Robust, low-cost, auditable random number generation for embedded system security

Ben Lampert\textsuperscript{*\circ}, Riad S. Wahby\textsuperscript{*}, Shane Leonard\textsuperscript{*}, and Philip Levis\textsuperscript{*}

\textsuperscript{*}Stanford University
\textsuperscript{\circ}NAUTO, Inc.

November 14\textsuperscript{th}, 2016
All secure systems depend on random numbers
DO YOU KNOW WHERE YOUR RANDOM NUMBERS COME FROM?
All secure systems depend on random numbers

Embedded systems face unique challenges
All secure systems depend on random numbers

Embedded systems face unique challenges

We present a hardware/software system for random number generation tailored to embedded devices:

- hardware costs $\approx 1.50, 1.5 \text{ cm}^2$ board area
- run once at boot, takes 25 ms to initialize
- energy cost equivalent to 10 ZigBee packets
Debian Bug Leaves Private SSL/SSH Keys Guessable

Posted by timothy on Tuesday May 13, 2008 @11:01AM from the security-is-a-process dept.
```c
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
    // guaranteed to be random.
}
```
A deterministic “random” number generator?

What properties would it have?

- Uniformly distributed.
  
  GNU libc’s rand() output is very nearly uniform
A deterministic “random” number generator?

What properties would it have?

- Uniformly distributed. Not enough.
  
  GNU libc’s rand() output is very nearly uniform
  But future outputs are easy to predict given past outputs
A deterministic “random” number generator?

What properties would it have?

- Uniformly distributed. Not enough.
  GNU libc’s `rand()` output is very nearly uniform
  But future outputs are easy to predict given past outputs

- …and no self correlation.
  $\pi$ is thought to be normal: it will pass many statistical tests
  What if we used digits of $\pi$?
A deterministic "random" number generator?

What properties would it have?

- Uniformly distributed. **Not enough.**
  GNU libc’s `rand()` output is very nearly uniform
  But **future outputs are easy to predict given past outputs**

- **...and no self correlation.** **Still no.**
  \( \pi \) is thought to be *normal*: it will pass many statistical tests
  What if we used digits of \( \pi \)?
  But **future outputs are easy to predict given knowledge of source**
A deterministic “random” number generator?

What properties would it have?

- Uniformly distributed. Not enough.
  GNU libc’s rand() output is very nearly uniform
  But future outputs are easy to predict given past outputs

- ...and no self correlation. Still no.
  $\pi$ is thought to be normal: it will pass many statistical tests
  What if we used digits of $\pi$?
  But future outputs are easy to predict given knowledge of source

Idea: add a secret!
Cryptographically secure pseudorandom number generator

**CSPRNG:** a deterministic algorithm that generates “good randomness” given a secret key $k$
Cryptographically secure pseudorandom number generator

**CSPRNG**: a deterministic algorithm that generates “good randomness” given a secret key \( k \)

**Key property (informal)**: given past outputs, no efficient algorithm can predict future outputs.
Cryptographically secure pseudorandom number generator

**CSPRNG**: a deterministic algorithm that generates “good randomness” given a secret key $k$

Key property (informal): given past outputs, no efficient algorithm can predict future outputs

Concrete example: using AES, encrypt the sequence \{0, 1, 2, 3, …\} under secret key $k$
Cryptographically secure pseudorandom number generator

**CSPRNG**: a deterministic algorithm that generates “good randomness” given a secret key $k$

**Key property (informal)**: given past outputs, no efficient algorithm can predict future outputs

**Concrete example**: using AES, encrypt the sequence \( \{0, 1, 2, 3, \ldots \} \) under secret key $k$ 

\[ \Rightarrow \text{can securely generate } > 2^{60} \text{ bytes!} \]
Cryptographically secure pseudorandom number generator

**CSPRNG:** a deterministic algorithm that generates “good randomness” given a secret key $k$

**Key property (informal):** given past outputs, no efficient algorithm can predict future outputs

**Concrete example:** using AES, encrypt the sequence $\{0, 1, 2, 3, \ldots\}$ under secret key $k$  
$\Rightarrow$ can securely generate $> 2^{60}$ bytes!

**Figure of merit:** entropy
informally: the number of bits in $k$ that an adversary does not know
Why not existing solutions?

- Gather entropy from many sources
  e.g., hard disk, keyboard, network timing
Why not existing solutions?

- Gather entropy from many sources
e.g., hard disk, keyboard, network timing

× Embedded systems may not have these sources
× Continuous gathering costs energy
Why not existing solutions?

- Gather entropy from many sources  
  e.g., hard disk, keyboard, network timing

  ✗ Embedded systems may not have these sources

  ✗ Continuous gathering costs energy

- RNG built into processor or SoC  
  e.g., RDRAND on Intel processors
Why not existing solutions?

- Gather entropy from many sources
e.g., hard disk, keyboard, network timing

- **Not suitable:** Embedded systems may not have these sources
- **Not suitable:** Continuous gathering costs energy

- RNG built into processor or SoC
e.g., RDRAND on Intel processors

- **Not suitable:** Embedded processors may not have RNG
Why not existing solutions?

- Gather entropy from many sources e.g., hard disk, keyboard, network timing

  ✗ Embedded systems may not have these sources
  ✗ Continuous gathering costs energy

- RNG built into processor or SoC e.g., RDRAND on Intel processors

  ✗ Embedded processors may not have RNG
  ✗ Integrated RNG is opaque, not auditable

  • Becker et al. [CHES ’13] showed that integrated hardware RNGs can be stealthily backdoored
Wish list

- Inexpensive
- Small
- Low power
- Insensitive to environmental factors (e.g., temperature, RF interference)
- Easy to detect failure: simple and auditable
- Generates a CSPRNG key quickly
Generating unpredictable bits: two easy pieces

Noise source: a device exhibiting an unpredictable physical phenomenon
Conversion circuit: detects state of device, produces corresponding bits
Generating unpredictable bits: two easy pieces

Example noise sources:
  - Radioactive decay
  - Beam splitting
  - Photoelectric effect
  - Circuit noise
    - thermal noise (all electronic devices)
    - shot noise, flicker noise (diodes and transistors)
    - Zener noise, avalanche noise (diodes)
Generating unpredictable bits: two easy pieces

Example noise sources:
- Radioactive decay
- Beam splitting
- Photoelectric effect

Circuit noise
- thermal noise (all electronic devices)
- shot noise, flicker noise (diodes and transistors)
- Zener noise, avalanche noise (diodes)
Generating unpredictable bits: two easy pieces

Example noise sources:
- Radioactive decay
- Beam splitting
- Photoelectric effect

Circuit noise
- thermal noise (all electronic devices)
- shot noise, flicker noise (diodes and transistors)
- Zener noise, avalanche noise (diodes)
Generating unpredictable bits: two easy pieces

Example noise sources:

- Radioactive decay
- Beam splitting
- Photoelectric effect

Circuit noise

- Thermal noise (all electronic devices)
- Shot noise, flicker noise (diodes and transistors)
- Zener noise, avalanche noise (diodes)
Generating unpredictable bits: two easy pieces

Example noise sources:
- × Radioactive decay
- × Beam splitting
- × Photoelectric effect
- ✓ Circuit noise

- × thermal noise (all electronic devices)
- × shot noise, flicker noise (diodes and transistors)
- ✓ Zener noise, avalanche noise (diodes)
Generating unpredictable bits: two easy pieces

Example noise sources:

✗ Radioactive decay
✗ Beam splitting
✗ Photoelectric effect
✓ Circuit noise
  ✗ thermal noise (all electronic devices)
  ✗ shot noise, flicker noise (diodes and transistors)
✓ Zener noise, **avalanche noise** (diodes)
Diodes, reverse breakdown, and avalanche

Voltage applied in forward direction: current can flow

Low voltage applied in reverse direction: current cannot flow

Avalanche current: electron collisions cause an "avalanche" of charge carriers
Diodes, reverse breakdown, and avalanche

Voltage applied in forward direction: current can flow

\[ i > 0 \]

Low voltage applied in reverse direction: current cannot flow

\[ i = 0 \]

High voltage applied in reverse direction: breakdown, current flow

\[ i > 0 \]
Diodes, reverse breakdown, and avalanche

Voltage applied in forward direction: current can flow

\[ + \rightarrow - \]
\[ i > 0 \]

Low voltage applied in reverse direction: current cannot flow

\[ - \xrightarrow{\times} + \]
\[ i = 0 \]

High voltage applied in reverse direction: breakdown, current flow

\[ - \rightarrow + \]
\[ i > 0 \]

Avalanche current:
electron collisions cause an “avalanche” of charge carriers
Avalanche current

12.16 V

\( V_{\text{noise}} \)

10 kΩ
Overcoming manufacturing variations

\[ V_{\text{ref}} \]

\[ + \]

\[ - \]

\[ R1 \]

10 kΩ

\[ V_{\text{noise}} \]
Converting $V_{\text{noise}}$ to bits

\[ V_{\text{ref}} \]

\[ + \]

\[ - \]

\[ R1 \quad 10 \, \text{k}\Omega \]

\[ V_{\text{noise}} \]

output bits
Issue: outside disturbances

\[ V_{ref} \]

\[ i_{dist} \sim \]

\[ R1 \]

\[ 10 \text{ k}\Omega \]

\[ V_{noise} \]

output bits
Overcoming disturbances using a differential circuit

\[ V_{\text{ref}} - + \]

\[ - + \]

\[ D1 \]

\[ R1 \quad 10 \, \text{k}\Omega \]

\[ V_{\text{noise},1} \]

\[ V_{\text{noise},2} \]

\[ \text{output bits} \]
Overcoming disturbances using a differential circuit

\[ V_{\text{ref}} \]

\[ V_{\text{noise,1}} \]

\[ V_{\text{noise,2}} \]

\[ i_{\text{dist}} \]

\[ R_1 \quad 10 \text{ k}\Omega \]

\[ R_2 \quad 10 \text{ k}\Omega \]

\[ \text{output bits} \]
Issue: how do we generate 12 V?

Controller

M1

L1 10 µH

D1

C_{out} 10 µF

V_{boost} > 3.3 V
Issue: how do we generate 12 V?

Issue: the boost converter causes large disturbances
Issue: how do we generate 12 V?

Issue: the boost converter causes large disturbances
Solution: interleave boost and output sampling
Interleaved boost operation

V_{boost}

i_{boost}

comparator output
Putting it all together

- At boot:
  1. run circuit to gather 1024 bits, $b_{\text{raw}}$
  2. compute $k = \text{SHA256} (b_{\text{raw}})$
  3. initialize global counter $c = 0$

- To generate a random number:
  1. increment counter $c$
  2. use AES to encrypt $c$ under key $k$
  3. return resulting ciphertext
Putting it all together

- At boot:
  1. run circuit to gather 1024 bits, $b_{\text{raw}}$
  2. compute $k = \text{SHA256}(b_{\text{raw}})$
  3. initialize global counter $c = 0$

- To generate a random number:
  1. increment counter $c$
  2. use AES to encrypt $c$ under key $k$
  3. return resulting ciphertext
Testing and monitoring

In the paper, we define methods for:

**Acceptance testing:**
after assembly and before deployment, each device should be checked for proper operation

**Online auditing:**
for systems requiring high assurance, further online testing in the field
Evaluation questions

- How quickly should the system sample the bit generator’s output?
- What are the statistical properties of the raw output versus time and temperature?
- What is the cost, in energy and time, of generating a CSPRNG key?
Built systems

Cost ≈ $1.50
Determining the sample rate

Serial Correlation vs Sampling Frequency

Sampling Frequency (Hz) vs Serial Correlation

Graph showing the relationship between sampling frequency (Hz) and serial correlation.
Statistical properties versus temperature

**Entropy vs. Temperature**

- Entropy (closer to 1 is better)

**Serial Correlation vs. Temperature**

- Serial Correlation (closer to 0 is better)
Statistical properties versus time

**Long Term Entropy**

- Entropy (closer to 1 is better)
- Days

**Long Term Serial Correlation**

- Serial Correlation (closer to 0 is better)
- Days
Time and energy costs to generate CSPRNG key

Time to gather 1024 bits:
- ≈13 ms running dc/dc converter
- ≈12 ms sampling output of bit generator

Energy to gather 1024 bits:
- ≈3 μJ per bit
- ≈10× more energy per bit than a ZigBee radio, amortized over all CSPRNG outputs
Conclusions

- You should worry about your random numbers!

- A CSPRNG can generate secure, effectively limitless output given a hard-to-guess key...

- ...but in embedded systems, generating a CSPRNG key is challenging

- We have presented a design tailored to embedded systems for secure, inexpensive pseudorandomness

- Future work: smaller, cheaper, faster

https://github.com/heleNA-project/imix
$V_{\text{high}}$

$R_1$ 10 kΩ

$D_1$

$C_f$ 0.47 μF

$V_{\text{ref}}$

Op-amp 1

$V_{\text{high}}$

$D_2$

$R_2$ 10 kΩ

$V_{\text{noise,1}}$

Op-amp 2

$V_{\text{high}}$

$D_3$

Comparator

3.3 V

$V_{\text{out}}$

Boost converter

3.3 V

$V_{\text{in}}$

enable

$V_{\text{out}}$

$V_{\text{high}}$

$R_3$ 10 kΩ

$V_{\text{noise,2}}$