Exploiting Poor Randomness

Nadia Heninger

University of Pennsylvania

October 13, 2014

Exploiting poor randomness

Part 1: A brief tour of PRNGs in practice. (today)

How are random numbers really generated in practice?

Part 2a: Scanning the Internet.
(Tuesday morning)

How do we get cryptographic data to study?

Part 2b: RSA, DSA, and ECDSA disasters.
(Tuesday morning)

Computing TLS and SSH private keys in practice.

Part 3: Lattice-based techniques.
(Thursday afternoon)

Computing smartcard-generated RSA private keys.

A brief tour of PRNGs in practice.

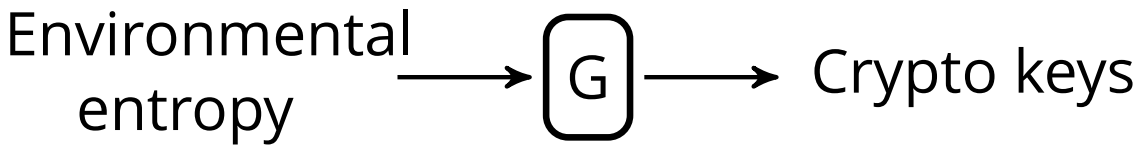
“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.”

–John von Neumann

Theory view

Definition

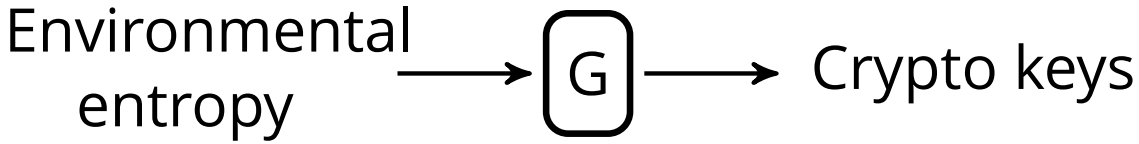
A *pseudorandom generator* is a polynomial-time deterministic function G mapping n-bit strings into $\ell(n)$-bit strings for $\ell(n) \geq n$ whose output distribution $G(U_n)$ is computationally indistinguishable from the uniform distribution $U_{\ell(n)}$.



Theory view

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 $U_{\ell(n)}$.

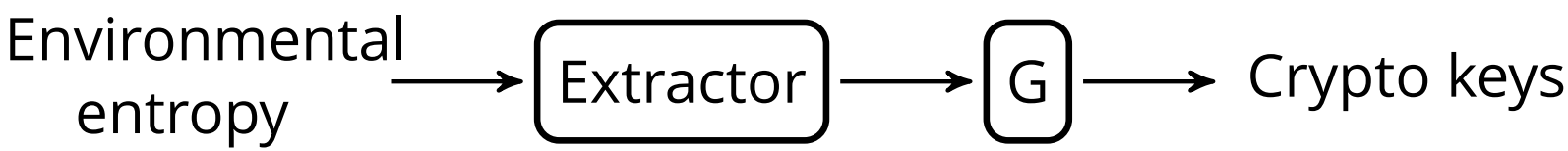


Problem: Environmental entropy not uniformly distributed.

Theory view

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 $U_{\ell(n)}$.



Practical Considerations/Threat Modeling

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

Practical Considerations/Threat Modeling

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

Practical Considerations/Threat Modeling

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

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

Practical Considerations/Threat Modeling

- **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/Threat Modeling

- **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/Threat Modeling

- **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: Reseed/incorporate new entropy from different sources and hope attacker doesn't control everything.

NIST SP800-90A

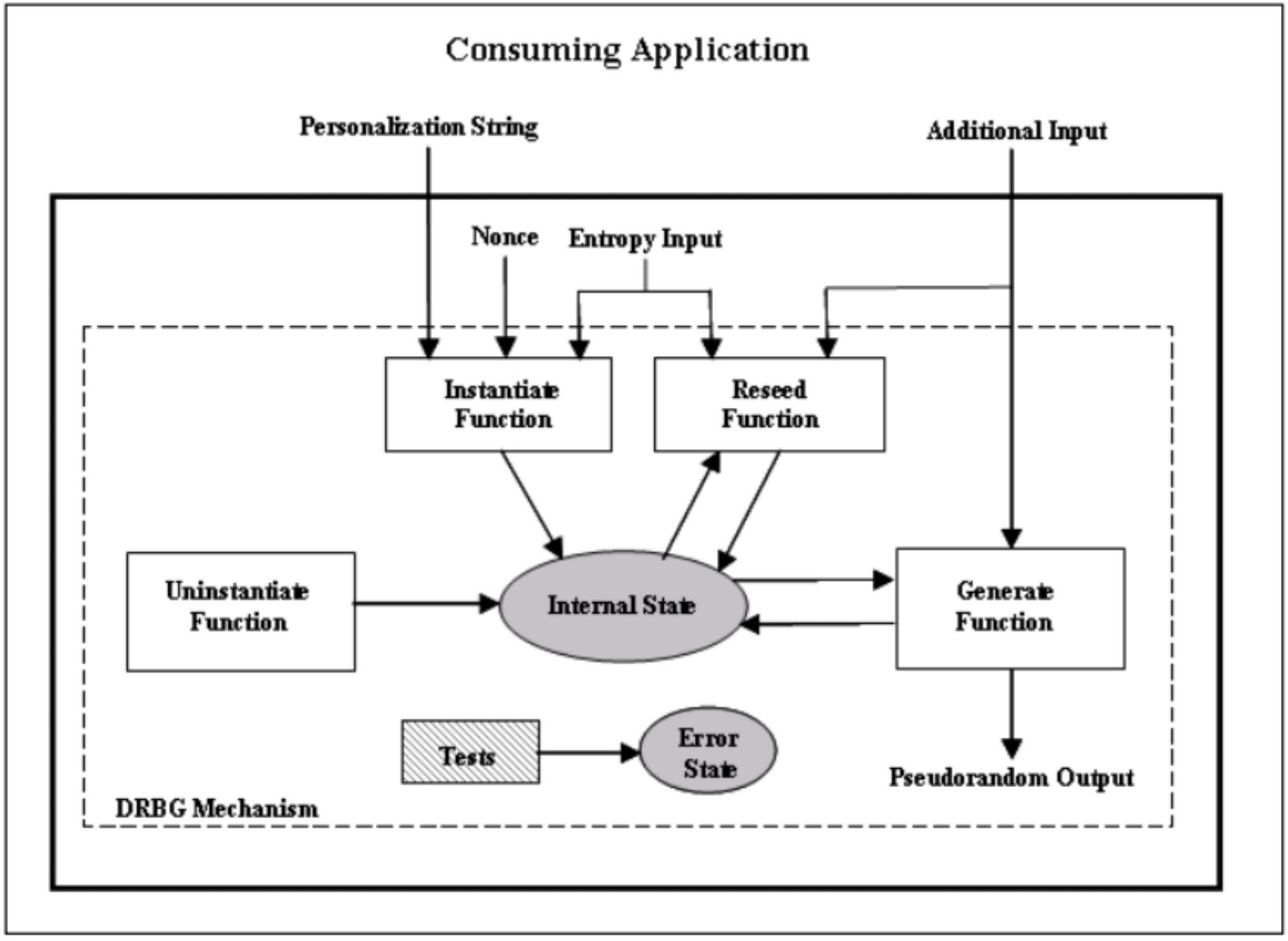


Figure 1: DRBG Functional Model

Design Considerations

1. What do you do if output is requested before seeding?
   Possible answers:
   1.1 Don’t provide output.
   1.2 Provide output.
   1.3 Raise an error flag.
2. How often do you reseed?
   Possible answers:
   2.1 On every new input.
   2.2 After k inputs accumulated in input pools.
   2.3 After ℓ blocks of outputs requested.

Design Considerations

1. What do you do if output is requested before seeding?
   Possible answers:
   1.1 Don’t provide output.
   1.2 Provide output.
   1.3 Raise an error flag. ✓
2. How often do you reseed?
   Possible answers:
   2.1 On every new input.
   2.2 After k inputs accumulated in input pools. ✓
   2.3 After ℓ blocks of outputs requested. ✓

Threat Modeling

- **Problem:** Attacker might influence PRNG design.

NIST SP800-90A

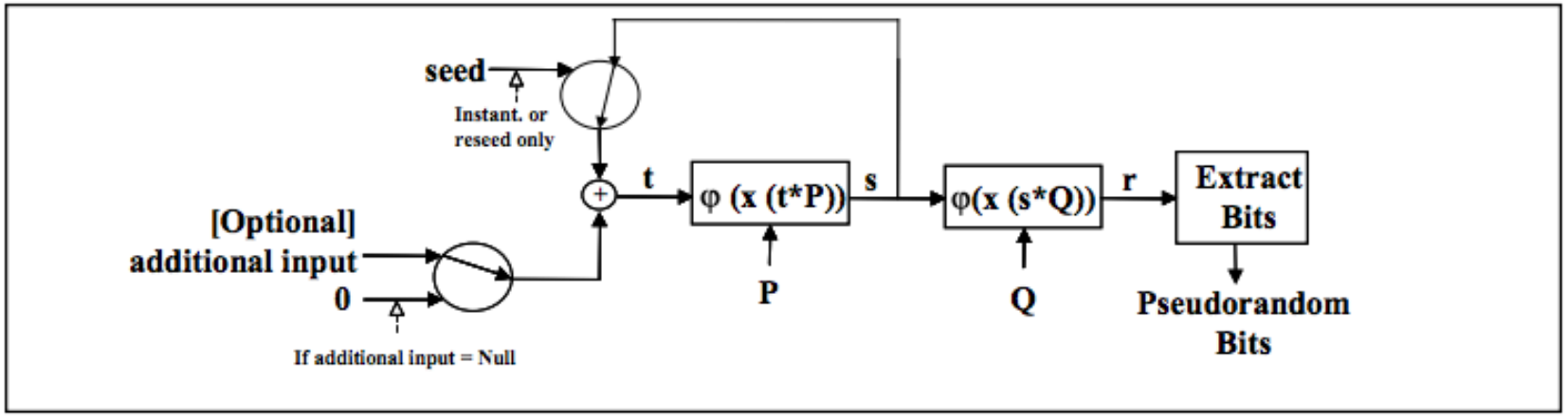


Figure 13: Dual_EC_DRBG

- 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.

NIST SP800-90A

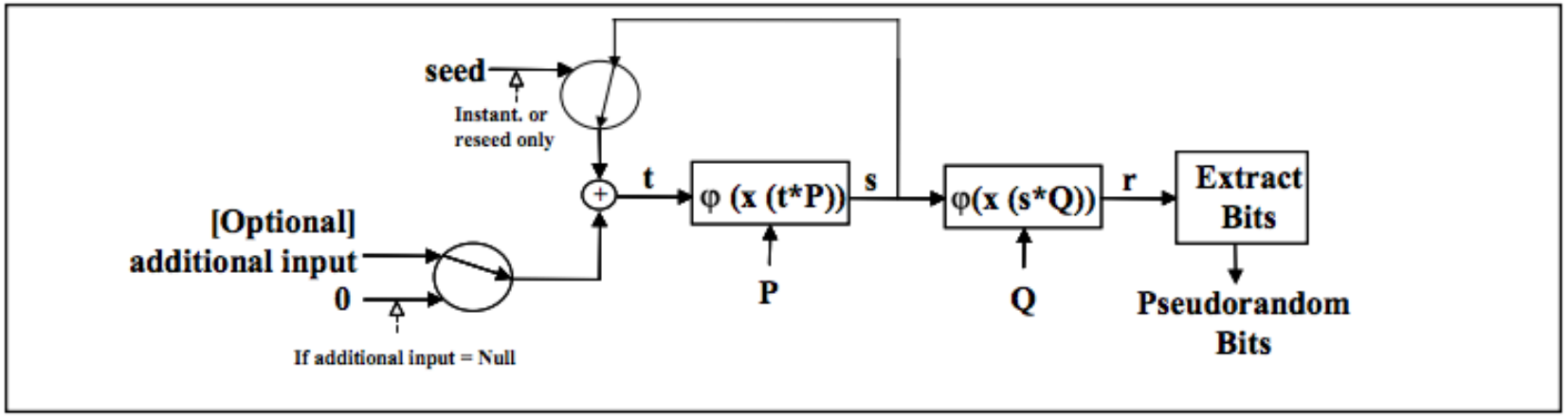


Figure 13: Dual_EC_DRBG

- 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.

Cryptographers: Dual-EC is biased and slow.

Shumow and Ferguson 2007 Crypto rump session

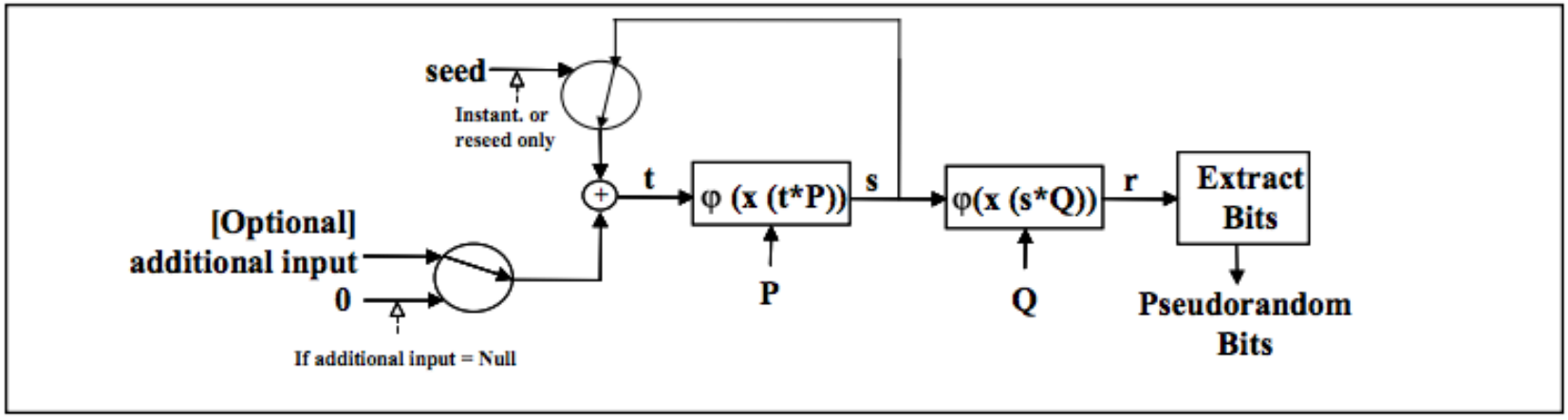


Figure 13: Dual_EC_DRBG

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.

On the Practical Exploitability of Dual EC in TLS Implementations

Checkoway, Fredrikson, Niederhagen, Everspaugh, Green, Lange, Ristenpart, Bernstein, Maskiewicz, Shacham Usenix Security 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	$35 + k$	$2^{15}(C_v + 3C_f) + 2^{35+k}(2C_f)$	$2^k \cdot 83.32$

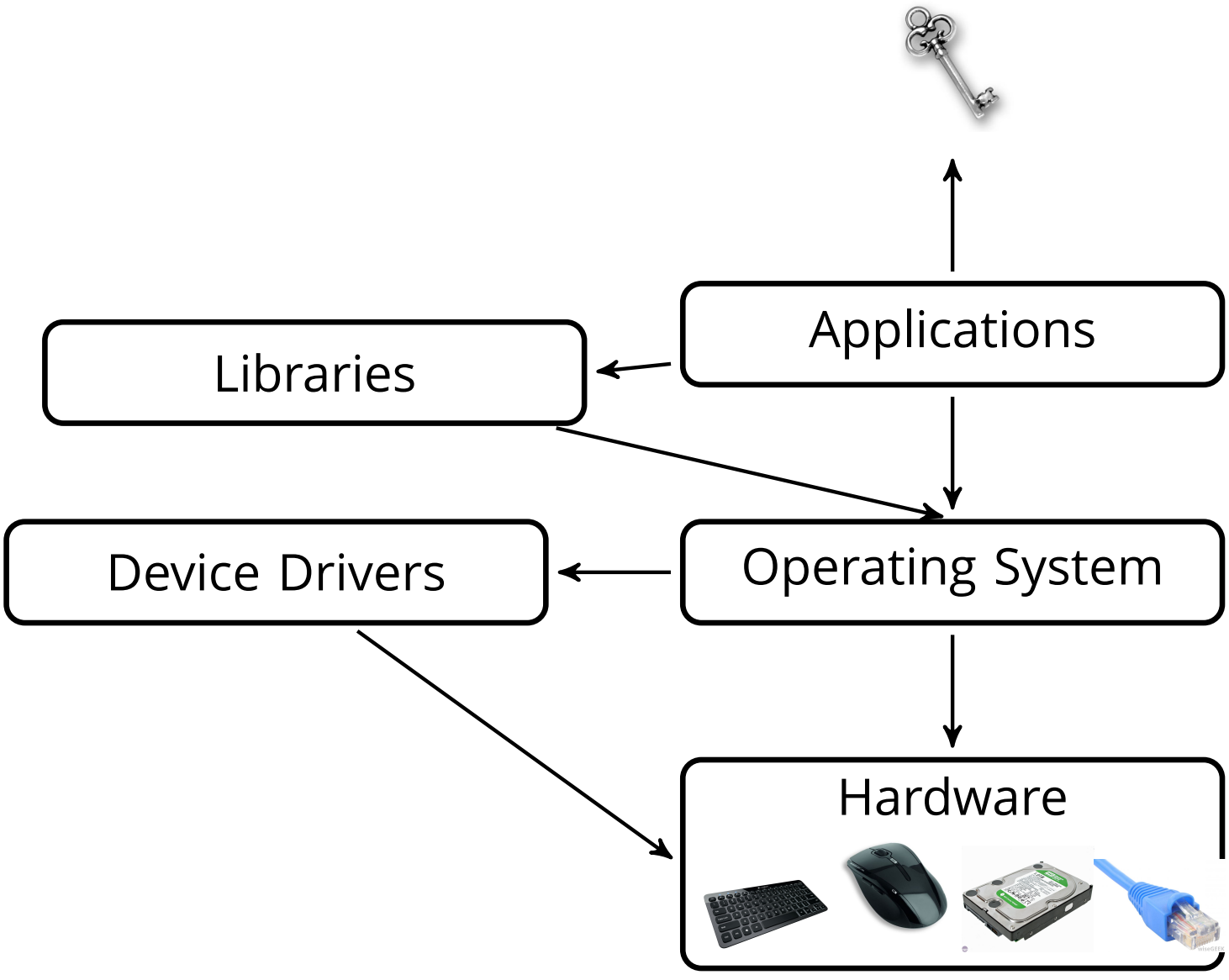
* Assuming process ID and counter known. ** Assuming 15 bits of entropy in process ID, maximum counter of 2^k. See Section 4.3.
† With a library–compile-time flag. ‡ Versions tested: Windows 7 64-bit Service Pack 1 and Windows Server 2010 R2.

Kleptography:
Using Cryptography Against Cryptography

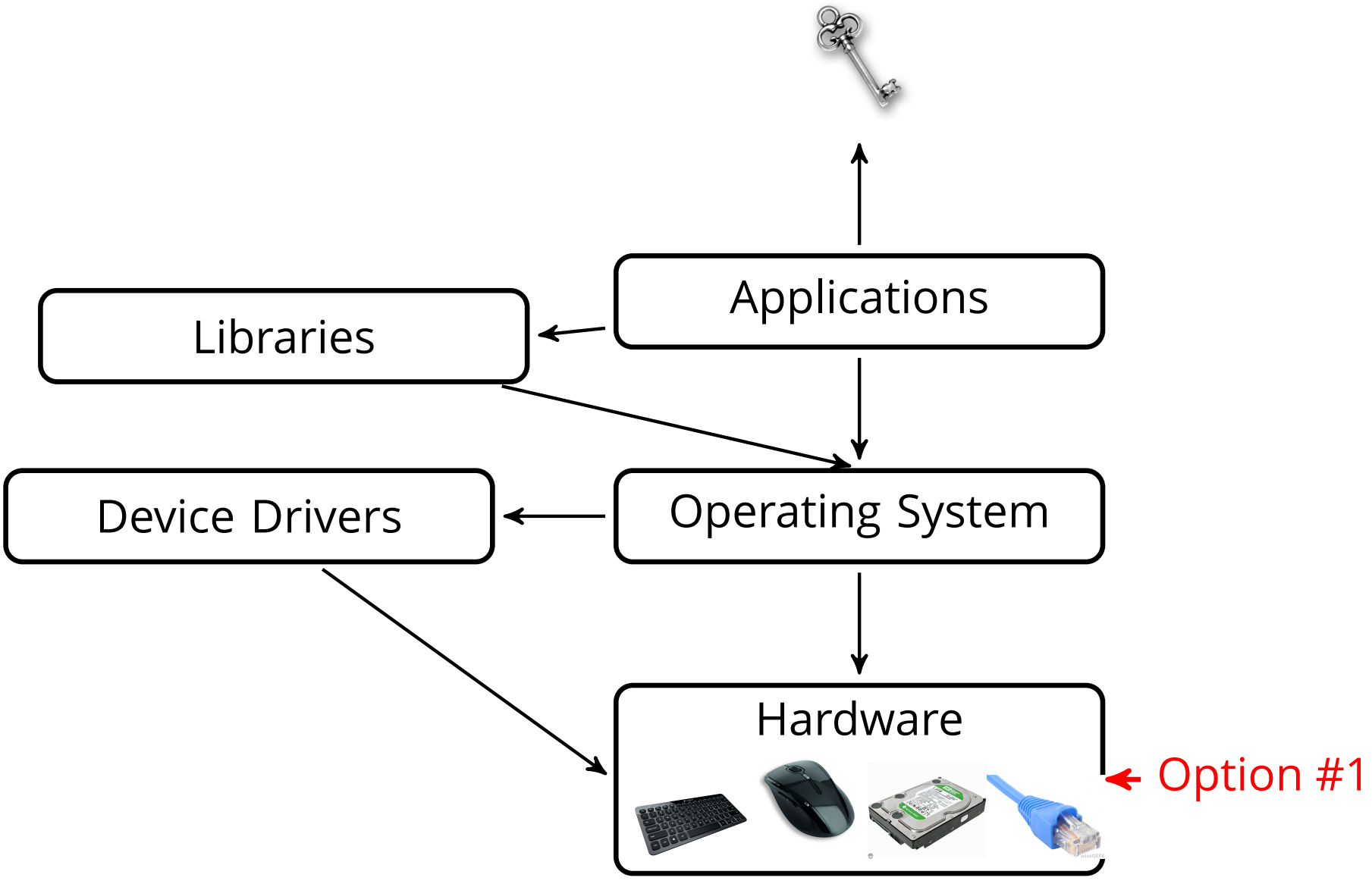
Adam Young* and Moti Yung**

Open problem: How might we detect kleptographic attacks in practice?

Where do we put the RNG?



Where do we put the RNG?



Intel RDRAND instruction

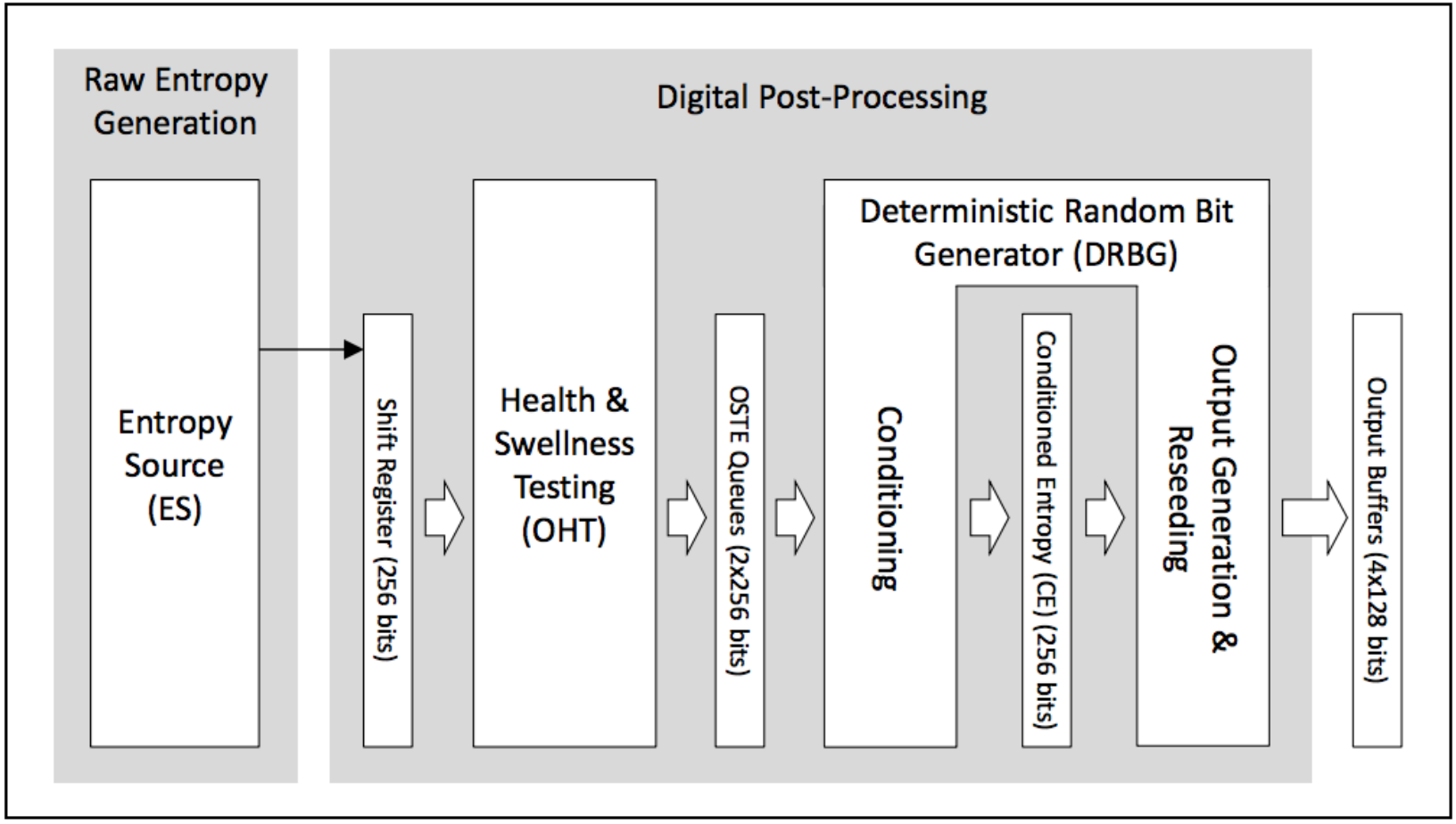


Figure 1: Block diagram of the Intel RNG (adapted from [7])

"ANALYSIS OF INTEL'S IVY BRIDGE DIGITAL RANDOM NUMBER GENERATOR" Hamburg Kocher Marson

Intel RDRAND Design Choices

The entropy source (ES) at the heart of the Intel RNG is a self-oscillating digital circuit with feedback, shown in Figure 2 below.

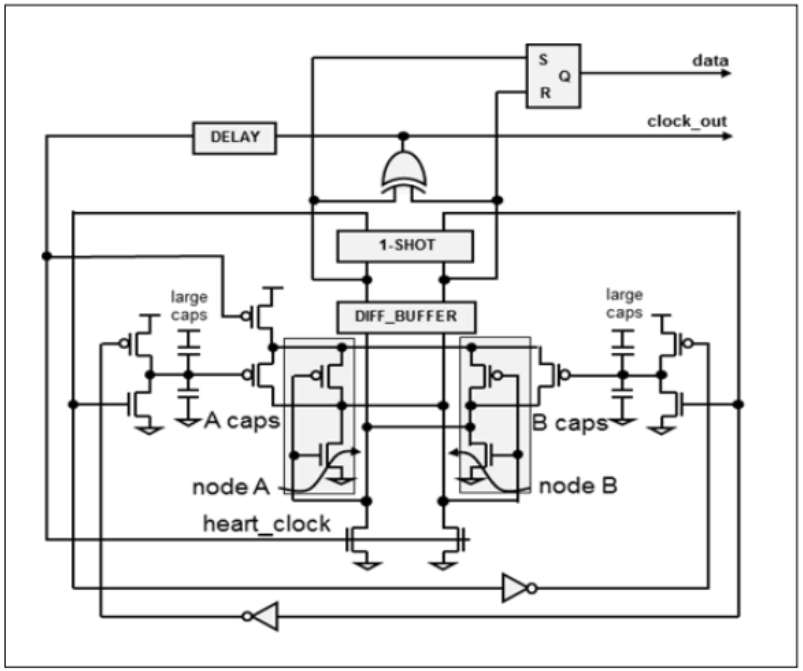


Figure 2: Entropy source for the Intel RNG (from [8])

"ANALYSIS OF INTEL'S IVY BRIDGE DIGITAL RANDOM NUMBER GENERATOR" Hamburg Kocher Marson

- Entropy source: Oscillating circuit
- Extractor/conditioning: Iterated AES-128 with fixed key.
- PRG: AES-128-CTR
- Reseeding: At least every 65536 bits of output.
- If tests fail, clear carry flag and return all 0s.

Threat Modeling

- **Problem:** Attacker might influence PRNG design.

Threat Modeling

- **Problem:** Attacker might influence PRNG design.
- **Problem:** Attacker might control hardware supply chain.

Threat Modeling

- **Problem:** Attacker might influence PRNG design.
- **Problem:** Attacker might control hardware supply chain.
  Solution: Audit hardware, compare to known good design.

Threat Modeling

- **Problem:** Attacker might influence PRNG design.

- **Problem:** Attacker might control hardware supply chain.
  Solution: Audit hardware, compare to known good design.
- **Problem:** Attacker might control hardware supply chain and be really clever.

Stealthy Dopant-Level Hardware Trojans

by Becker, Regazzoni, Paar, and Burleson, CHES 2013

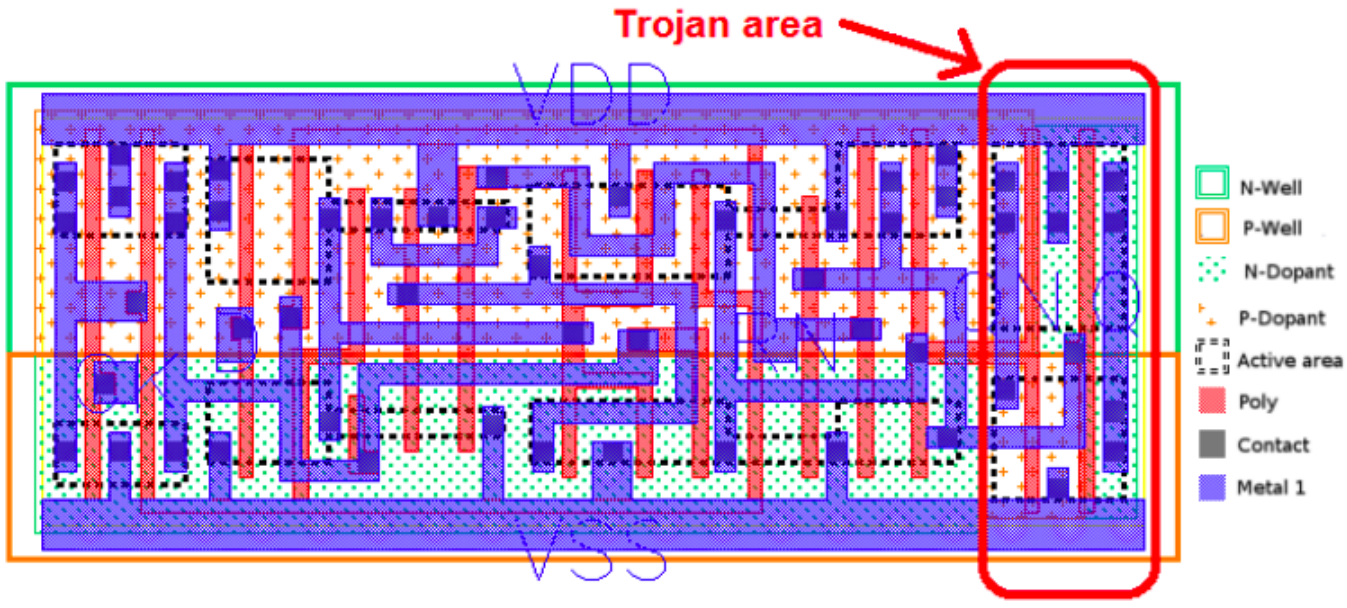
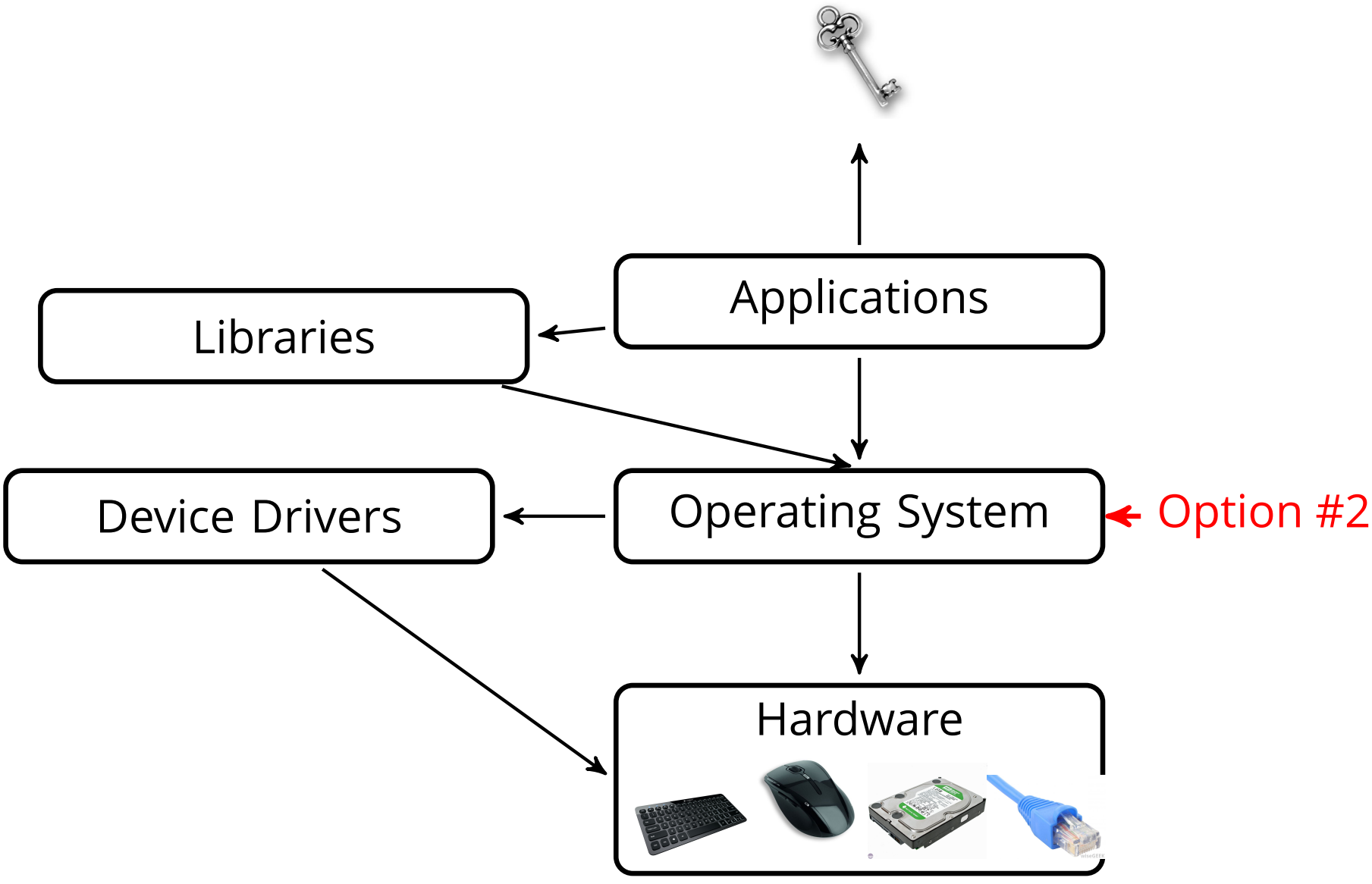


Fig. 2. Layout of the Trojan DFFR_X1 gate. The gate is only modified in the highlighted area by changing the dopant mask. The resulting Trojan gate has an output of $Q = V_{DD}$ and $QN = GND$.

Undetectable Trojan outputs AES with fixed *k* and 32-bit ctr.

Where do we put the RNG?



Linux random number generators

- heuristically measures input entropy
- input pool mixed into output once counter reaches 192 bits
- CRC-based state mixing function
- output is SHA-1 hash of state

`/dev/random`

- blocks if insufficient randomness available

`/dev/urandom`

- output never blocks

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

Misconceptions abound…

/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

Blocking output is a usability problem

```
 /* 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"
```

/dev/random is not robust

Dodis Pointcheval Ruhault Vergnaud Wichs CCS 2013

- Can adversarially construct inputs to fool `/dev/random`'s entropy estimator.
- CRC-based state mixing function does not recover well from state compromise.

Open Problem: Produce attack strong enough to convince Ted Ts'o or Linus Torvalds these are real vulnerabilities.

Linux and Intel RDRAND

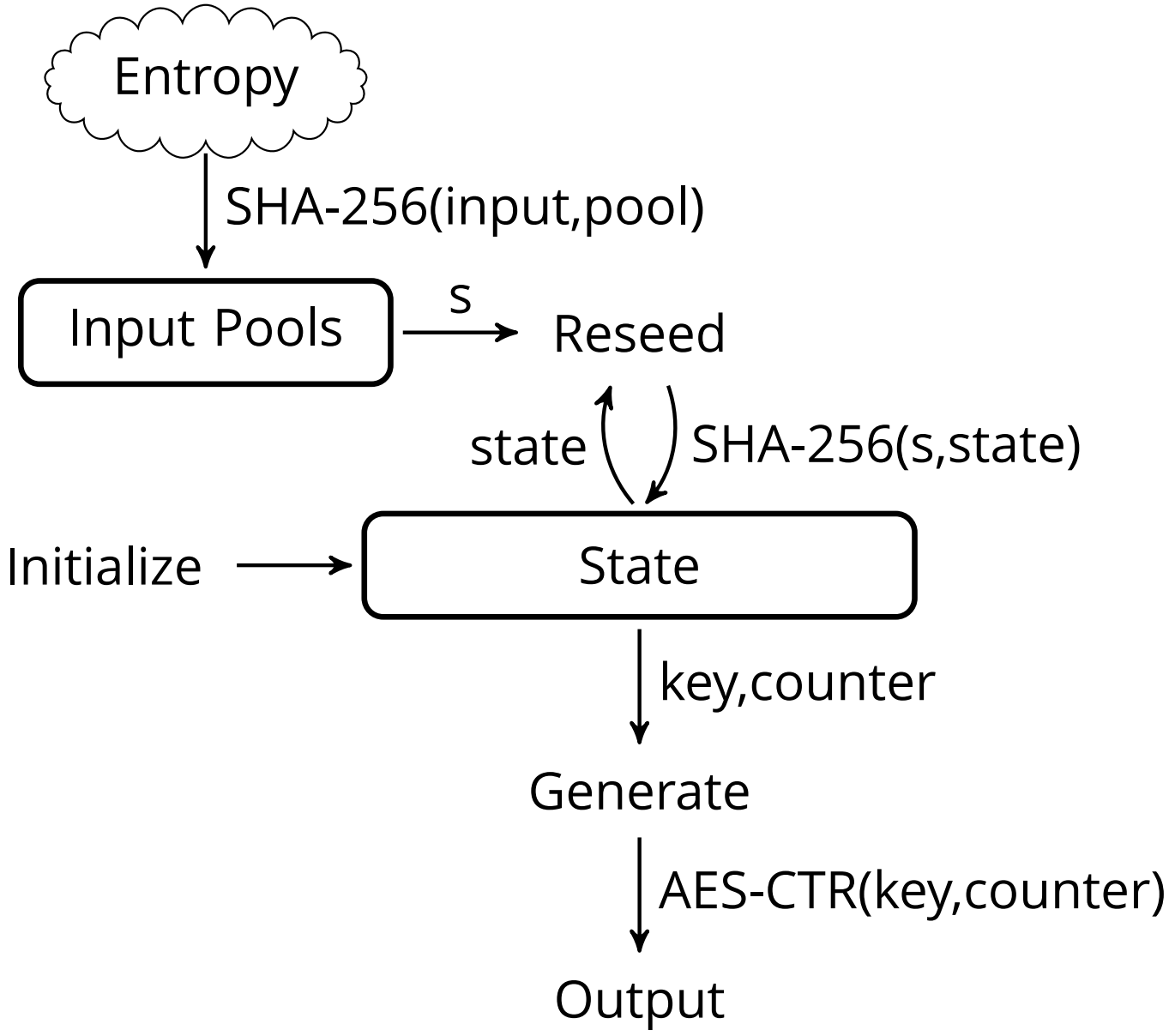
```
/*
 * This function is the exported kernel interface.  It returns some
 * number of good random numbers, suitable for key generation, seeding
 * TCP sequence numbers, etc.  It does not rely on the hardware random
 * number generator.  For random bytes direct from the hardware RNG
 * (when available), use get_random_bytes_arch().
 */
void get_random_bytes(void *buf, int nbytes)

/*
 * This function will use the architecture-specific hardware random
 * number generator if it is available.  The arch-specific hw RNG will
 * almost certainly be faster than what we can do in software, but it
 * is impossible to verify that it is implemented securely (as
 * opposed, to, say, the AES encryption of a sequence number using a
 * key known by the NSA).  So it's useful if we need the speed, but
 * only if we're willing to trust the hardware manufacturer not to
 * have put in a back door.
 */
void get_random_bytes_arch(void *buf, int nbytes)
```

getrandom() system call

```
Date: Thu, 17 Jul 2014 05:18:15 -0400
From: Theodore Ts'o <tytso@....edu>
To: linux-kernel@...r.kernel.org
Cc: linux-abi@...r.kernel.org, linux-crypto@...r.kernel.org,
beck@...nbsd.org, Theodore Ts'o <tytso@....edu>
Subject: [PATCH, RFC] random: introduce getrandom(2) system call

The getrandom(2) system call was requested by the LibreSSL Portable
developers.  It is analoguous to the getentropy(2) system call in
OpenBSD.

The rationale of this system call is to provide resiliance against
file descriptor exhaustion attacks, where the attacker consumes all
available file descriptors, forcing the use of the fallback code where
/dev/[u]random is not available.  Since the fallback code is often not
well-tested, it is better to eliminate this potential failure mode
entirely.

The other feature provided by this new system call is the ability to
request randomness from the /dev/urandom entropy pool, but to block
until at least 128 bits of entropy has been accumulated in the
/dev/urandom entropy pool.
```

Fortuna

Ferguson, Schneier, Kohno



Design Choices

- Avoid entropy estimation by mixing inputs into many pools and mixing pools at different rates.
- Cryptographic state mixing function.
- No distinction between `/dev/random` and `/dev/urandom` on BSD-based systems using Yarrow (precursor to Fortuna).

===

FreeBSD-SA-08.11.arc4random Security Advisory
The FreeBSD Project

Topic: arc4random(9) predictable sequence vulnerability

Category: core
Module: sys
Announced: 2008-11-24

...

II. Problem Description

When the arc4random(9) random number generator is initialized, there may be inadequate entropy to meet the needs of kernel systems which rely on arc4random(9); and it may take up to 5 minutes before arc4random(9) is reseeded with secure entropy from the Yarrow random number generator.

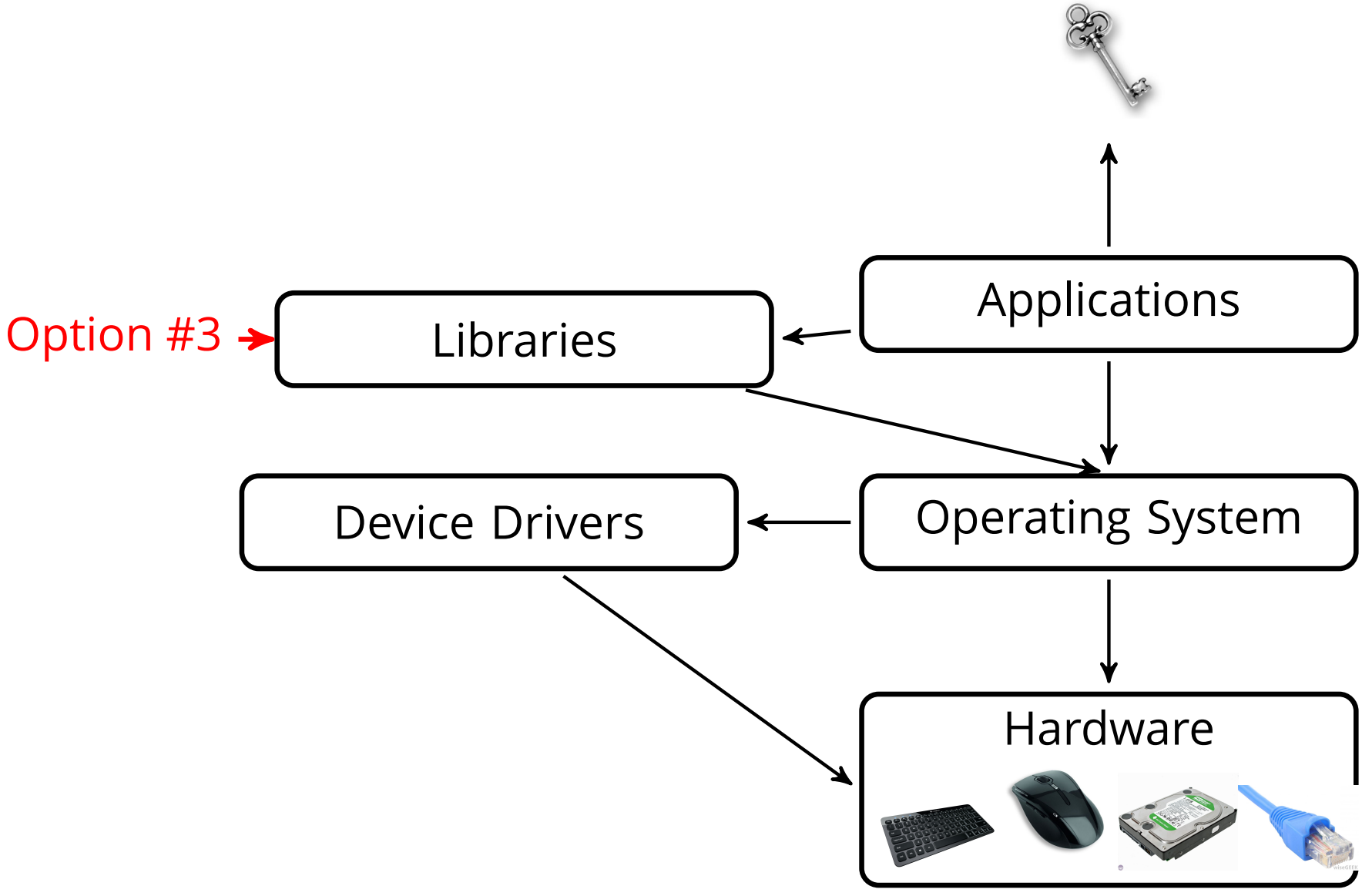
Windows

RNG not publicly documented.

Dorrendorf, Gutterman, Pinkas 2007 reverse-engineered RNG in Windows 2000.

- State refreshed every 128KB of output.
- State compromise leads to forward and backward recovery of all outputs for that state.

Where do we put the RNG?



OpenSSL PRNG

- Seeds from `/dev/urandom` (or `/dev/random` if urandom not present), pid, time()
- time() (in seconds) mixed into state before some output queries.
- State mixing function is configurable but defaults to SHA-1 hash.
- 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?

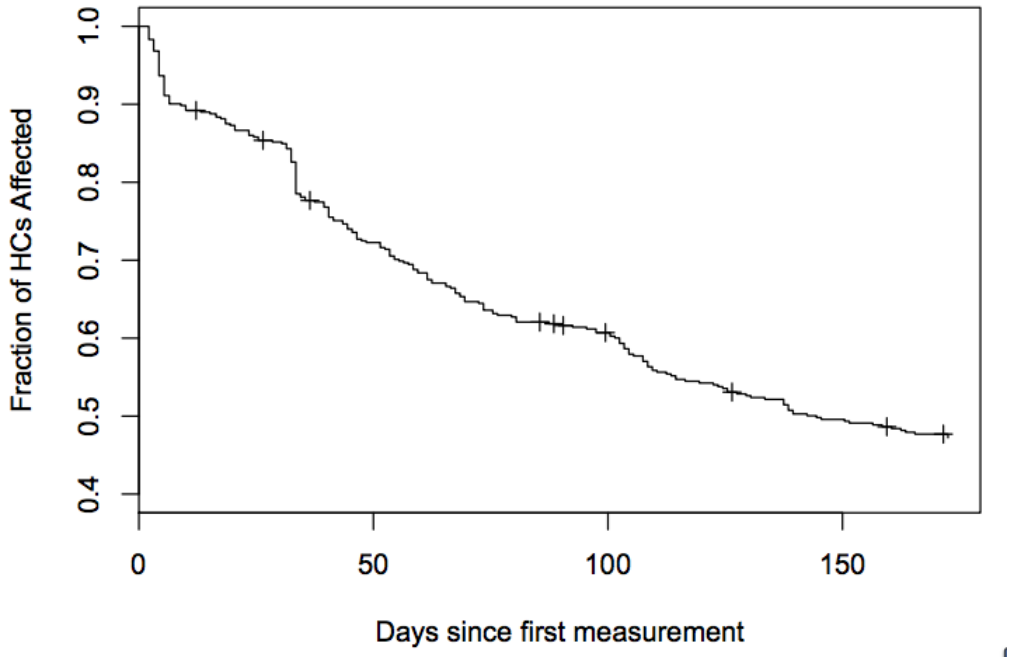
Kurt

The Debian OpenSSL entropy disaster

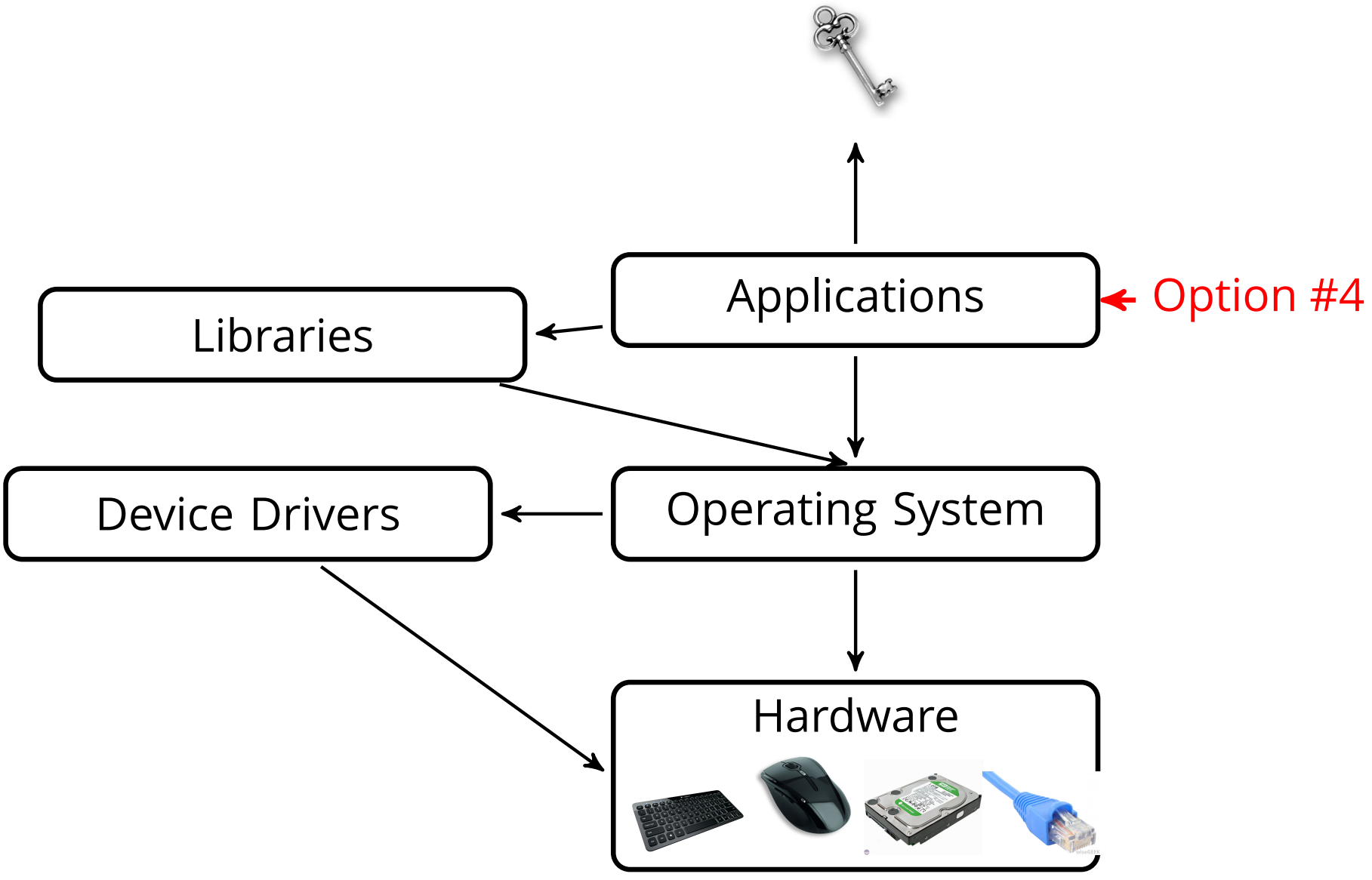
August, 2008: Discovered by Luciano Bello

Keys dependent only on pid and machine architecture: 294,912 keys per key size.

“When Private Keys are Public: Results from the 2008 Debian OpenSSL Vulnerability” [Yilek, Rescorla, Shacham, Enright, Savage 2009]



Where do we put the RNG?



1996 Netscape SSL RNG [Goldberg, Wagner]

```
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();
     tmp = RNG_GenerateRandomBytes();
     tmp = RNG_GenerateRandomBytes();
     challenge = RNG_GenerateRandomBytes();
     secret_key = RNG_GenerateRandomBytes();
```

Common pattern:

Seed once from OS, maintain individual application-specific PRNG.

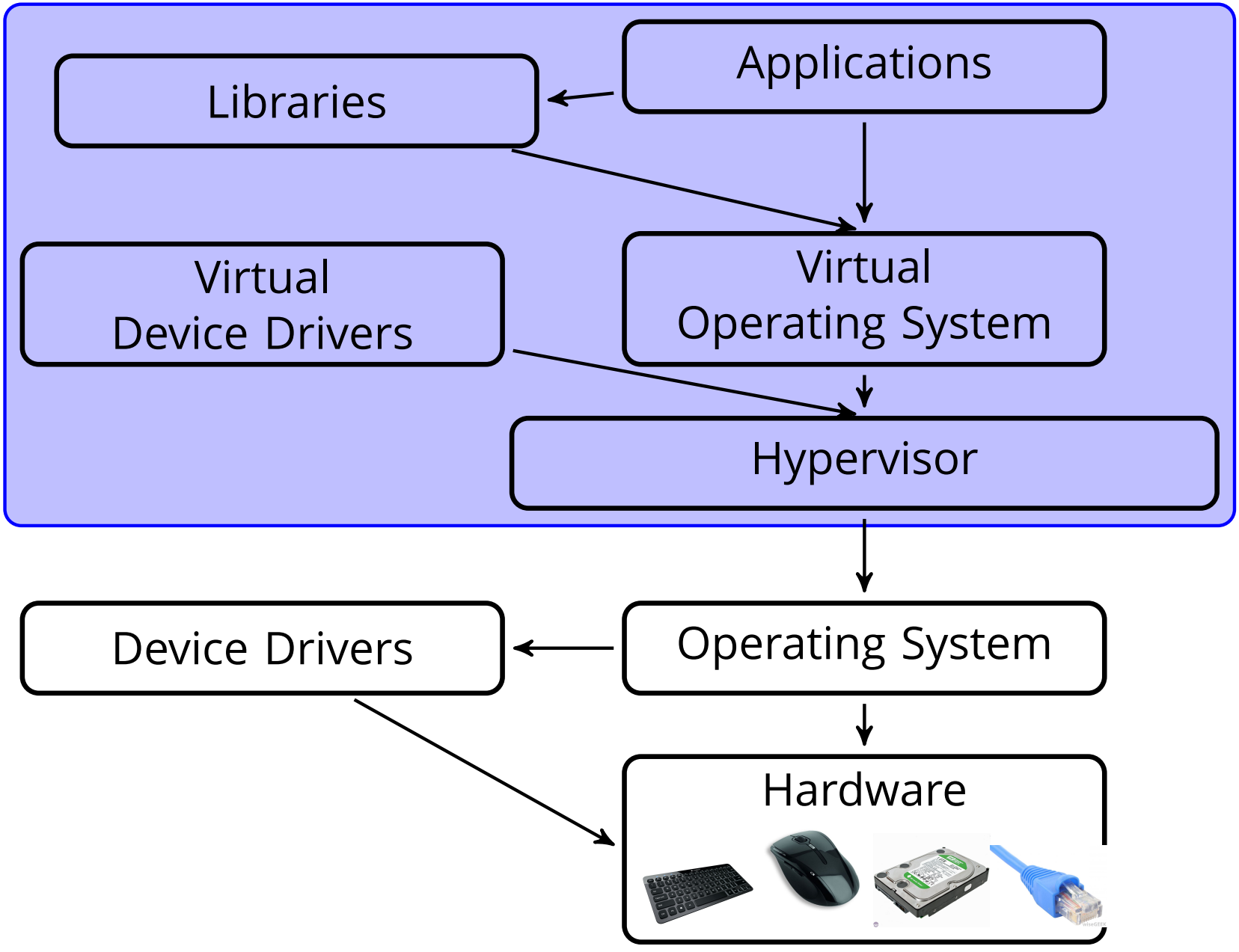
August 2013: Android SecureRandom vulnerability

> *We recently learned that a component of Android responsible for generating secure random numbers contains critical weaknesses, that render all Android wallets generated to date vulnerable to theft. Because the problem lies with Android itself, this problem will affect you if you have a wallet generated by any Android app.*

– Bitcoin security advisory

Underlying problem: OpenSSL not fork-safe. RNG state not re-initialized on fork, so only new element to mix in is pid, which has 2^{16} bits of entropy.

What about virtualization?



Virtual Machine Reset Vulnerabilities

Ristenpart Yilek NDSS 2010

1. Guest OS running in VM initializes RNG state.
2. User snapshots guest OS.
3. User performs actions on guest OS.
4. Guest OS reset to snapshot state.

July 2013: DigitalOcean security advisory

The SSH host keys for some Ubuntu-based systems could have been duplicated by DigitalOcean's snapshot and creation process. Therefore, our system is now configured to remove the host keys on Droplets that are created from snapshots at the time of the first boot.

Summary

Random number generation is hard.

Practical constraints:

- Performance
- Usability for developers
- Cross-platform software compatibility:
  high-performance servers, mobile devices, embedded devices...
- Diverse and hard to audit hardware