Mining Your *P*s and *Q*s
Detection of Widespread Weak Keys in Embedded Devices

Nadia Heninger

UC San Diego

↓

Microsoft Research New England

Zakir Durumeric Eric Wustrow Alex Halderman

University of Michigan

January 10, 2012

—or—

Requirements for random number generators II

—or—

Requirements for random number generators II:

Seed them.

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

`factorable.net`

Cryptography and randomness

Randomness is required for cryptography.

Long list of failures in practice:

1996 Goldberg and Wagner Netscape SSL vulnerability

2008 Bello Debian OpenSSL entropy disaster

Cryptography and randomness

Randomness is required for cryptography.

Long list of failures in practice:

1996 Goldberg and Wagner Netscape SSL vulnerability

2008 Bello Debian OpenSSL entropy disaster

Our research plan:

1. Acquire cryptographic keys.
2. Look for obvious key generation problems.

Step 1: Acquire cryptographic keys.

1. Scan IPv4 space on port 443 (https) and 22 (ssh).
   - nmap from 25 hosts on Amazon ec2, $<$ 24 hours.
2. For each host with port open, initiate handshake.
   - 3 hosts, $<$ 48 hours
3. Store certificate/key.

Step 1: Acquire cryptographic keys.

1. Scan IPv4 space on port 443 (https) and 22 (ssh).
   - nmap from 25 hosts on Amazon ec2, $<$ 24 hours.
2. For each host with port open, initiate handshake.
   - 3 hosts, $<$ 48 hours
3. Store certificate/key.

	Our SSH Scans (02-04/2012)	Our SSL Scan (10/2011)	EFF Observatory (10/2010)
Open Port	23,237,081	28,923,800	16,000,000
Handshake	10,216,369	12,828,612	7,704,837
Distinct Certs	-	5,847,957	4,021,766
Browser-Trusted	-	1,956,267	1,455,391

Step 2: Look for problems. Repeated keys.

- Two hosts share same public key:
  $\rightarrow$ both know private key of the other.

Step 2: Look for problems. Repeated keys.

- Two hosts share same public key:
  $\rightarrow$ both know private key of the other.

	SSH	SSL
Handshake	10,216,369	12,828,612
Distinct Certs	-	5,847,957
Browser-Trusted	-	1,956,267
Distinct RSA keys	3,821,639	5,656,519
Distinct DSA keys	2,789,662	6,241

Looking for problems. Repeated public keys

Many valid (and common) reasons to share keys:

- Shared hosting situations. Virtual hosting.
- A single organization registers many domain names with the same key.

Looking for problems. Repeated public keys

Common (and unwise) reasons to share keys:

- Device default certificates/keys.
- Apparent entropy problems in key generation.

Looking for problems. Repeated public keys

Common (and unwise) reasons to share keys:

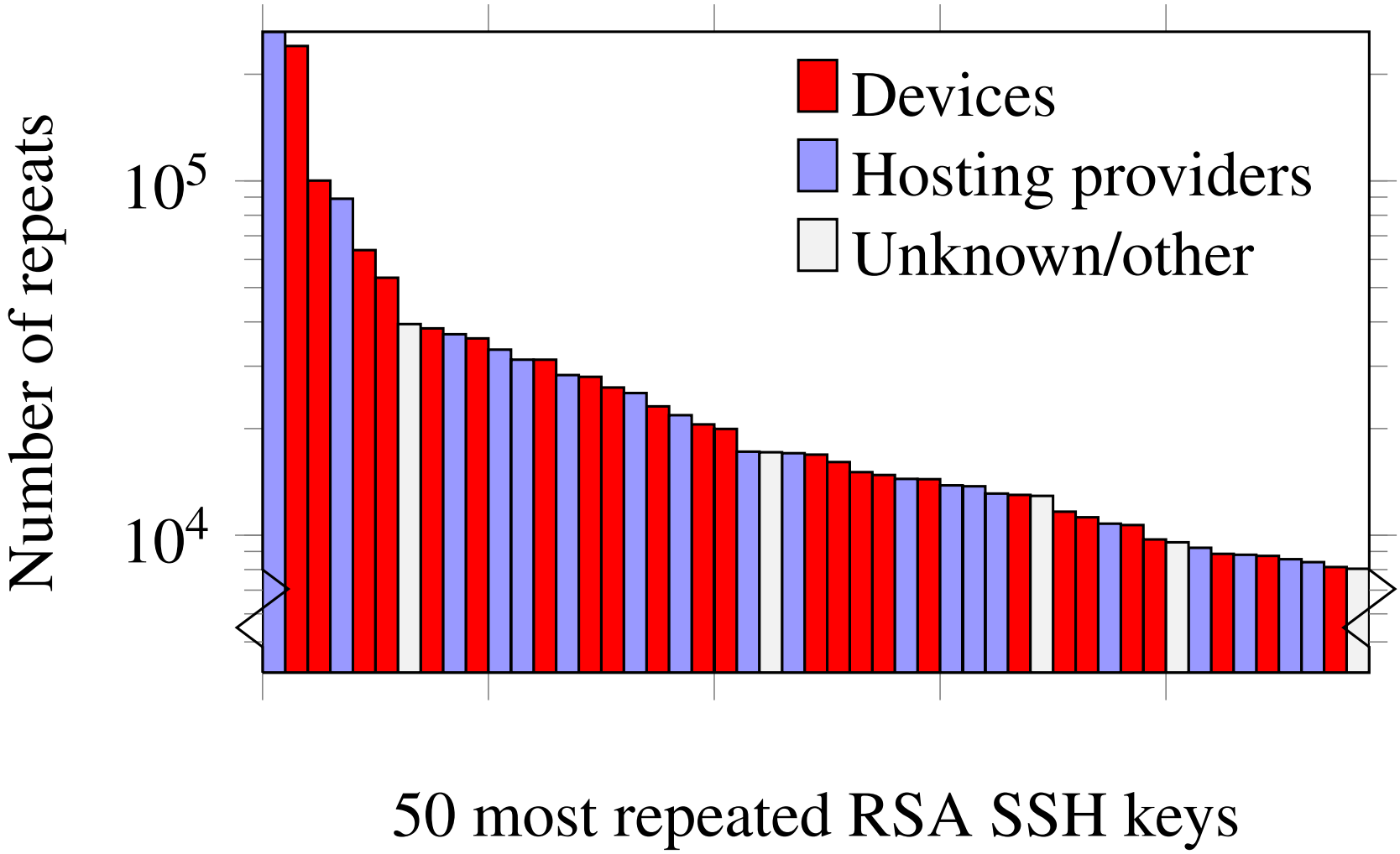
- Device default certificates/keys.
- Apparent entropy problems in key generation.

TLS:
default certificates/keys:
670,000 hosts (5%)

low-entropy repeated keys:
40,000 hosts (0.3%)

SSH:
default or low-entropy keys:
1,000,000 hosts (10%)

Looking for problems. Repeated keys



Step 2: Look for problems. Factoring RSA moduli

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

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

- Two hosts share RSA moduli with a prime factor in common $\rightarrow$ outside observer can factor both keys by calculating the GCD of public moduli.

Time to factor 768-bit RSA modulus:
two years [Kleinjung et al. 2010]

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

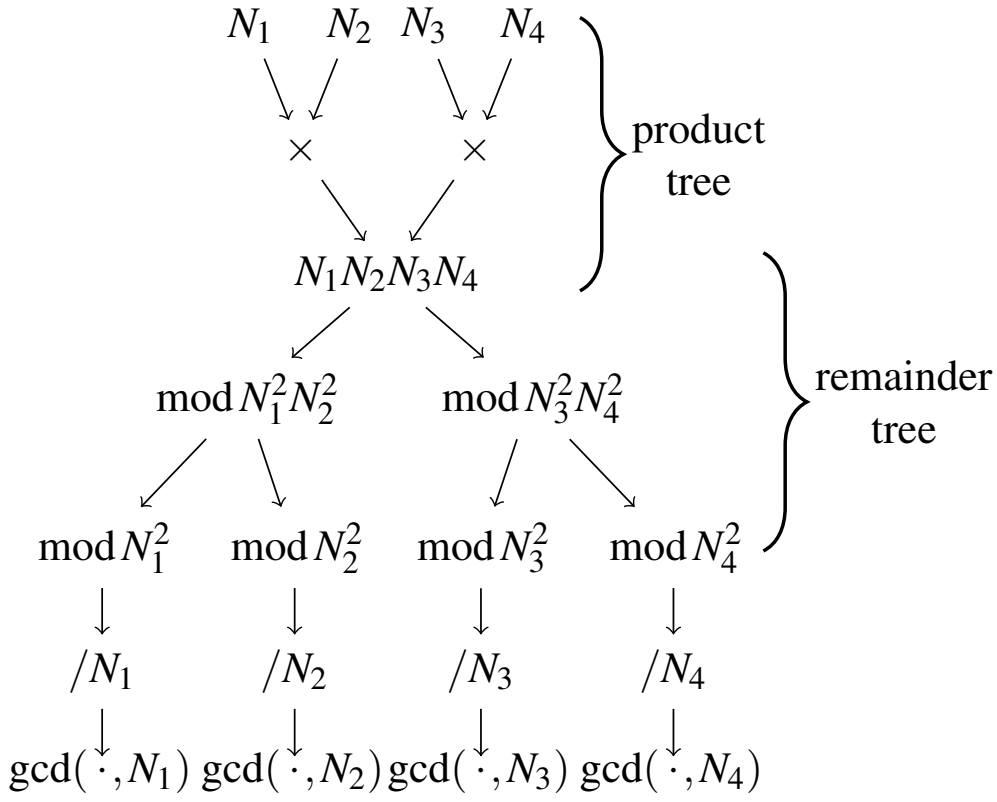
Looking for problems: Factoring RSA keys

Implemented efficient algorithm due to [Bernstein 2004].

- 350 lines of C using GMP

Ran on 11,170,883 distinct moduli in SSL, SSH, and EFF datasets

- 1.5 hours on 16 cores.
- $5 of Amazon ec2 time.



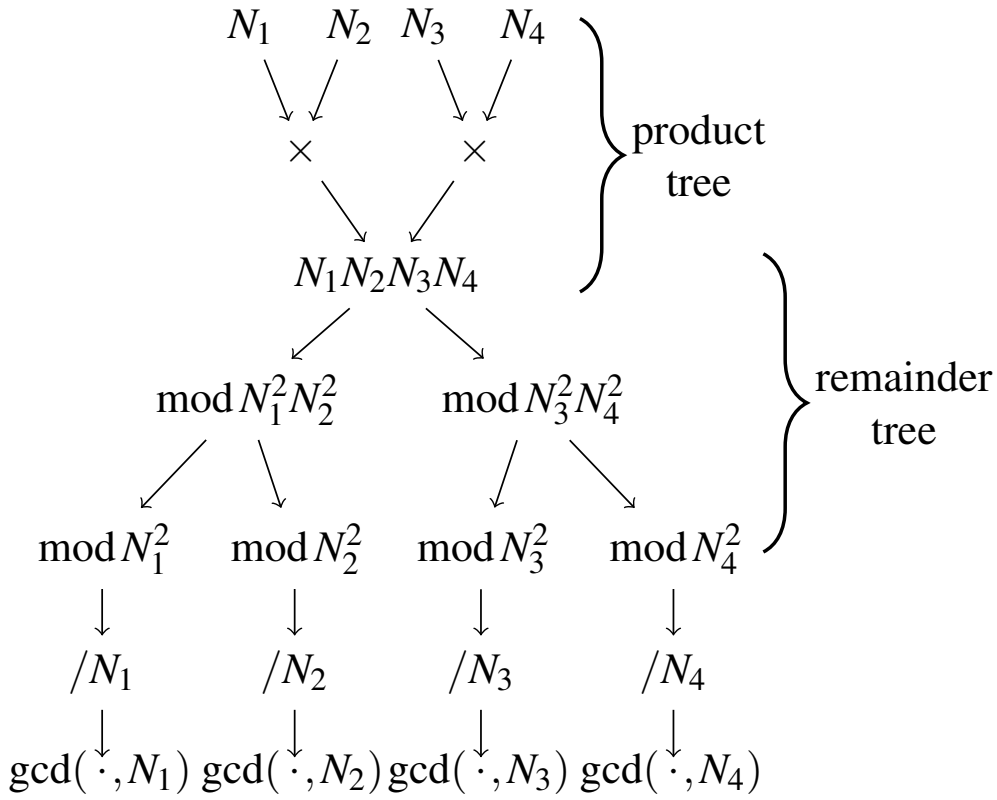
Looking for problems: Factoring RSA keys

Implemented efficient algorithm due to [Bernstein 2004].

- 350 lines of C using GMP

Ran on 11,170,883 distinct moduli in SSL, SSH, and EFF datasets

- 1.5 hours on 16 cores.
- $5 of Amazon ec2 time.



Results:

- 2,314 distinct prime factors factored 16,717 moduli
- Private keys for 64,081 TLS (0.50%) and 2,459 SSH (0.03%) hosts in our scan

Step 2: Looking for problems. Weak DSA signature nonce

DSA signatures require a random nonce.

- If the nonce is predictable
  $\rightarrow$ can easily compute long-term private key.
- Two distinct signatures use same nonce and private key
  $\rightarrow$ can easily compute nonce
  $\rightarrow$ can easily compute long-term private key.

Step 2: Looking for problems. Weak DSA signature nonce

DSA signatures require a random nonce.

- If the nonce is predictable
  $\rightarrow$ can easily compute long-term private key.
- Two distinct signatures use same nonce and private key
  $\rightarrow$ can easily compute nonce
  $\rightarrow$ can easily compute long-term private key.
- Collected 9,114,925 signatures from two scans.
- 4,365 contained the same nonce as another signature.
- Computed 281 distinct private DSA keys.
- Keys were served by 105,728 (1.03%) of SSH DSA hosts.

... only two of the factored certificates were signed by a CA, and both are expired. The web pages aren't active.

... only two of the factored certificates were signed by a CA, and both are expired. The web pages aren't active.

Look at subject information for certificates:

```
CN=self-signed, CN=system generated, CN=0168122008000024
CN=self-signed, CN=system generated, CN=0162092009003221
CN=self-signed, CN=system generated, CN=0162122008001051
C=CN, ST=Guangdong, O=TP-LINK Technologies CO., LTD., OU=TP-LINK SOFT, CN=TL-R478+1145D5C30089/emailAddres
C=CN, ST=Guangdong, O=TP-LINK Technologies CO., LTD., OU=TP-LINK SOFT, CN=TL-R478+139819C30089/emailAddres
CN=self-signed, CN=system generated, CN=0162072011000074
CN=self-signed, CN=system generated, CN=0162122009008149
CN=self-signed, CN=system generated, CN=0162122009000432
CN=self-signed, CN=system generated, CN=0162052010005821
CN=self-signed, CN=system generated, CN=0162072008005267
C=US, O=2Wire, OU=Gateway Device/serialNumber=360617088769, CN=Gateway Authentication
CN=self-signed, CN=system generated, CN=0162082009008123
CN=self-signed, CN=system generated, CN=0162072008005385
CN=self-signed, CN=system generated, CN=0162082008000317
C=CN, ST=Guangdong, O=TP-LINK Technologies CO., LTD., OU=TP-LINK SOFT, CN=TL-R478+3F5878C30089/emailAddres
CN=self-signed, CN=system generated, CN=0162072008005597
CN=self-signed, CN=system generated, CN=0162072010002630
CN=self-signed, CN=system generated, CN=0162032010008958
CN=109.235.129.114
CN=self-signed, CN=system generated, CN=0162072011004982
CN=217.92.30.85
CN=self-signed, CN=system generated, CN=0162112011000190
CN=self-signed, CN=system generated, CN=0162062008001934
CN=self-signed, CN=system generated, CN=0162112011004312
CN=self-signed, CN=system generated, CN=0162072011000946
C=US, ST=Oregon, L=Wilsonville, CN=141.213.19.107, O=Xerox Corporation, OU=Xerox Office Business Group,
CN=XRX0000AAD53FB7.eecs.umich.edu, CN=(141.213.19.107|XRX0000AAD53FB7.eecs.umich.edu)
CN=self-signed, CN=system generated, CN=0162102011001174
CN=self-signed, CN=system generated, CN=0168112011001015
CN=self-signed, CN=system generated, CN=0162012011000446
```

Attributing vulnerabilities to implementations

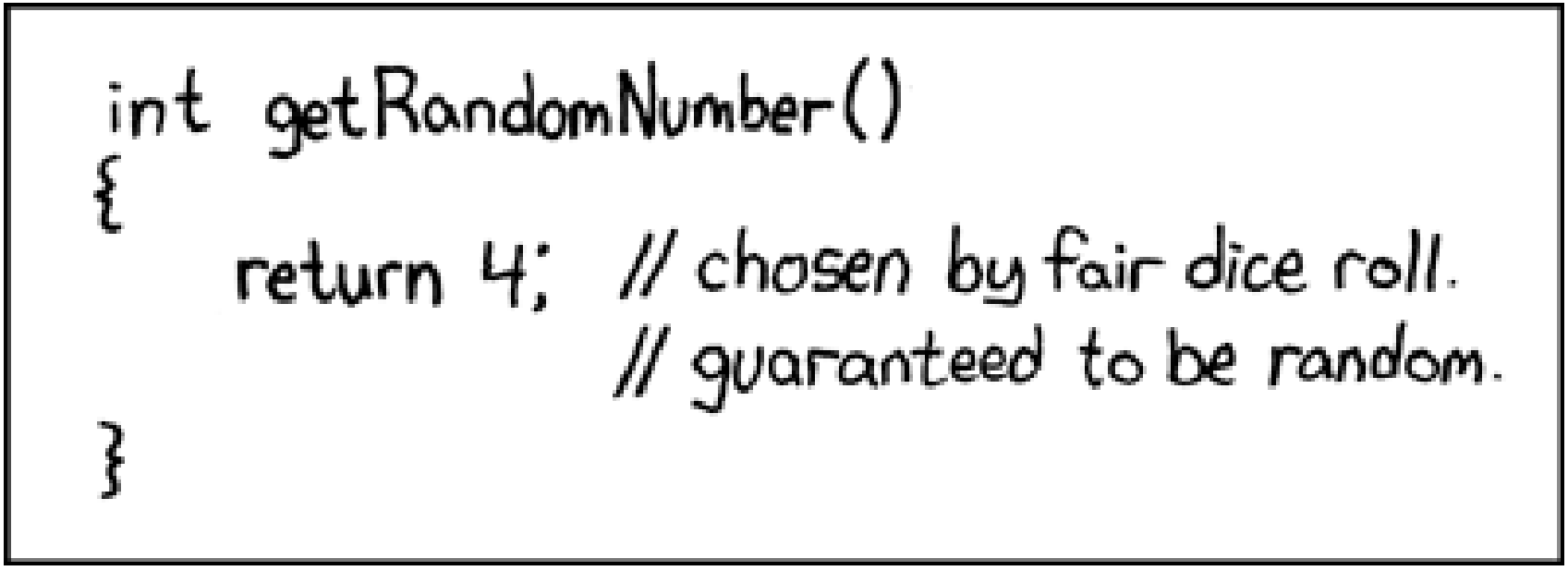
- Used information in certificate subjects, version strings, served over https or http, etc. to cluster hosts by implementation.

Vast majority of compromised keys generated by headless or embedded network devices.

- Routers, firewalls, switches, server management cards, cable modems, VOIP devices, printers, projectors...

Identified compromised devices from > 50 manufacturers.

How could this happen?



http://xkcd.com/221/

Linux: A tale of two RNGs

`/dev/random`	`/dev/urandom`
"high-quality" randomness	pseudorandomness
blocks if insufficient entropy available	never blocks

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

Linux: A tale of two RNGs

`/dev/random`

"high-quality" randomness

blocks if insufficient entropy available

`/dev/urandom`

pseudorandomness

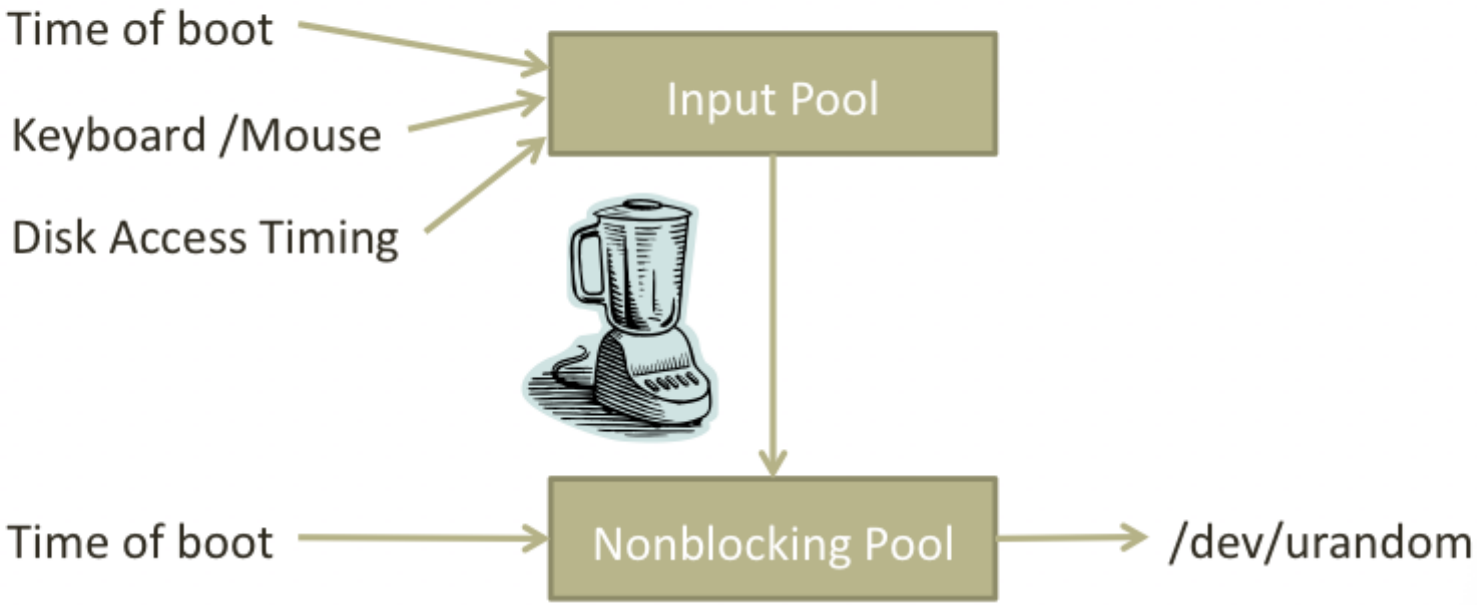
never blocks

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

```
 /* 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"
```

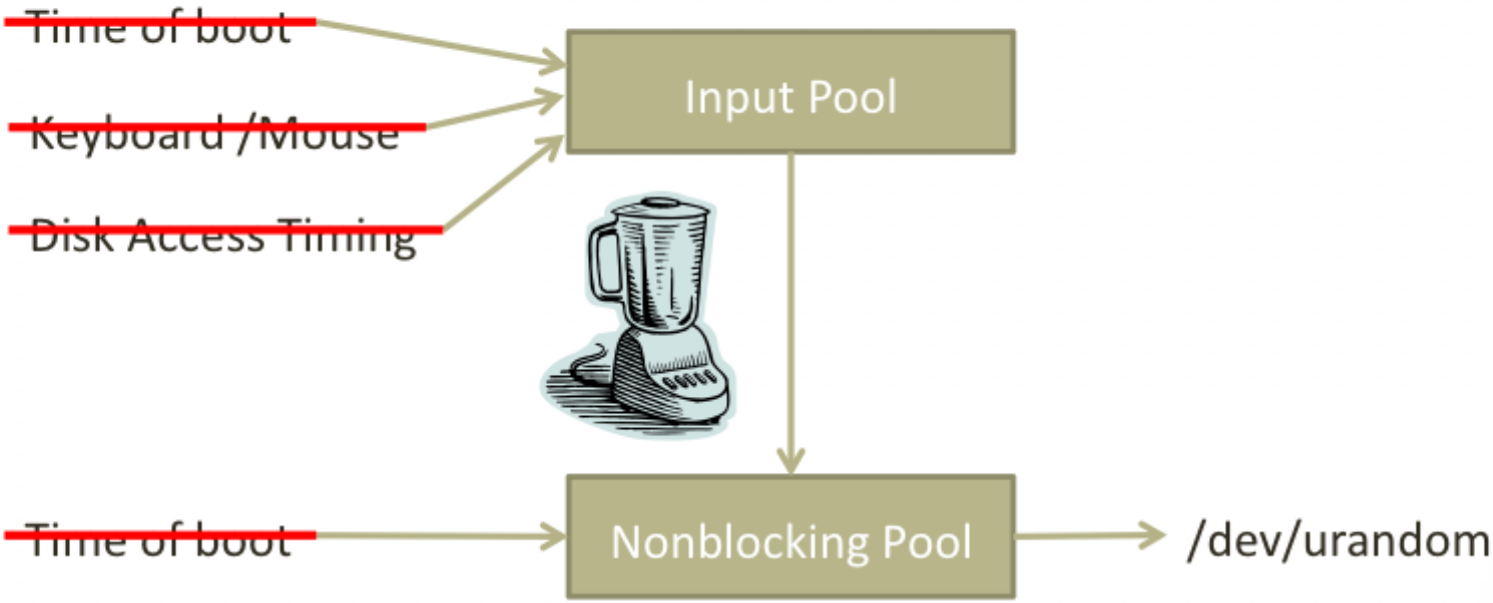
Linux /dev/urandom

Nearly everything uses /dev/urandom



Linux /dev/urandom

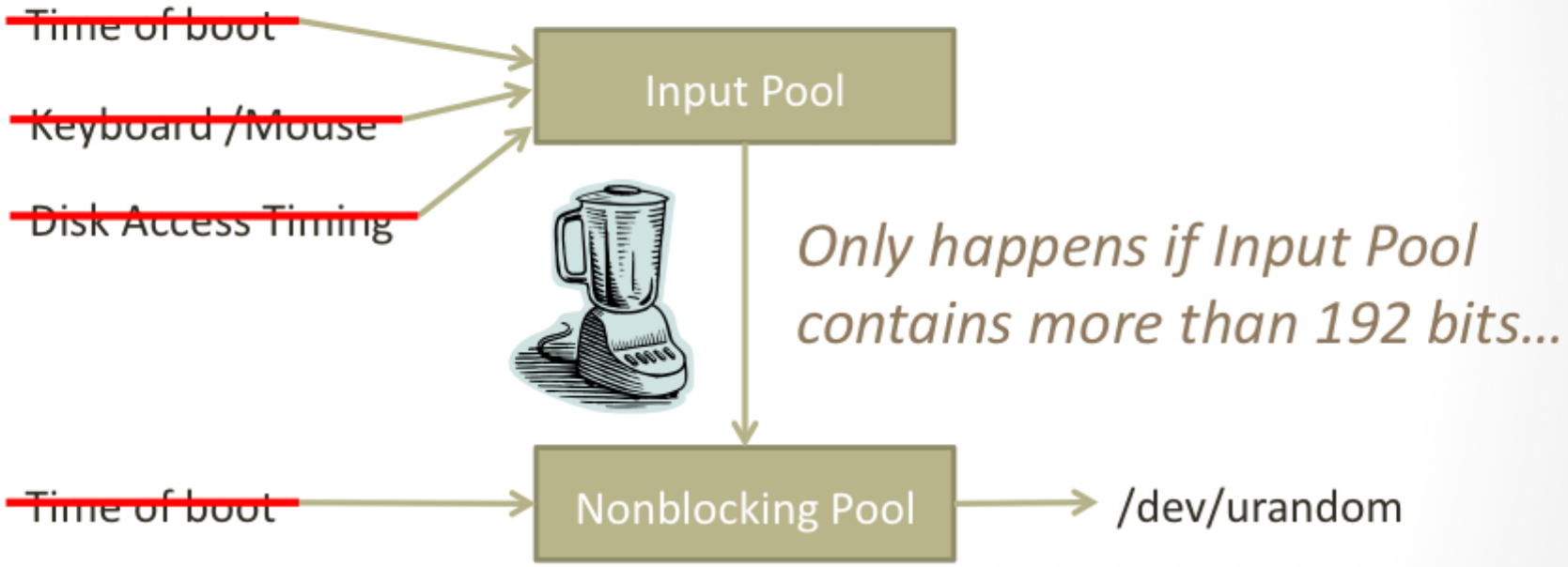
Nearly everything uses /dev/urandom



Problem 1: Embedded devices may lack all these sources

Linux /dev/urandom

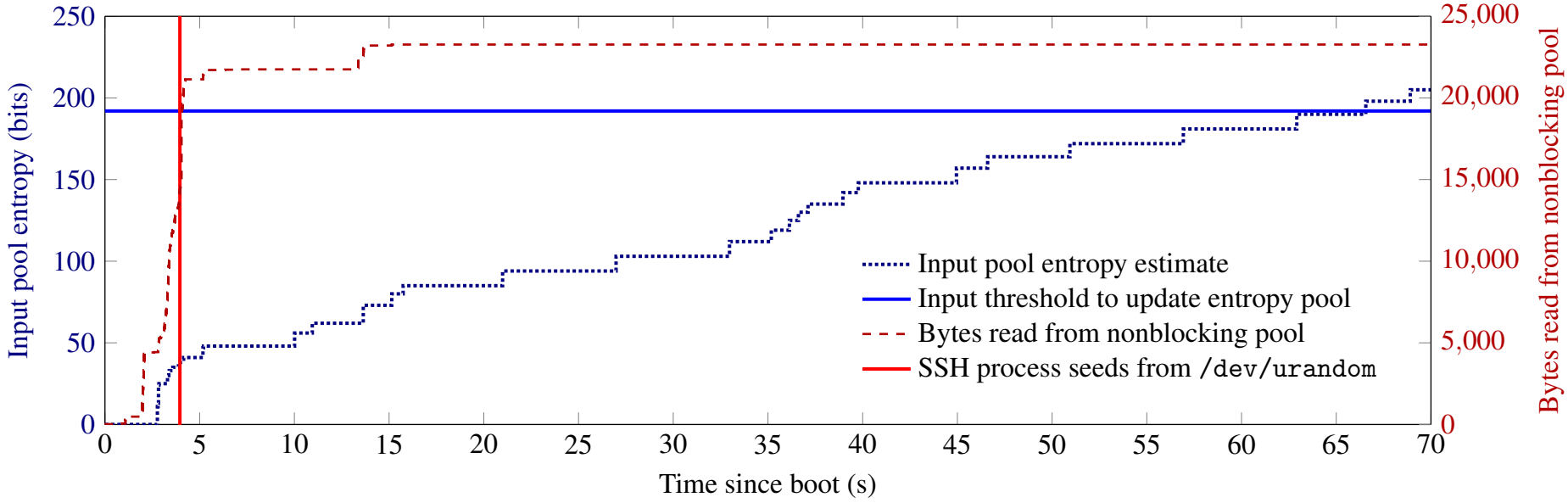
Nearly everything uses /dev/urandom



Problem 1: Embedded devices may lack all these sources

Problem 2: /dev/urandom can take a long time to "warm up"

Linux boot-time entropy hole



Generating factorable RSA keys

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

Insufficient randomness while seeding leads to repeated moduli, probably not factorable keys.

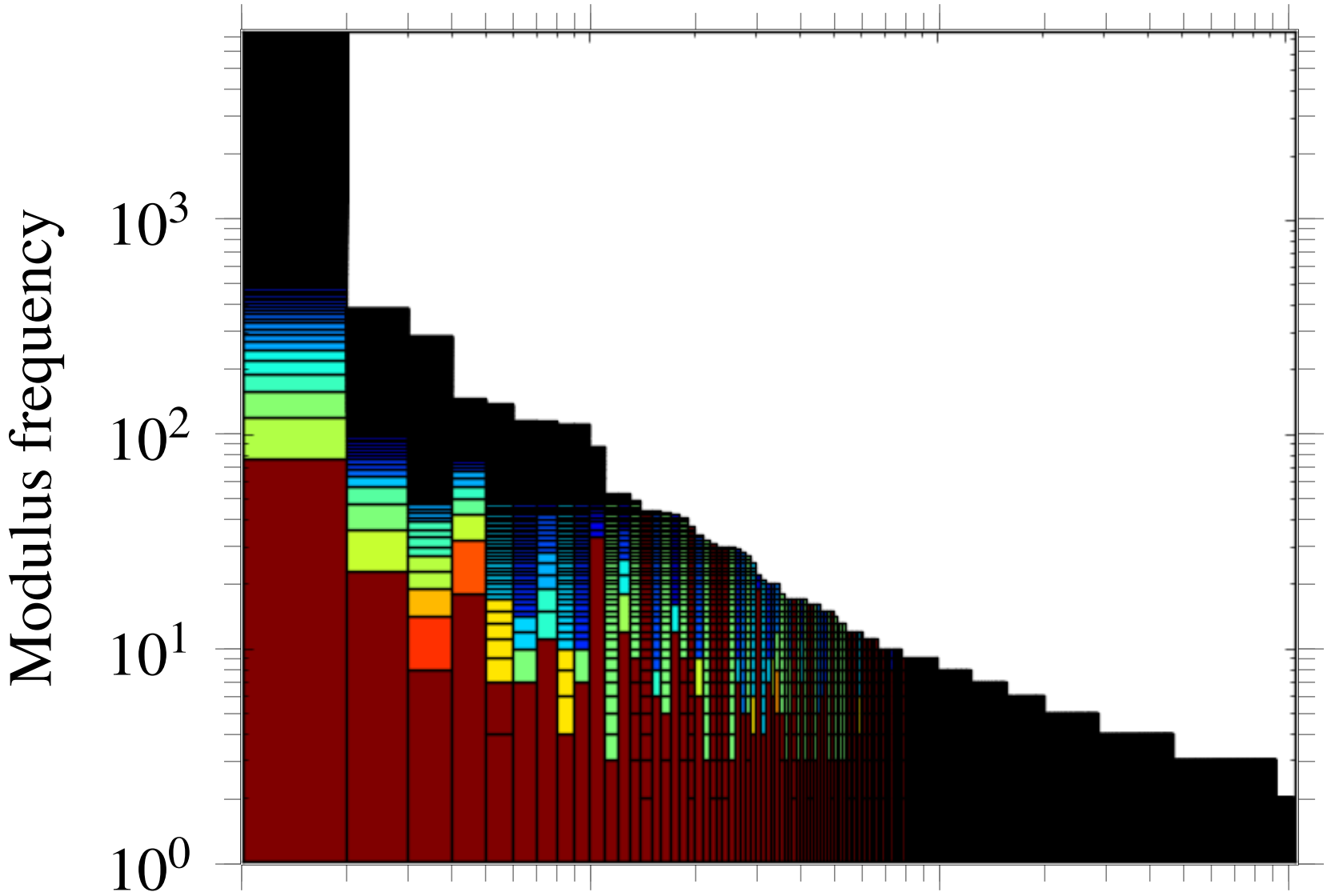
Generating factorable RSA keys

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

Insufficient randomness can lead to factorable keys.

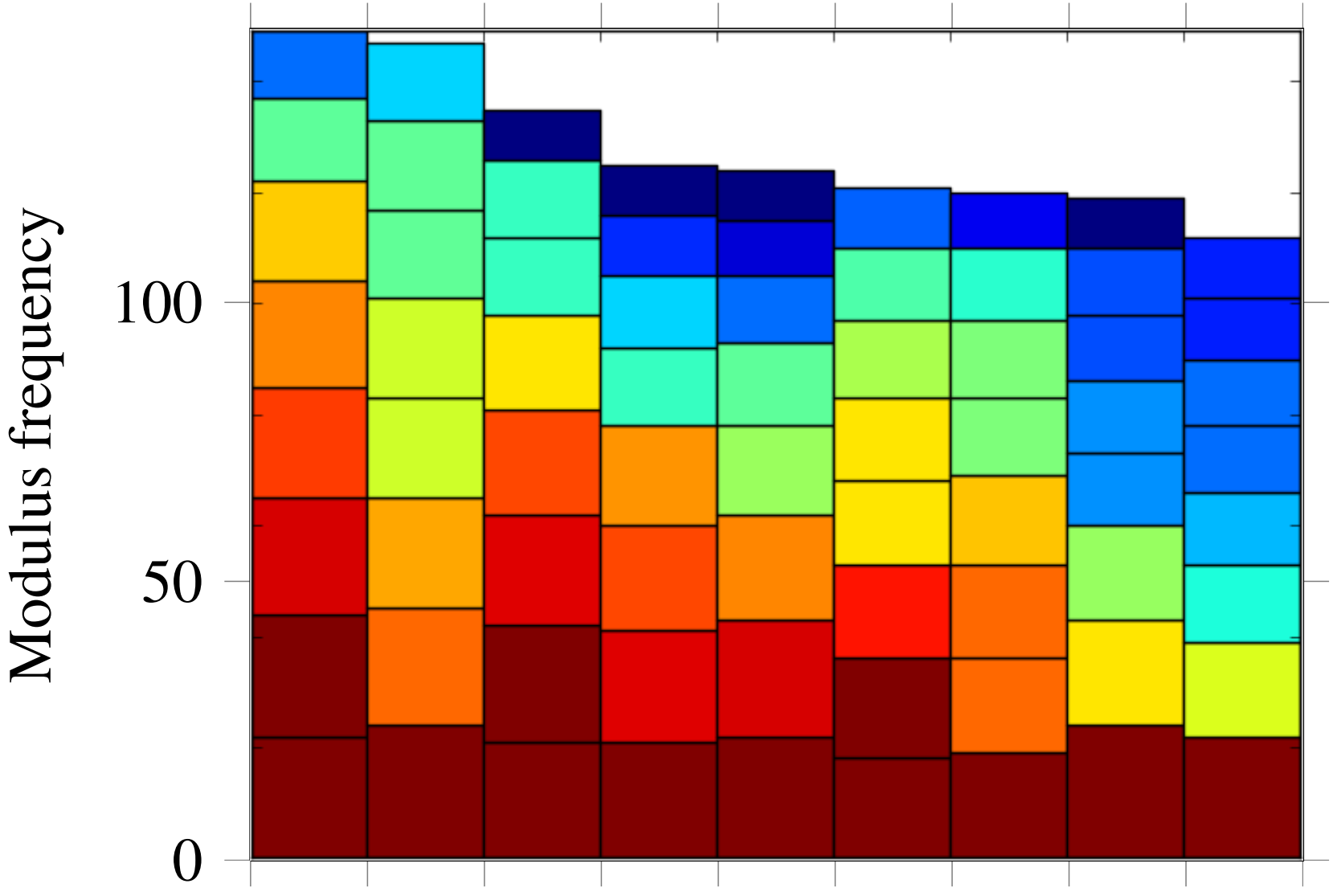
Distribution of prime factors

Juniper SRX branch devices



Distribution of prime factors

IBM Remote Supervisor Adapter II and Bladecenter Management Module



Generating weak DSA signatures

Step 1: DSA keys generated using insufficient entropy, so many hosts shared keys.

Step 2: Signature nonces generated from PRNG counter seeded once using insufficient entropy and never updated.

Two counters in same state $\rightarrow$ keys revealed.

Disclosure

- Wrote disclosures to 61 companies.
- 13 had Security Incident Response Team contact information available.
- Have gotten responses from 28.
- 13 have told us they have fixed the problem
- 5 have informed us of security advisories
- Coordinating through US-CERT, ICS CERT, JP-CERT

Linux kernel has been patched.

This is just the tip of the iceberg

More examples of bad randomness!

This is just the tip of the iceberg

More examples of bad randomness!

- PGP database. [Lenstra et al. 2012]
  2 factored RSA keys out of 700,000. Why?

This is just the tip of the iceberg

More examples of bad randomness!

- PGP database. [Lenstra et al. 2012]
  2 factored RSA keys out of 700,000. Why?
- Smartcards. [2012 Chou (slides in Chinese)]
  Taiwan Citizen Digital Certificates smartcard certificates used for paying taxes, etc.
  Factored 103 (out of 2.26 million)

This is just the tip of the iceberg

More examples of bad randomness!

- PGP database. [Lenstra et al. 2012]
  2 factored RSA keys out of 700,000. Why?
- Smartcards. [2012 Chou (slides in Chinese)]
  Taiwan Citizen Digital Certificates smartcard certificates used for paying taxes, etc.
  Factored 103 (out of 2.26 million)
- On traditional PCs, margin of safety for keys generated on first boot is slim
  Not observed to be exploitable (so far)

Practical mitigations

Developers and manufacturers:

- Collect entropy more aggressively, add hardware sources
- Provide seed in factory
- Run for a while before generating cryptographic keys.

CAs:

- Test for repeated, factorable, and other weak keys.

Users:

- Check against known weak keys.
- Regenerate keys.

Thoughts for cryptographers

How is cryptography even possible?

- On the academic side: 'hedged' cryptography [Bellare et al. 2009]
- On the practical side: Design resilient standards. (e.g. DSA nonce deterministically generated from message, salt.)

Thoughts for cryptographers

How is cryptography even possible?

- On the academic side: 'hedged' cryptography [Bellare et al. 2009]
- On the practical side: Design resilient standards. (e.g. DSA nonce deterministically generated from message, salt.)

Widespread lack of understanding of what randomness *is*, pseudorandomness, realistic attack models...

Thoughts for system designers

- Cryptographic entropy is very hard to get right.
- Problems involve interactions between many layers of abstraction: hardware, OS, application.
- Some of these problems should have been found during testing.
- Some might only be detectable with a large sample.

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

https://factorable.net