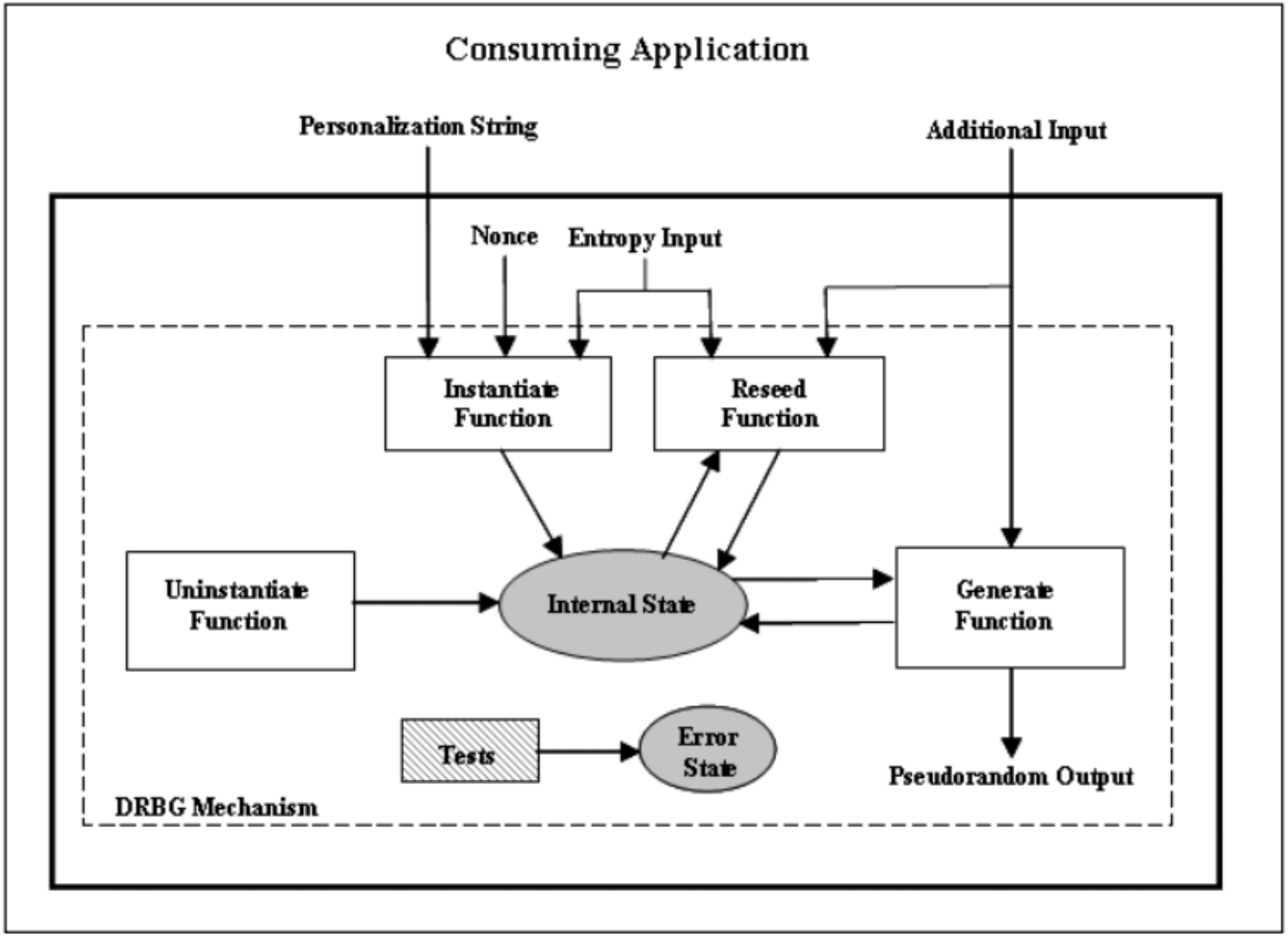


Figure 1: DRBG Functional Model

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:** User might request output before seeding.

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

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)

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

Defenses

- Possible to automatically detect unseeded PRNGs in source code in some circumstances. [Dörre Klebanov 2016]
- How to make more rigorous?

Disaster 2: Shared RSA factors

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*

Underlying cause: Failure to seed PRNG.

RSA and factoring

Public Key
$(N = pq, e)$

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

RSA and factoring

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 factoring

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
768-bit RSA modulus:
2.5 calendar years
[Kleinjung et al. 2010]

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.**

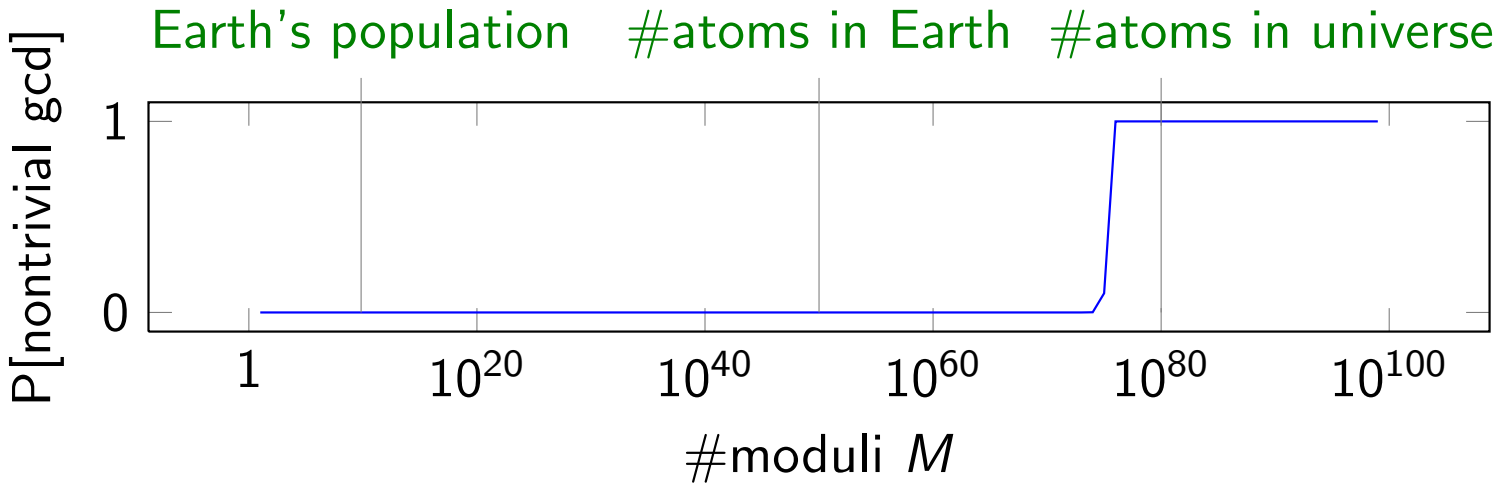
Should we expect to find prime collisions in the wild?

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]

Widespread RNG failures on low resource devices

We accidentally found *multiple independent cascading PRNG failures*.

Factor #1: Weak keys generated by low resource devices ($>$ 50 manufacturers).

1. Linux PRNG inputs: keyboard, mouse, disk
2. OpenSSL inputs: time, pid, OS PRNG
3. Headless or embedded devices lack these entropy sources.

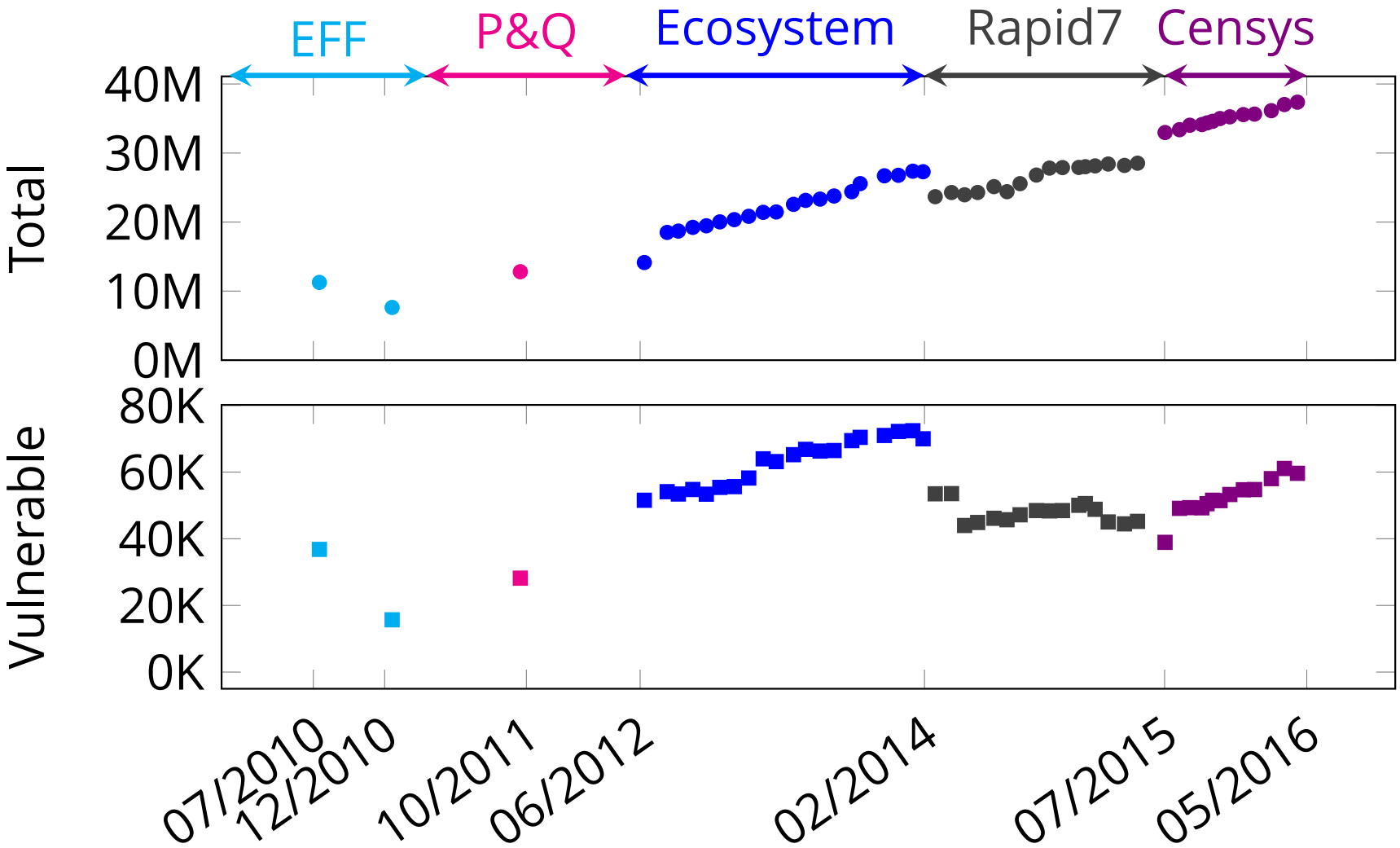
Factor #2: Boot-time entropy hole on Linux PRNG

- Devices automatically generated keys on first boot.
- Linux PRNG had not yet been seeded when queried by OpenSSL.
- Fixed since July 2012.

Follow-up study: Six years of factoring keys

Question: Do vendors actually fix flaws after vulnerability disclosure?

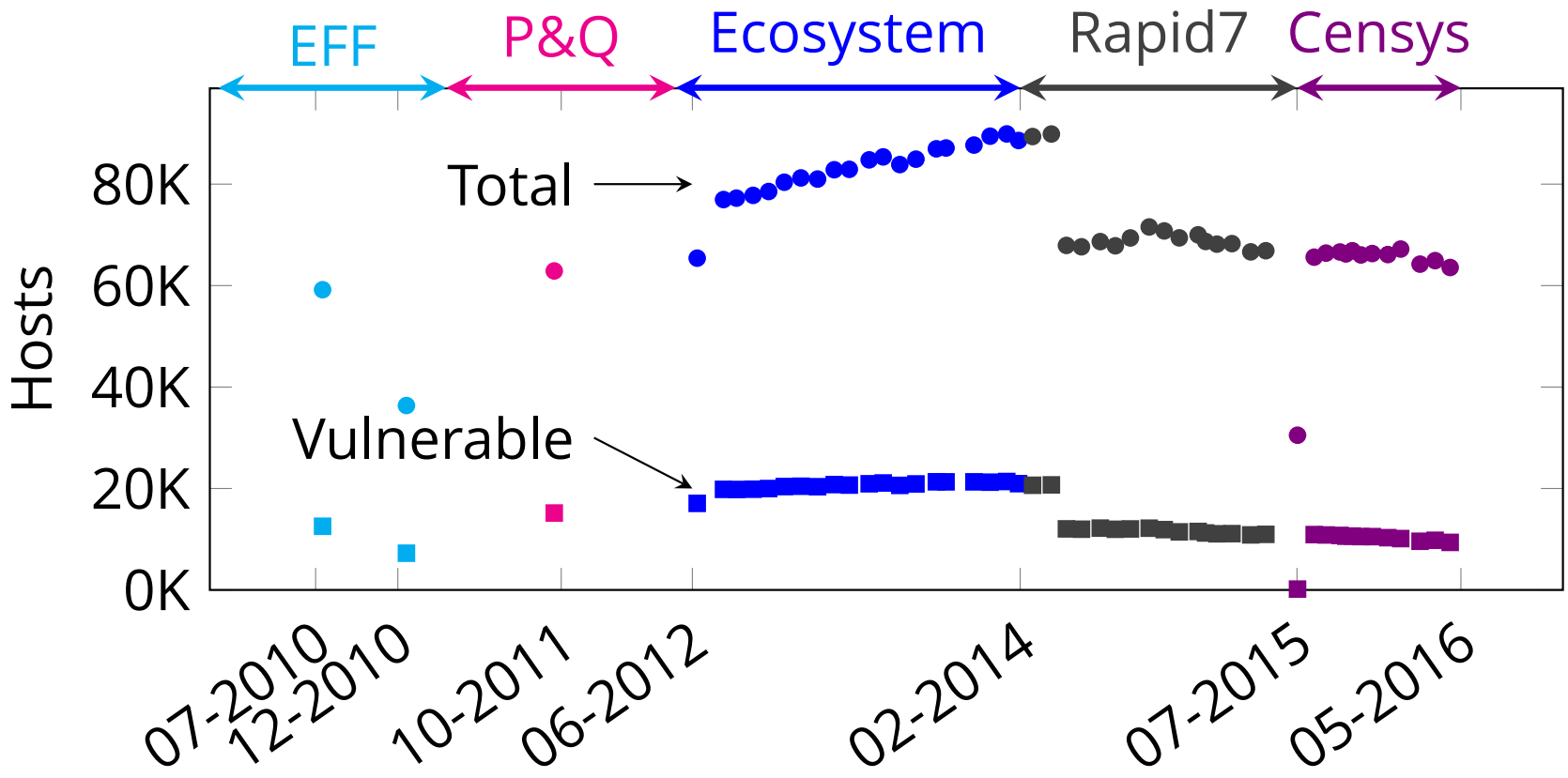
- 65 million distinct HTTPS certificates : 2.2% vulnerable
- 1.5 billion HTTPS host records : 0.19% vulnerable



Juniper

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

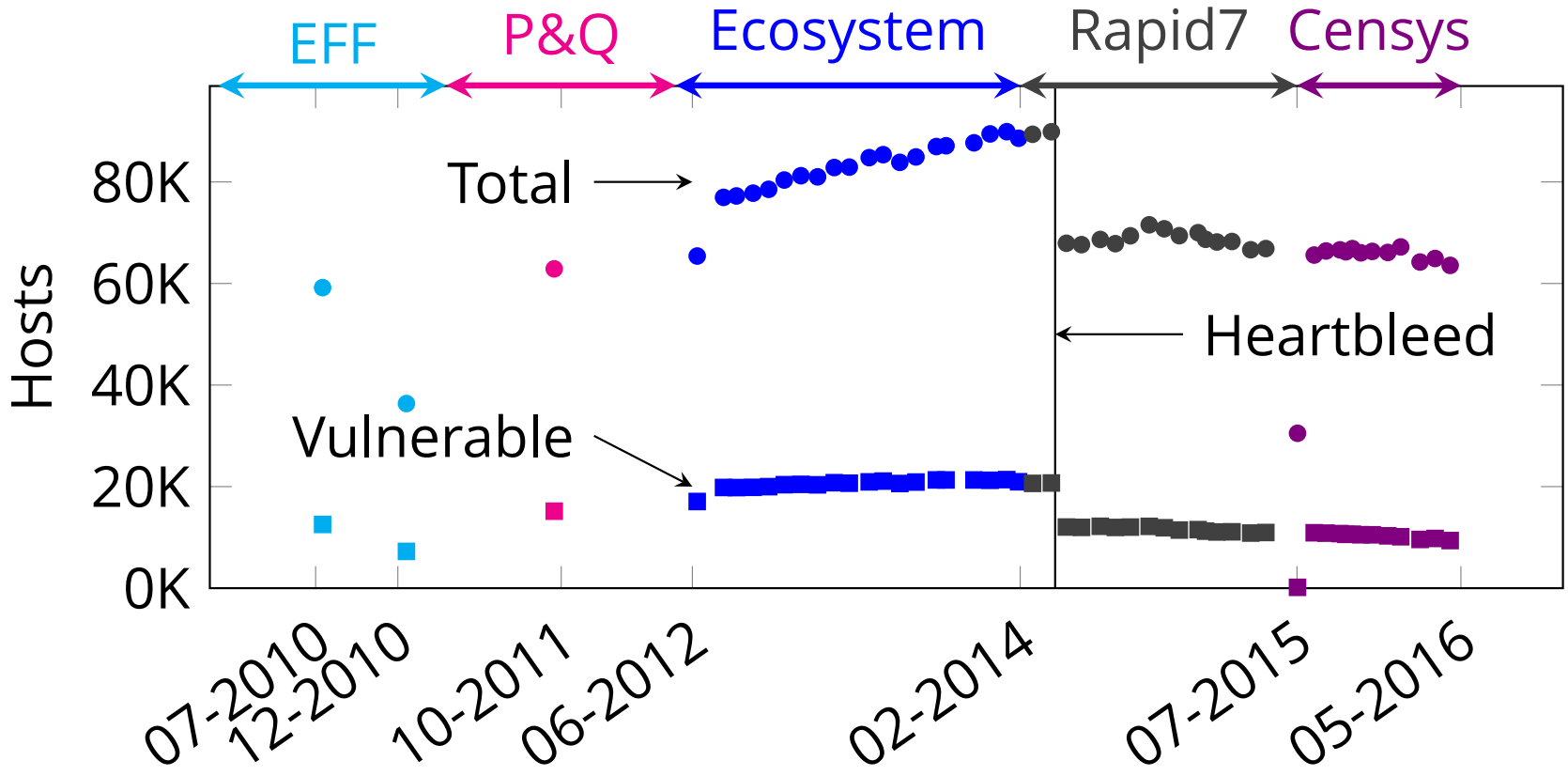
- Security advisorie in April, July 2012
- Majority of factored keys in 2012 were Juniper hosts
- Weird behavior in April 2014



Juniper

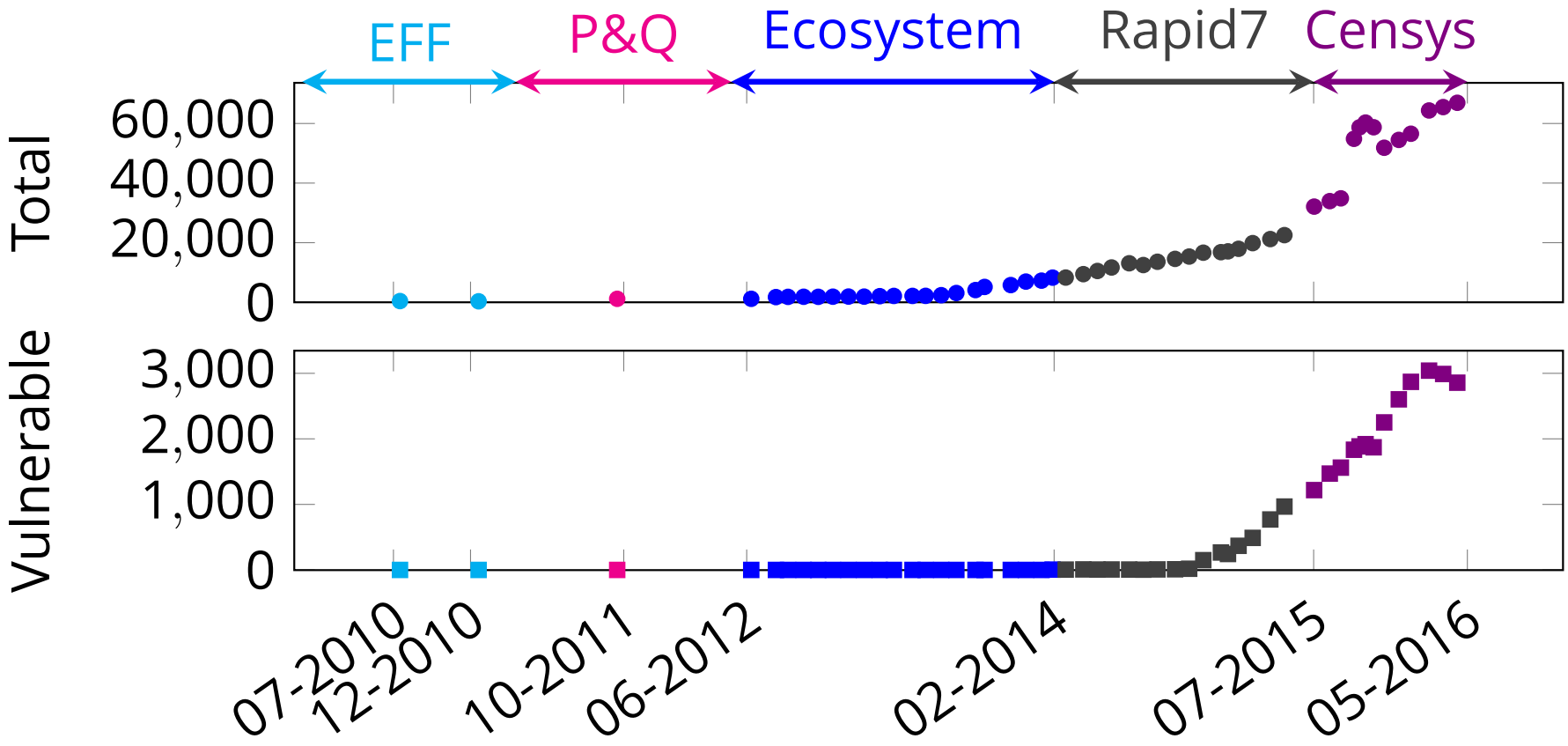
SRX Series Service Gateways (SRX100, SRX110, SRX210, SRX220, SRX240, SRX550, SRX650), LN1000 Mobile Secure Router

- 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



Discussion and Lessons

- Widespread vulnerabilities were hiding in plain sight for years.
- Difficult to eradicate vulnerabilities from fundamental infrastructure.
- Disclosure process flawed: $>$ 50% of vendors never responded.
- Patching rates are low to nonexistent for networked devices.
- Big gap between theory and practice.
  - Theoretical models did not reflect reality.
  - Practitioners have incorrect received knowledge about RNG threats.

Disaster 3: Netscape SSL RNG [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();
```

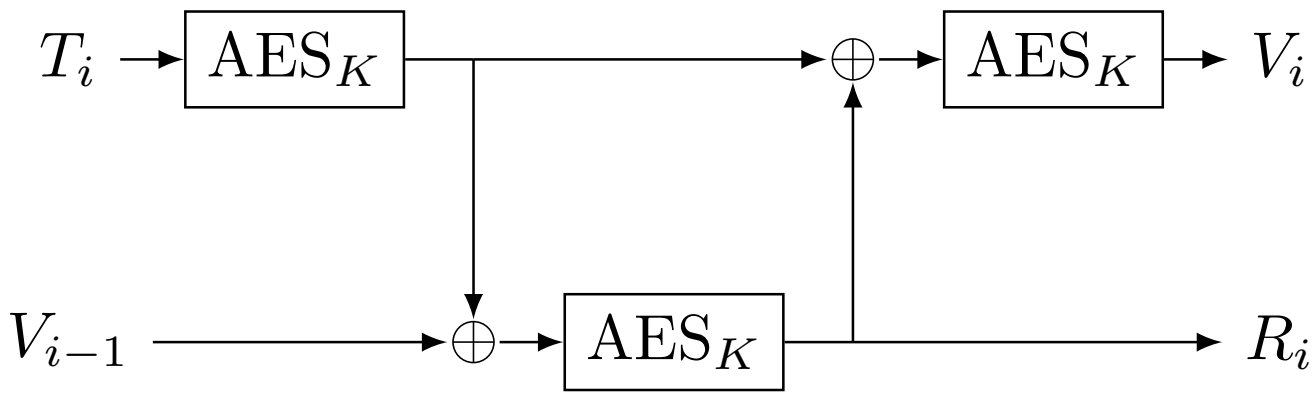
Disaster 4: ANSI X9.31 and the DUHK attack

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

Underlying cause: Seeding invertible PRNG with insufficient entropy.

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.



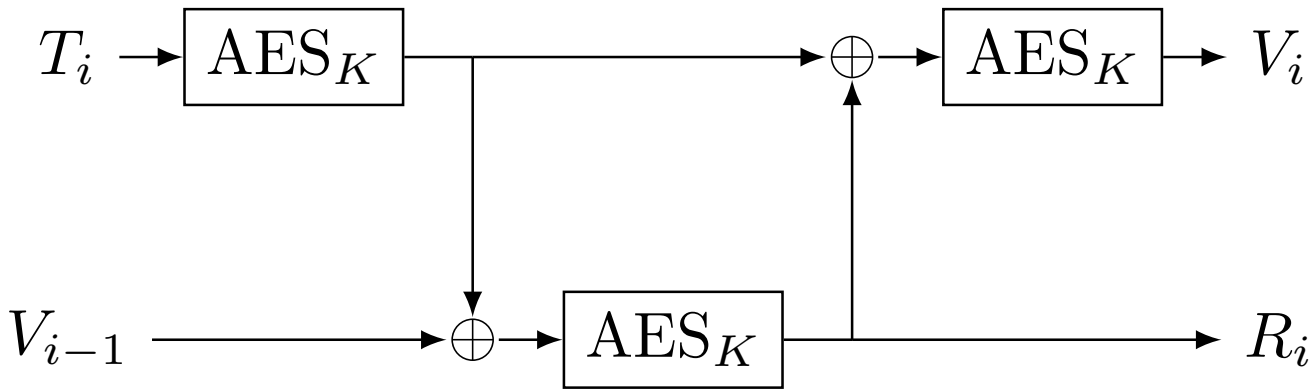
ANSI X9.31 PRNG History

- 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

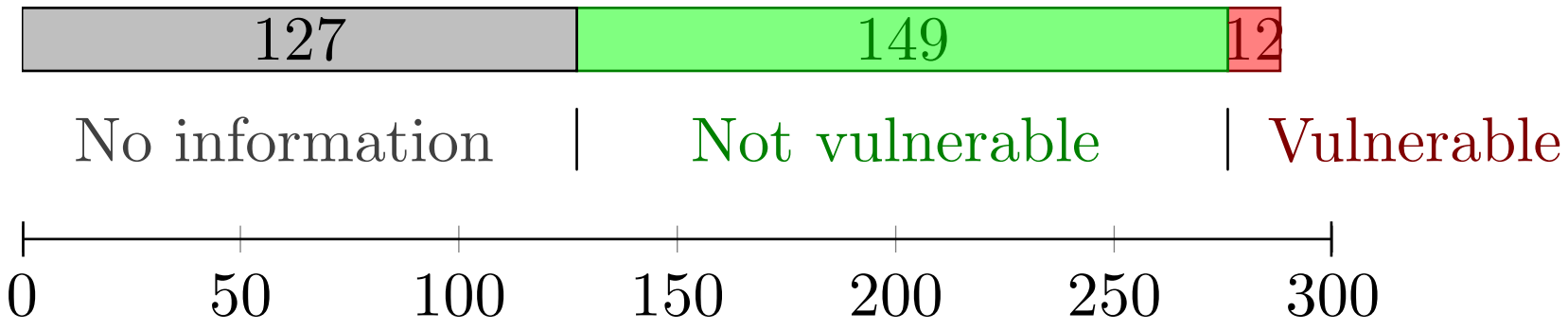
"For AES 128-bit key, let *K be a 128 bit key."

"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

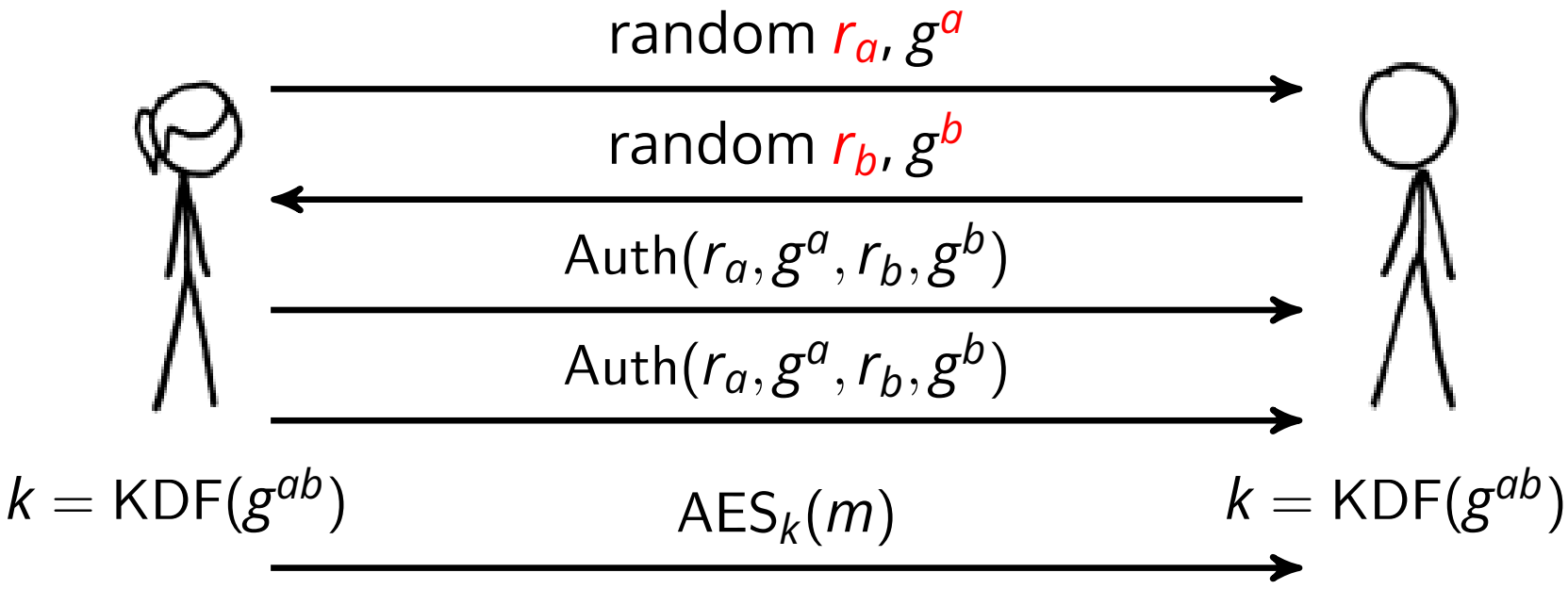
- FIPS 140 requires vendors to document key generation and storage policies in detail.
- We searched FIPS security policies to find documented hard-coded X9.31 keys.



"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

Discussion and Lessons

- Impact of academic work not always noticed in real world.
- This is not a “NOBUS” backdoor because it is symmetric.
- Weak design continued to be used long after better constructions were known.
- This type of flaw may explain some of NSA’s passive VPN decryption capabilities.
- Disclosure process is flawed: 10 of 12 vendors we contacted never responded.
- FIPS security validation does not imply a security audit.

Defensive work

- Formal verification can help prove that designs match security model.
- Multiple (recent!) security models for real-world PRNGs. (e.g. [Dodis et al. 2013])
- New attacks introduce new threat models.
- Is it possible to detect (flawed) algorithms in a binary?

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.

Underlying cause: Backdoored PRNG design.

Dual EC DRBG

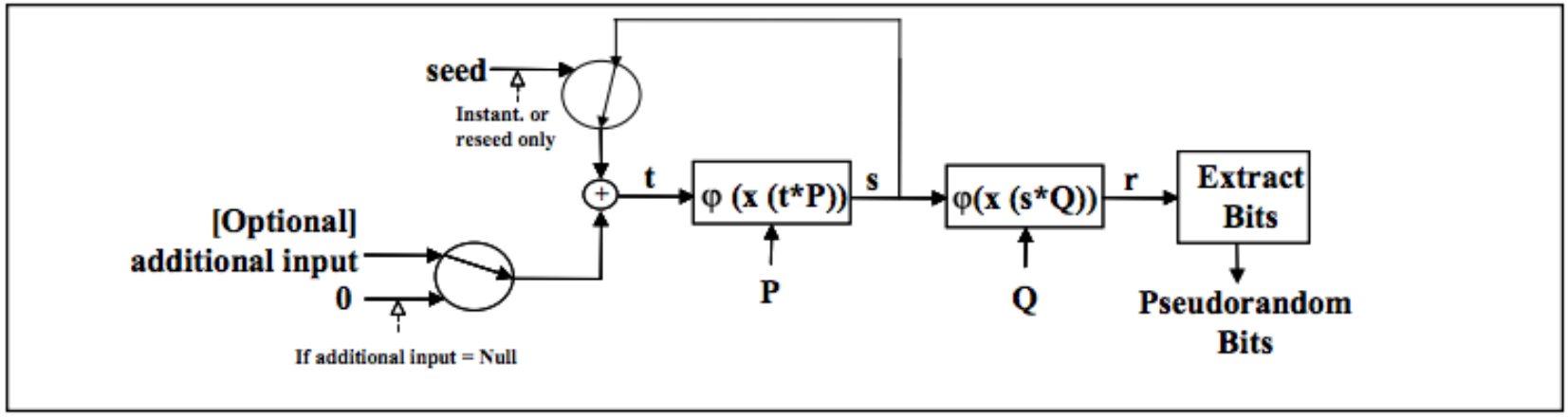


Figure 13: Dual_EC_DRBG

- Parameters: Pre-specified elliptic curve points P and Q.
- Seed: 32-byte integer s
- State: x-coordinate of point sP. ($\phi(x(sP))$ above.)
- Update: $t = s \oplus$ optional additional input. State $s = x(tP)$.
- Output: At state s, compute x-coordinate of point $x(sQ)$, discard top 2 bytes, output 30 bytes.

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

Shumow and Ferguson 2007

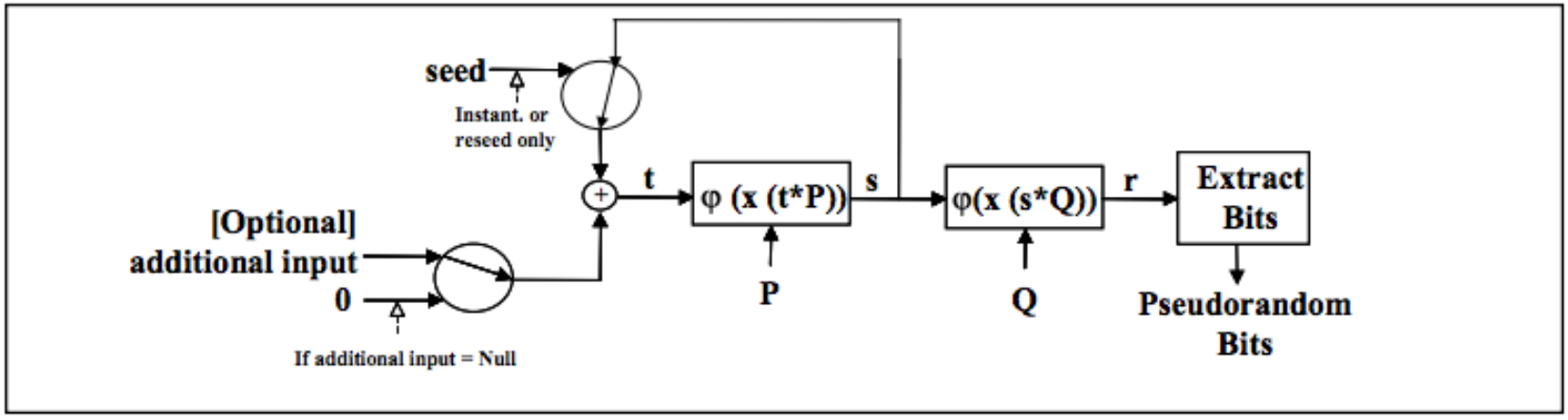


Figure 13: Dual_EC_DRBG

1. Assume attacker controls standard and constructs points with known relationship $P = dQ$.
2. Attacker gets 30 bytes of x-coordinate of sQ. Attacker brute forces 2^{16} MSBs, gets 2^{17} possible y-coordinates, ends up with 2^{15} candidates for sQ.
3. For each candidate sQ attacker computes $dsQ = sP$ and compares 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.

Underlying cause: Backdoored PRNG design.

the grugq
@thegrugq

Follow

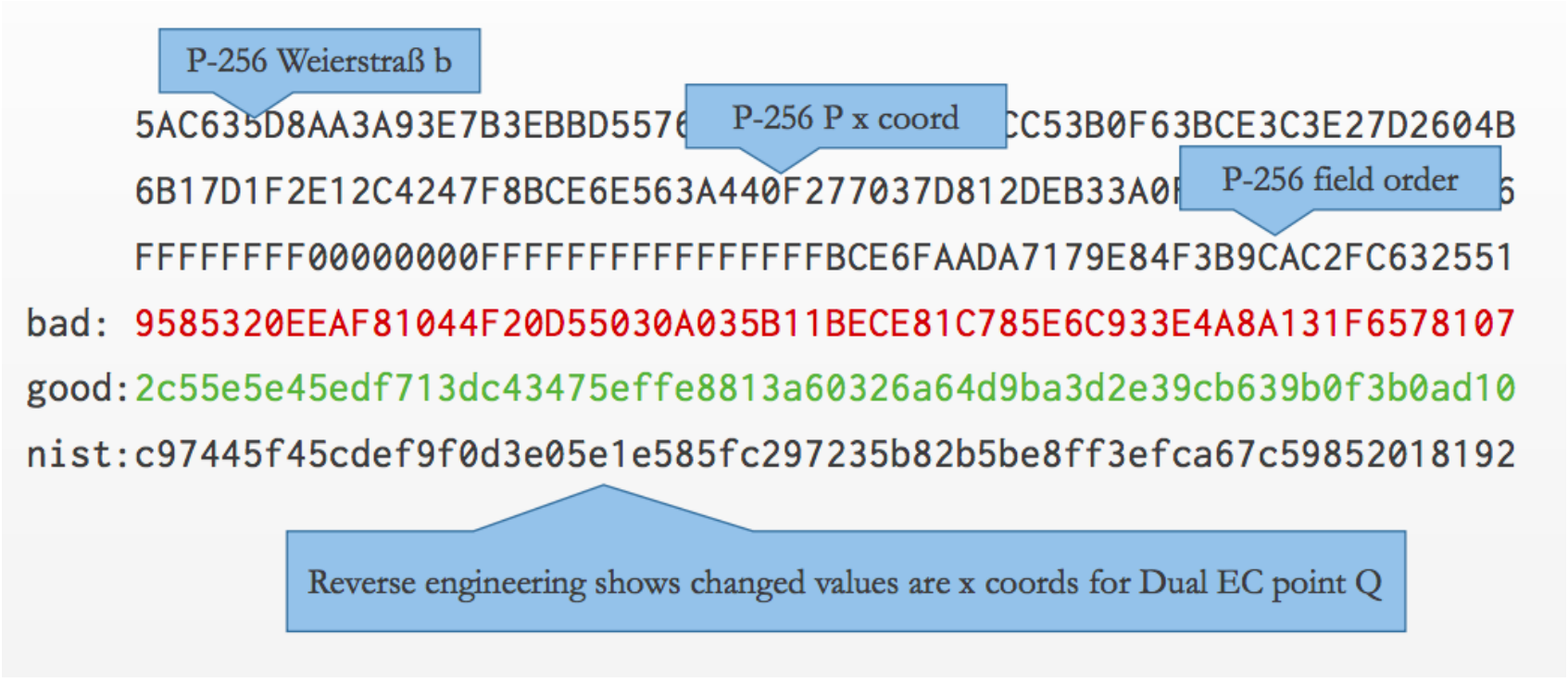
Woah! Juniper discovers a backdoor to decrypt VPN traffic (and remote admin) has been inserted into their OS source

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 implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    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 implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();     // conditional reseed
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    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 implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    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); // copy output
  prng_output_index = 32;
}
```

ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block); // gen output
    // FIPS checks removed for clarity
    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 implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;      // global variable
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())       // always true
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    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 implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    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)     // global variable
    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
}
```

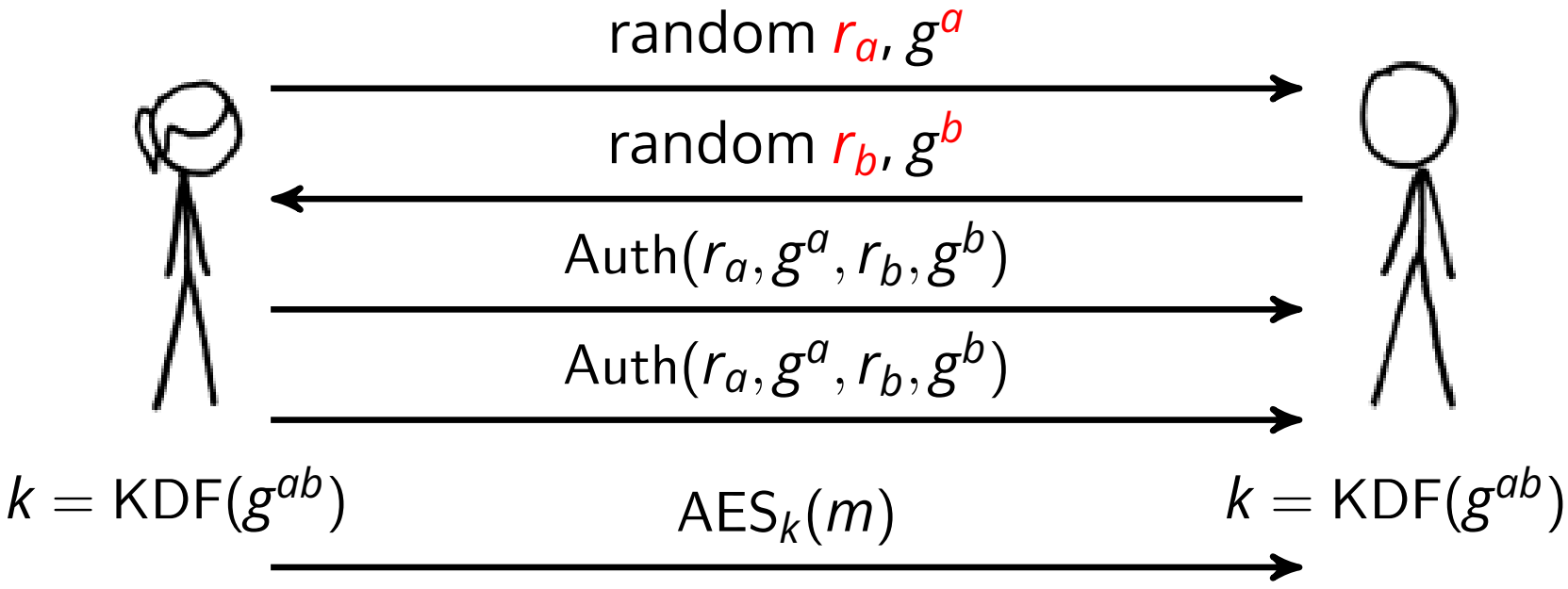
ScreenOS RNG implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) { // never runs
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8); // reuses buffer
  }
}

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 implementation

```
void prng_generate(void) {
  int time[2];
  time[0] = 0;
  time[1] = get_cycles();
  prng_output_index = 0;
  ++blocks_generated_since_reseed;
  if (!one_stage_rng())
    prng_reseed();
  for (; prng_output_index <= 0x1F; prng_output_index += 8) {
    // FIPS checks removed for clarity
    x9_31_generate_block(time, prng_seed, prng_key, prng_block);
    // FIPS checks removed for clarity
    memcpy(&prng_temporary[prng_output_index], prng_block, 8);
  }  // output is raw Dual EC output!
}

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;
}
```

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.
- We demonstrated attack with our own backdoored P, Q.

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 and Lessons

- “NOBUS” backdoors can be repurposed.
- Don’t know how Juniper’s parameters were generated, or who wrote their Dual EC cascade.
- Juniper wasn’t certified for Dual EC, so it wasn’t on the radar of researchers who looked for vulnerable implementations. Who else are we missing?
- Could we detect both implementations and bugs automatically?
- How do we prevent backdoors in standards?

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`.