Course Announcements

- Project
  - Project due Wednesday 4/22
  - Send a private Piazza post to the TA/grader overseeing your project
- Notes
  - Reading: End-to-End Verifiability
  - Remaining lectures: hash functions + cryptanalysis next week, then guest lectures on cryptography and the law for the final week

Lecture 22: Random Number Generation

1. Defining ‘good’ randomness

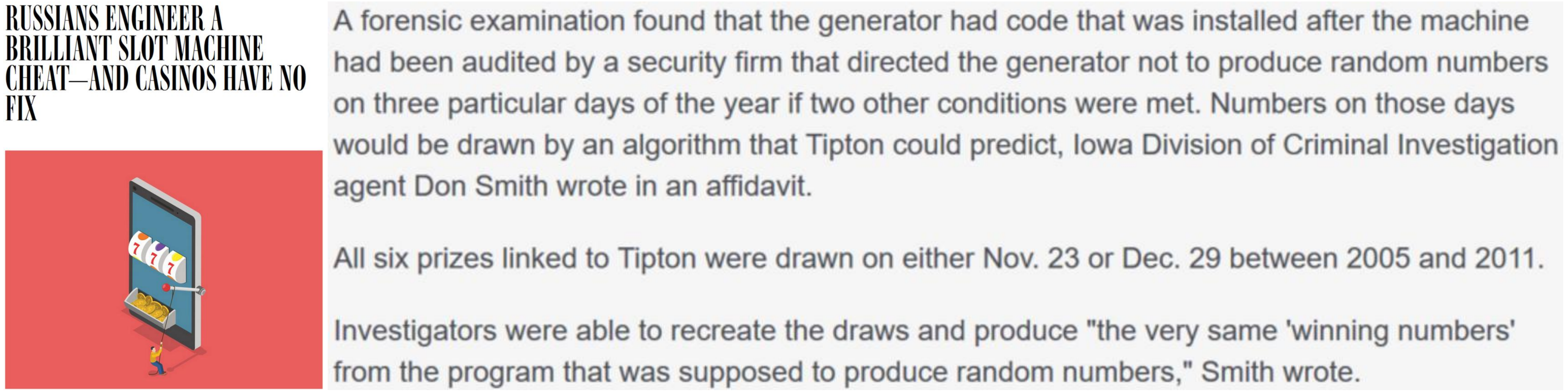
2. Constructing RNGs: harvest, extract, expand

Randomness ⇒ Unpredictability ⇒ Secrecy

1. Defining ‘good’ randomness

Effects of bad randomness

- Lottery fraud



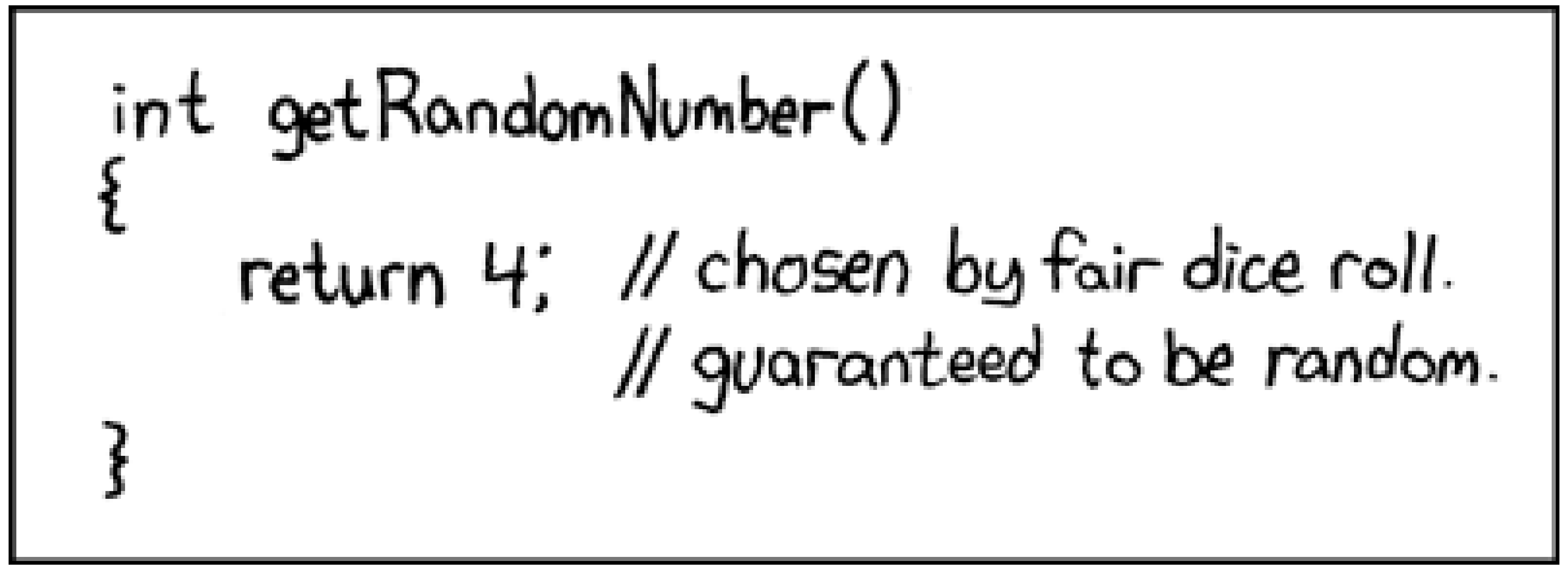
RUSSIANS ENGINEER A BRILLIANT SLOT MACHINE CHEAT—AND CASINOS HAVE NO FIX

A forensic examination found that the generator had code that was installed after the machine had been audited by a security firm that directed the generator not to produce random numbers on three particular days of the year if two other conditions were met. Numbers on those days would be drawn by an algorithm that Tipton could predict, Iowa Division of Criminal Investigation agent Don Smith wrote in an affidavit.

All six prizes linked to Tipton were drawn on either Nov. 23 or Dec. 29 between 2005 and 2011.

Investigators were able to recreate the draws and produce "the very same 'winning numbers' from the program that was supposed to produce random numbers," Smith wrote.

- Weak TLS keys on Debian computers in 2006-2008
- Weak RSA keys
- …and more



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Source: xkcd.com/221

Bad randomness in Debian

Bug forum discussion, 2003

I'm using Valgrind to debug a program that uses the OpenSSL libraries, and got warnings about uninitialized data in the function RSA_padding_add_PKCS1_type_2(), on the line with "} while (*p == '\0');" (line 171 in version 0.9.7a). The following patch ensures that the data is always modified, something that the bytes() method obviously fails to do.

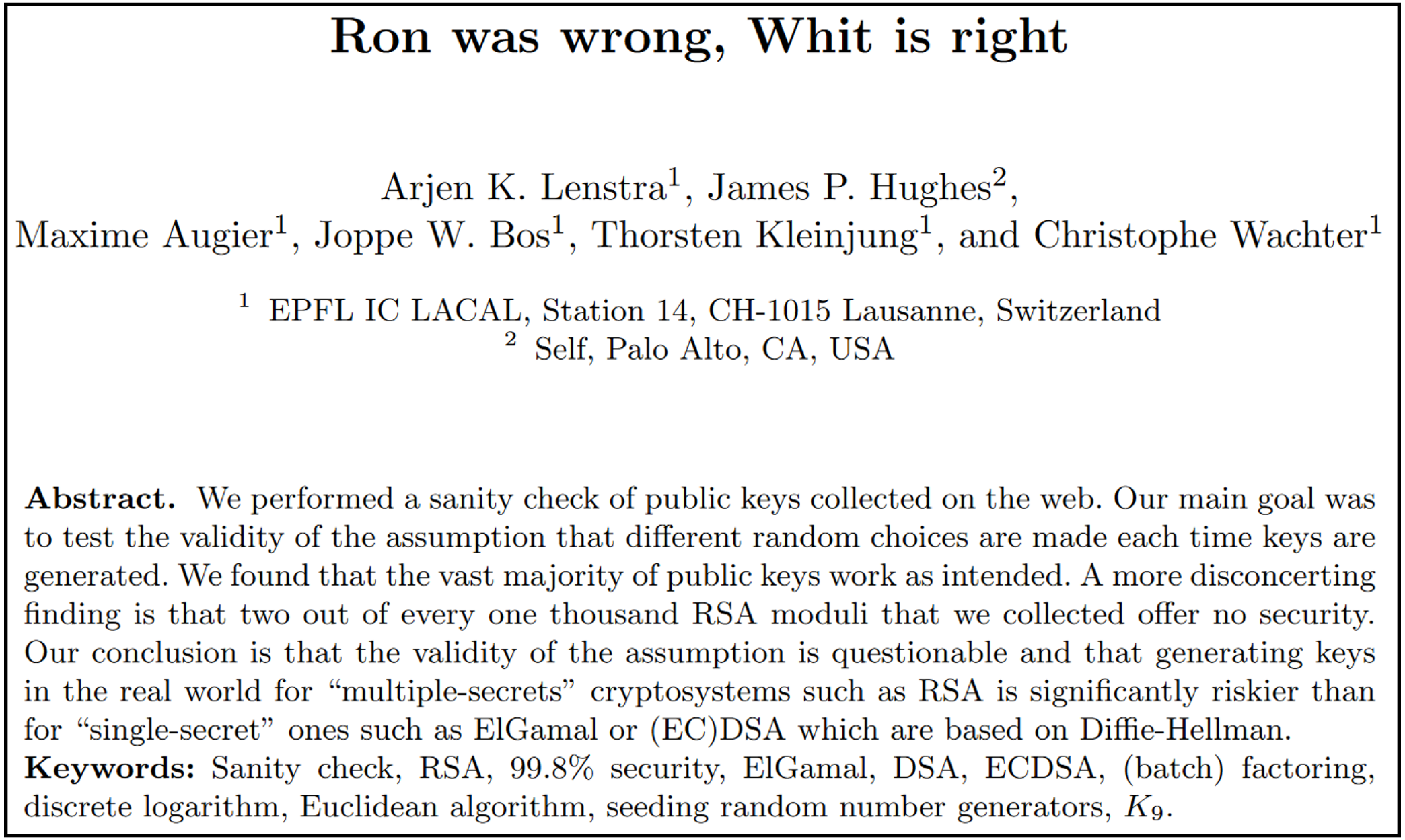
```
--- rand_lib.c Thu Jan 30 2003
+++ rand_lib.c Wed Feb 26 2003
@@ -154,6 +154,7 @@
int RAND_bytes(unsigned char *buf, int num)
{
[new code here]
```

Debian security advisory, 2008

Luciano Bello discovered that the random number generator in Debian's openssl package is predictable. This is caused by an incorrect Debian-specific change to the openssl package. As a result, cryptographic key material may be guessable. ...

It is strongly recommended that all cryptographic key material which has been generated by OpenSSL versions starting with 0.9.8c-1 on Debian systems is recreated from scratch. Furthermore, all DSA keys ever used on affected Debian systems for signing or authentication purposes should be considered compromised.

Bad randomness in RSA key generation



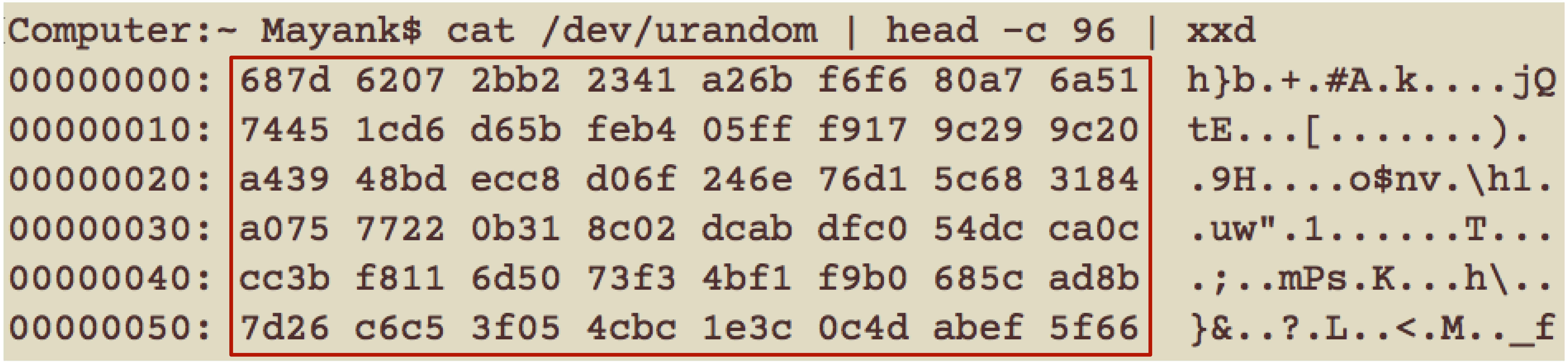
Ron was wrong, Whit is right

Arjen K. Lenstra[1], James P. Hughes[2],
Maxime Augier[1], Joppe W. Bos[1], Thorsten Kleinjung[1], and Christophe Wachter[1]

[1] EPFL IC LACAL, Station 14, CH-1015 Lausanne, Switzerland
[2] Self, Palo Alto, CA, USA

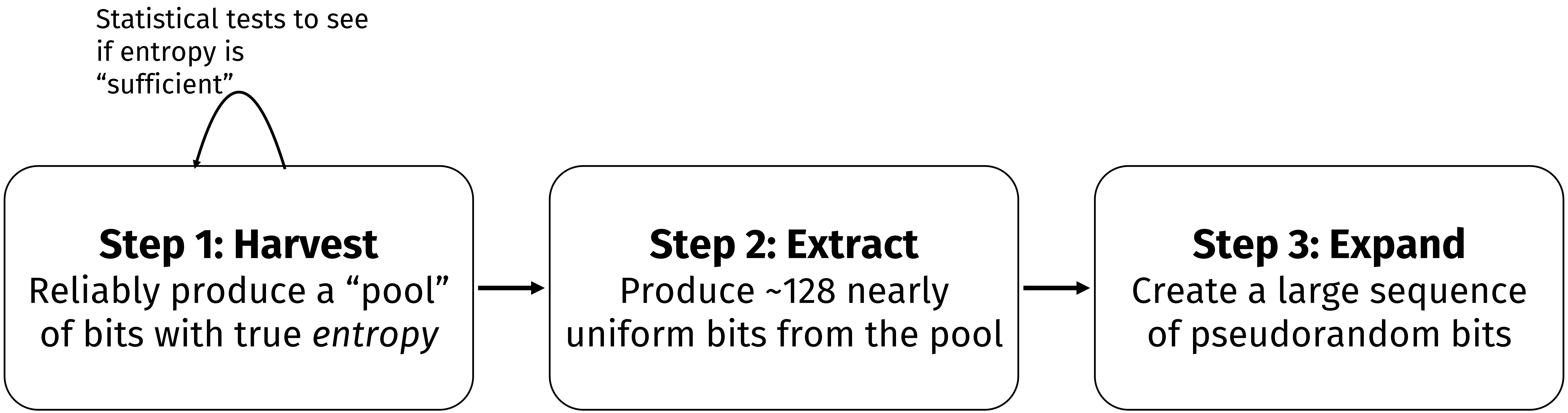
Abstract. We performed a sanity check of public keys collected on the web. Our main goal was to test the validity of the assumption that different random choices are made each time keys are generated. We found that the vast majority of public keys work as intended. A more disconcerting finding is that two out of every one thousand RSA moduli that we collected offer no security. Our conclusion is that the validity of the assumption is questionable and that generating keys in the real world for "multiple-secrets" cryptosystems such as RSA is significantly riskier than for "single-secret" ones such as ElGamal or (EC)DSA which are based on Diffie-Hellman.
Keywords: Sanity check, RSA, 99.8% security, ElGamal, DSA, ECDSA, (batch) factoring, discrete logarithm, Euclidean algorithm, seeding random number generators, K_9.

How you obtain randomness: /dev/urandom



```
Computer:~ Mayank$ cat /dev/urandom | head -c 96 | xxd
00000000: 687d 6207 2bb2 2341 a26b f6f6 80a7 6a51  h}b.+.#A.k....jQ
00000010: 7445 1cd6 d65b feb4 05ff f917 9c29 9c20  tE...[.......).
00000020: a439 48bd ecc8 d06f 246e 76d1 5c68 3184  .9H....o$nv.\h1.
00000030: a075 7722 0b31 8c02 dcab dfc0 54dc ca0c  .uw".1......T...
00000040: cc3b f811 6d50 73f3 4bf1 f9b0 685c ad8b  .;..mPs.K...h\..
00000050: 7d26 c6c5 3f05 4cbc 1e3c 0c4d abef 5f66  }&..?.L..<.M.._f
```

How a computer generates randomness



Statistical tests to see if entropy is "sufficient"

Step 1: Harvest
Reliably produce a "pool" of bits with true *entropy*

Step 2: Extract
Produce ~128 nearly uniform bits from the pool

Step 3: Expand
Create a large sequence of pseudorandom bits

Security requirements of randomness generation

1. *Performance:* Be fast enough that people will use it
2. *Hard fail:* Only expand once the system has been adequately seeded with true entropy
3. *Resilience:* Adversary can't predict outputs, even if she can partially influence the source of true randomness ⇨ Use multiple sources of entropy, and combine them in a smart way
4. *Forward + backward secrecy:* Adversary cannot predict past or future PRNG outputs even if she knows the current seed and state ⇨ Re-seed the PRNG periodically with new truly random numbers

2. Constructing RNGs: Harvest, Extract, Expand

Step 1: Harvest
Reliably produce a “pool” of bits with true *entropy*

→

Step 2: Extract
Produce ~128 nearly uniform bits from the pool

→

Step 3: Expand
Create a large sequence of pseudorandom bits

“Fortunately, it’s not hard to harvest truly unpredictable randomness by tapping the chaotic universe that surrounds a computer's orderly, deterministic world of 1s and 0s.”

– *IEEE Spectrum*

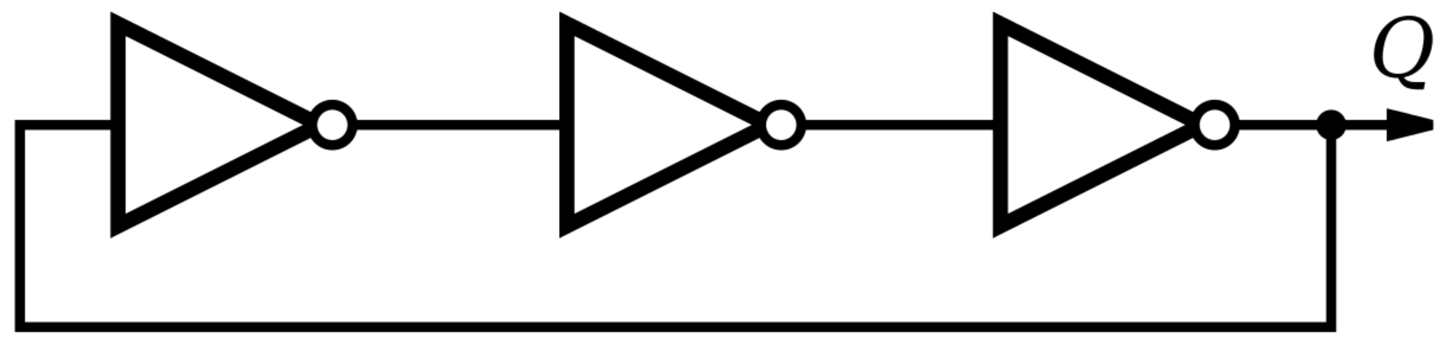
Step 1: Sources of entropy to harvest

- Physics: EM radiation, temperature (random.org/history)

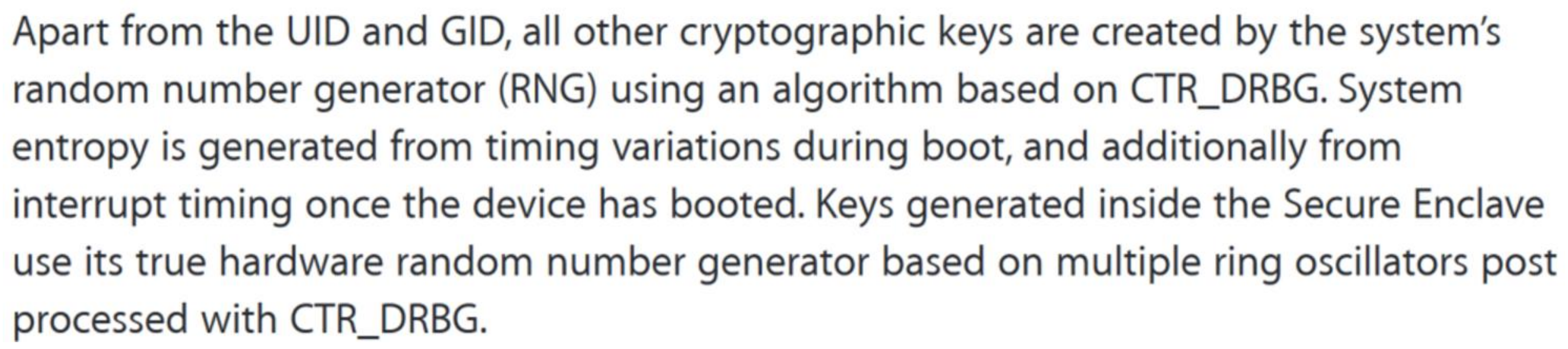
SONY
PLEASE DON'T TOUCH THESE RADIOS!
(they're very carefully calibrated)
Thanks!
—Mads
SIEMENS
SCENIC 300

Step 1: Sources of entropy to harvest

- Physics: EM radiation, temperature (random.org/history)
- Logical gates: Clock drift, thermal noise

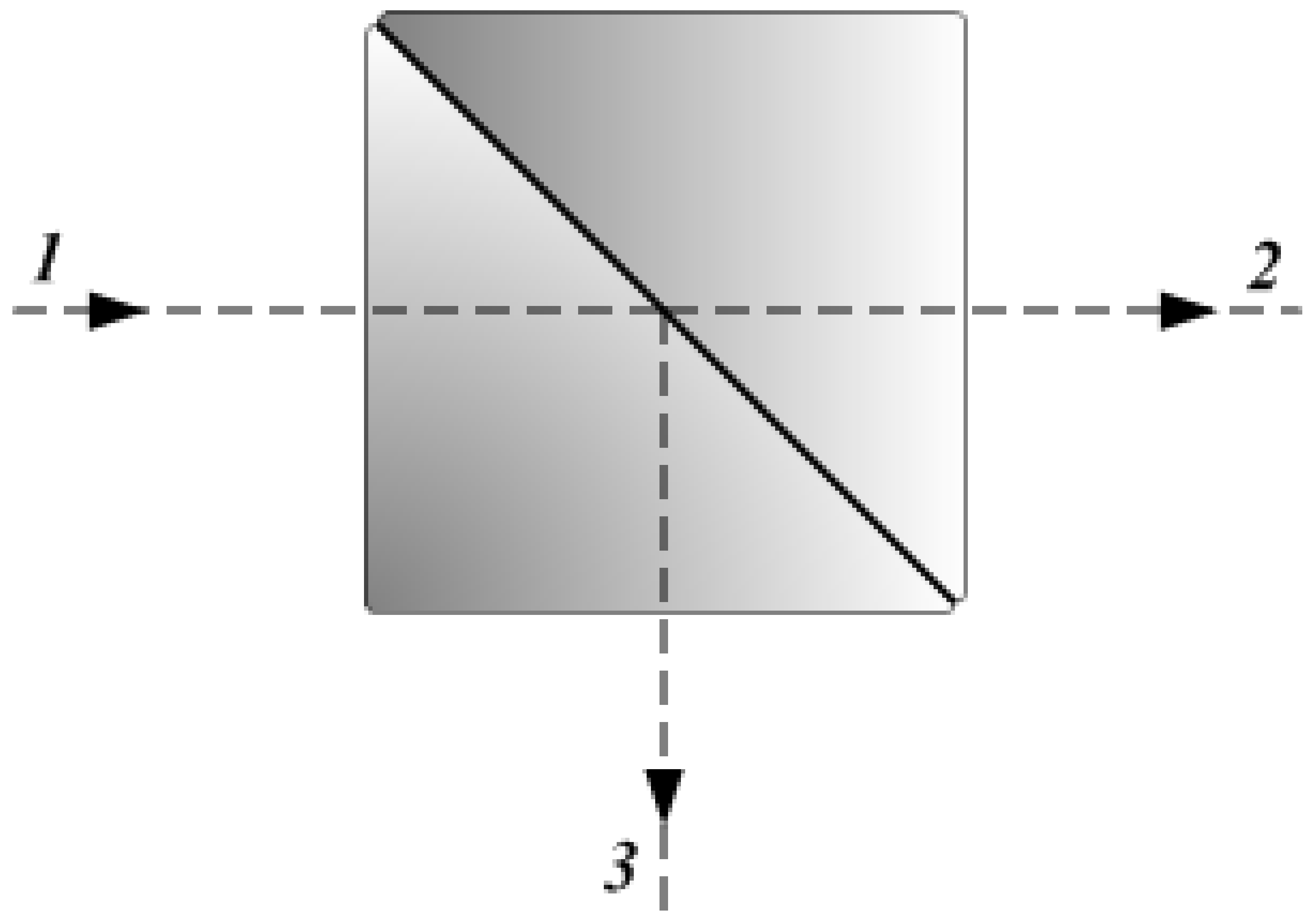


Apple's Secure Enclave


Apart from the UID and GID, all other cryptographic keys are created by the system's random number generator (RNG) using an algorithm based on CTR_DRBG. System entropy is generated from timing variations during boot, and additionally from interrupt timing once the device has booted. Keys generated inside the Secure Enclave use its true hardware random number generator based on multiple ring oscillators post processed with CTR_DRBG.

Step 1: Sources of entropy to harvest

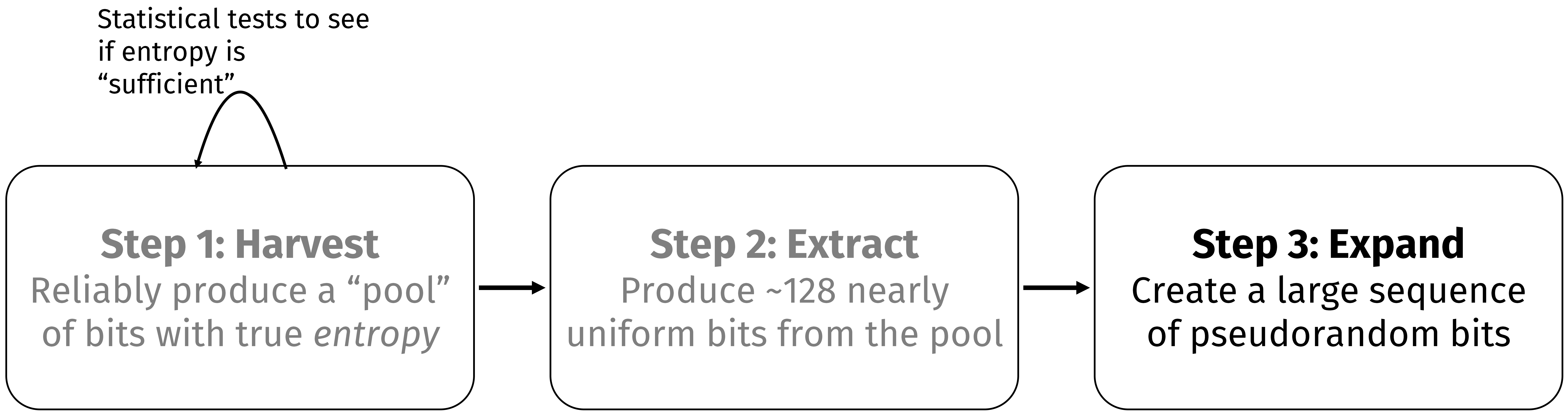
- Physics: EM radiation, temperature (random.org/history)
- Logical gates: Clock drift, thermal noise
- Quantumness: beam splitters & polarization, tunneling, entanglement



Step 1: Sources of entropy to harvest

- Physics: EM radiation, temperature (random.org/history)
- Logical gates: Clock drift, thermal noise
- Quantumness: beam splitters & polarization, tunneling, entanglement
- Human: keystroke timings, mouse movements, hard drive seek times
- Sensors: microphone, camera, gyroscope, Bluetooth/GPS/wifi signal

Step 3: Pseudorandom expansion


Statistical tests to see if entropy is "sufficient"
Step 1: Harvest
Reliably produce a "pool" of bits with true *entropy*
Step 2: Extract
Produce ~128 nearly uniform bits from the pool
Step 3: Expand
Create a large sequence of pseudorandom bits

Step 3: NIST standards for DRBGs

- Use counter mode as a stream cipher (CTR_DBRG)

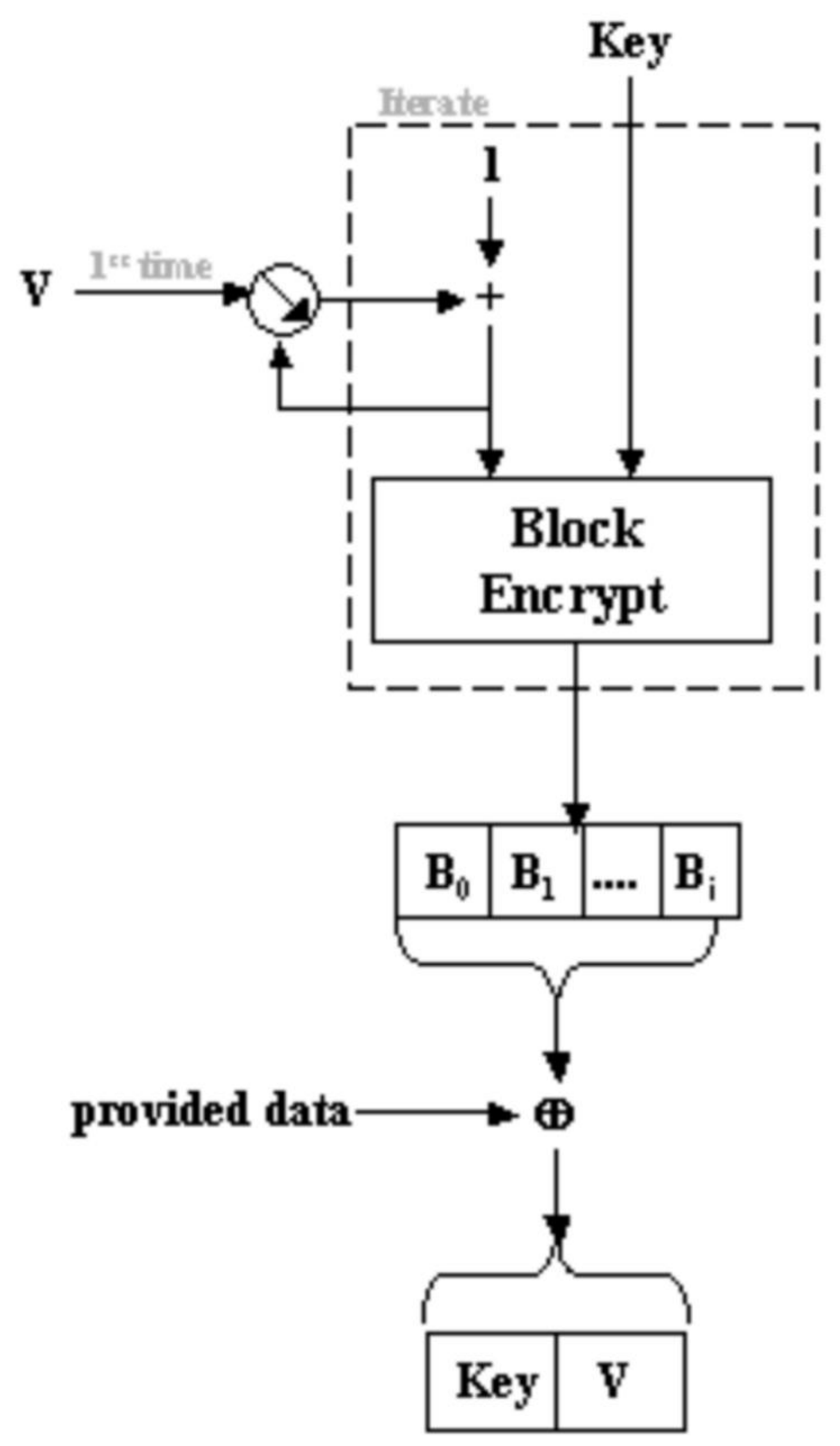
Short *K*

+

Nonce *N*

⇒

Synthetic, one time use key *K'*

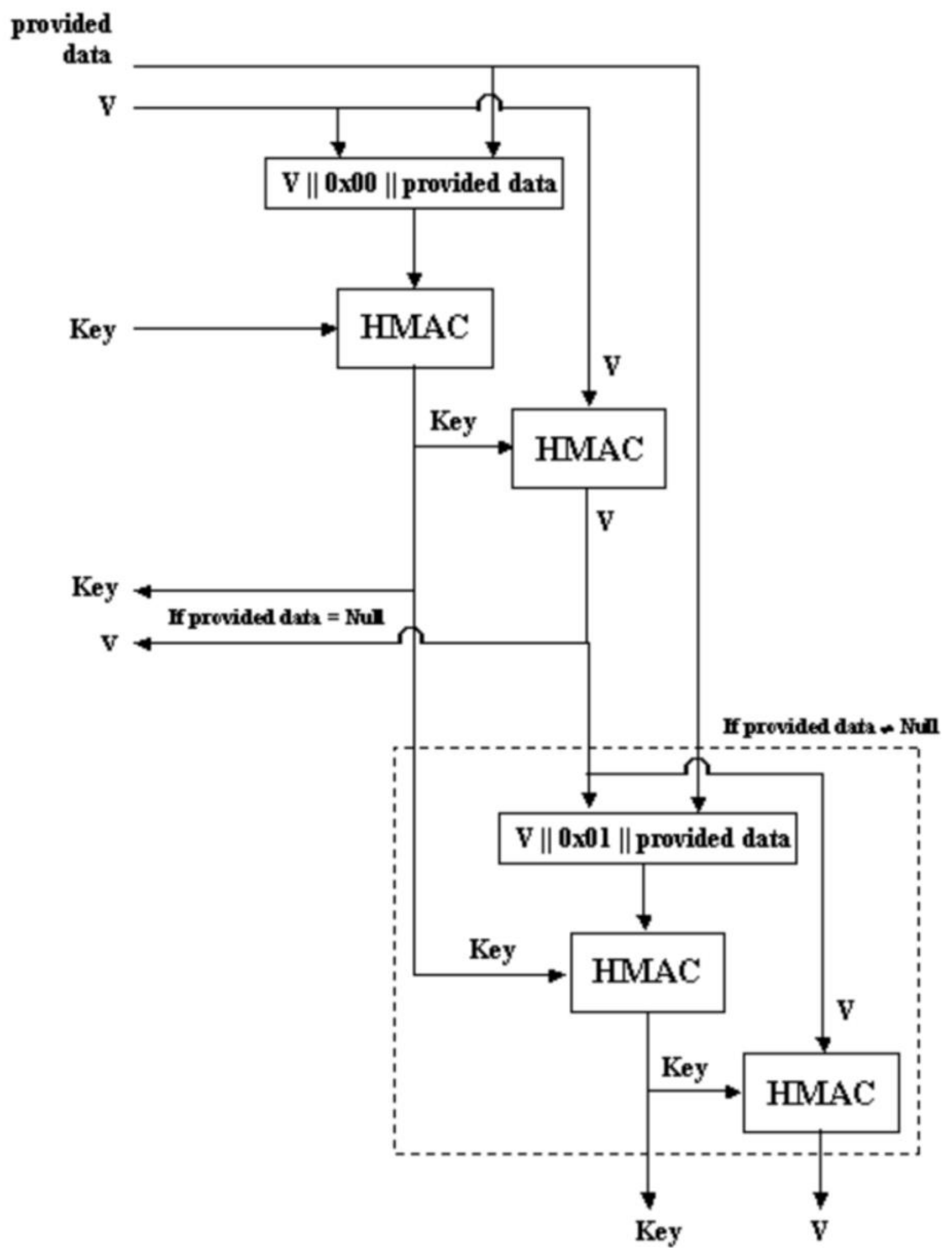


<u>Source:</u> NIST Special Publication 800-90A

Recommendation for Random Number Generation Using Deterministic Random Bit Generators

Step 3: NIST standards for DRBGs

- Use counter mode as a stream cipher (CTR_DRBG)
- A MAC is pseudorandom (HMAC_DRBG)

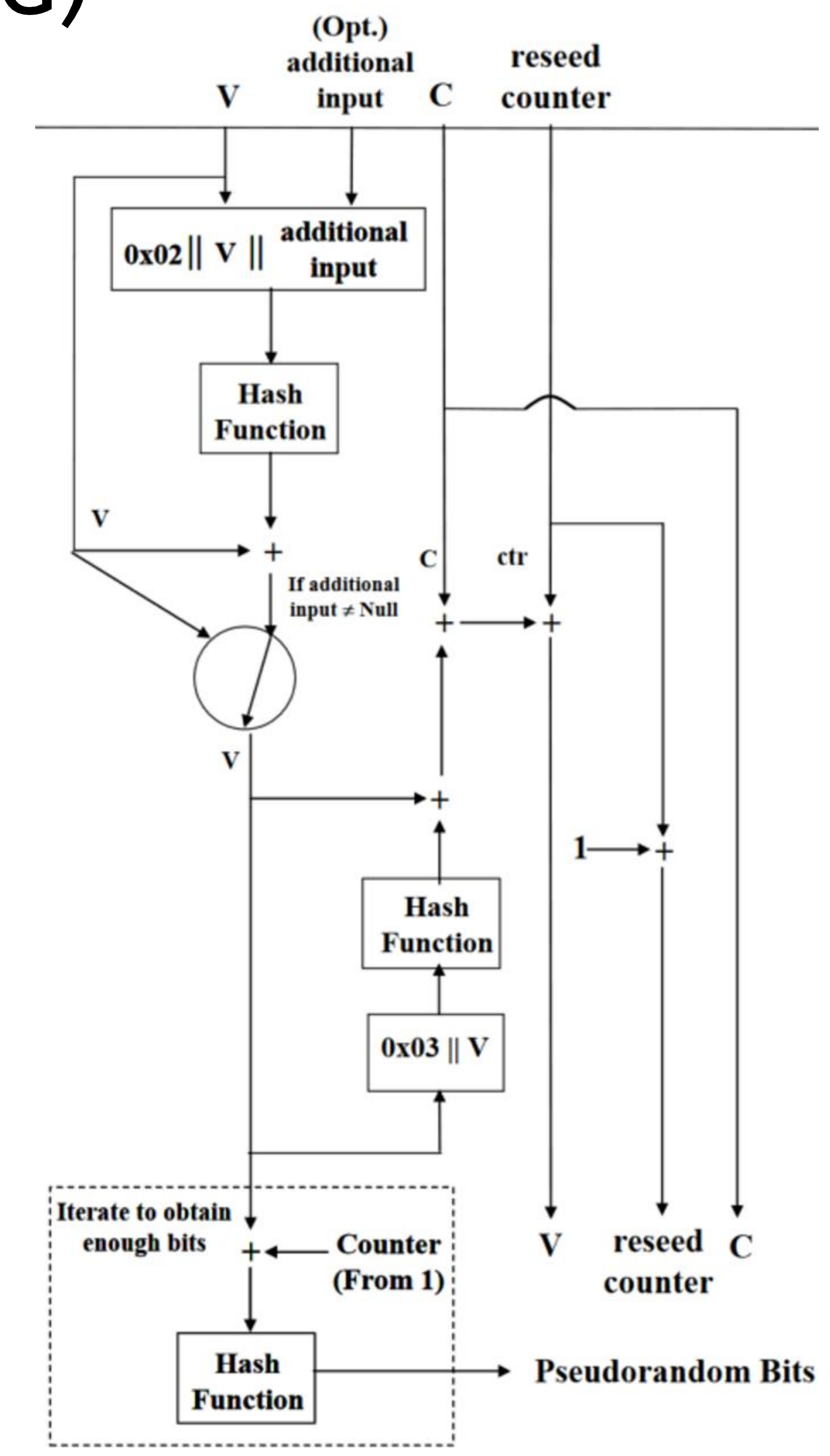


Source: NIST Special Publication 800-90A

Recommendation for Random Number Generation Using Deterministic Random Bit Generators

Step 3: NIST standards for DRBGs

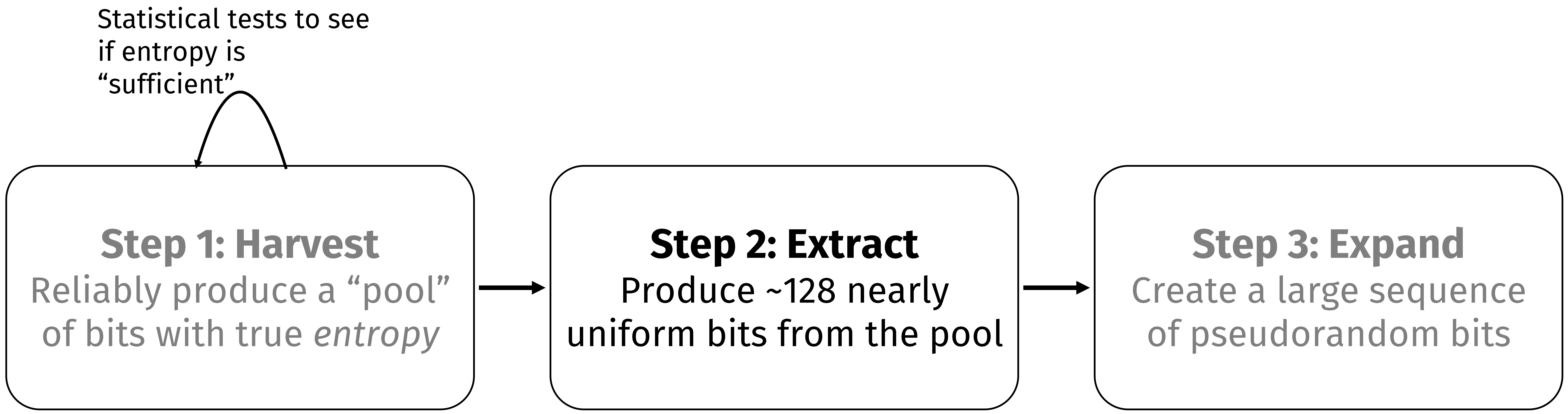
- Use counter mode as a stream cipher (CTR_DRBG)
- A MAC is pseudorandom (HMAC_DRBG)
- Use a hash function (Hash_DRBG)



Source: NIST Special Publication 800-90A

Recommendation for Random Number Generation
Using Deterministic Random Bit Generators

Step 2: Extraction of uniform-looking bits



Statistical tests to see if entropy is “sufficient”

Step 1: Harvest
Reliably produce a “pool” of bits with true *entropy*

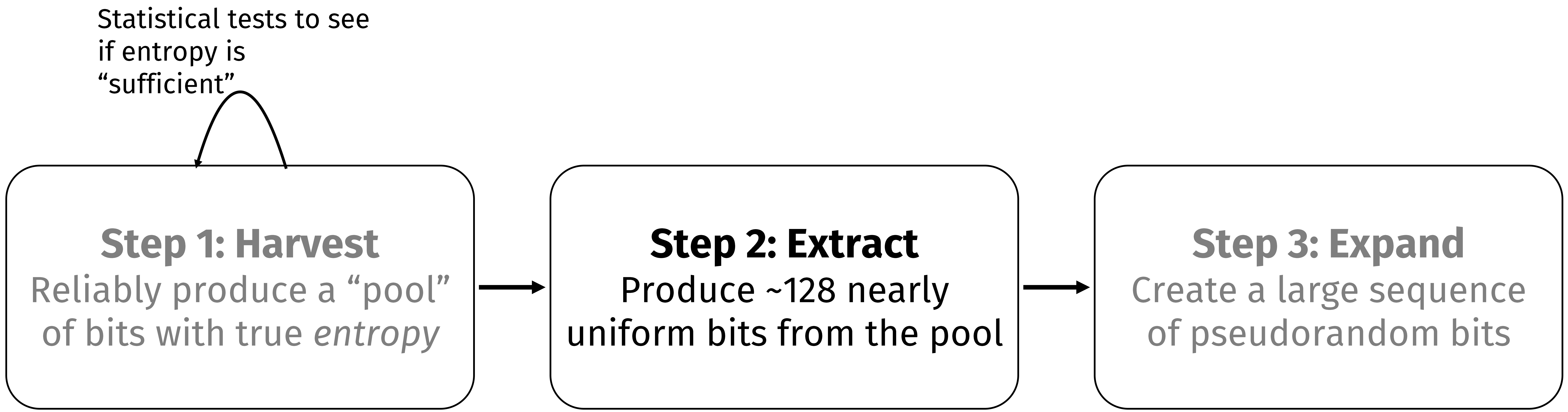
Step 2: Extract
Produce ~128 nearly uniform bits from the pool

Step 3: Expand
Create a large sequence of pseudorandom bits

Hashing as an extractor?

- Let’s try to use a hash function H as an extractor (spoiler: it won’t work)
- Extractors operate on the principle that including more entropy sources can't hurt: H(x,y,z) is at least as good a random number as H(x,y), no matter how awful z is
- Issue: the entity that chooses z can strongly influence the resulting “random” number
  1. Generate a random z
  2. Try computing H(x,y,z)
  3. If H(x,y,z) doesn't start with bits 0000, go back to step 1
  4. Else, output this value of z
- Result: H(x,y,z) begins with four known bits of 0000, even if x and y were perfectly random
- Also, this attack is fast: it only takes 16 computations of H on average

Extraction is hard (especially with forward secrecy)



Statistical tests to see if entropy is "sufficient"

Step 1: Harvest
Reliably produce a "pool" of bits with true *entropy*

Step 2: Extract
Produce ~128 nearly uniform bits from the pool

Step 3: Expand
Create a large sequence of pseudorandom bits