Solving The Platform Entropy Problem

George Cox

Security Architect

Intel Corporation

Session Introduction

Much of computer security is based upon the use of cryptography.

Cryptography is based upon two things:

- Good algorithms (e.g., AES) and
- Good keys (e.g., good random numbers).

Historically, computing platforms broadly have not had high quality/high performance entropy (or random numbers).

By enabling ALL future Intel platforms with our RdRand instruction (supported by our underlying Digital Random Number Generator (DRNG)), we:

Provide a fundamental solution to the long standing "Platform Entropy Problem" and

Give Intel's customers the "common brand promise" of high quality/high performance entropy (or random numbers) everywhere across ALL Intel products in which it is embedded.

=> the "Platform Entropy Problem" just "goes away" on future Intel-based platforms.

Agenda

The Platform’s Need For Entropy

Entropy is valuable in a variety of uses, the first of which that comes to mind being for “keying material” in cryptography.

Cryptography is a basic building block for modern computer security and is based upon the use of comparably high quality algorithms and keys.

Either being “weak” has resulted in successful attacks on cryptographic systems.

Over time, cryptographic algorithms and their implementations have continually been improved, as needed (e.g., our AES NI).

Comparably, the availability, quality, and performance of entropy sources have not.

The Platform Entropy Problem

Historically, computing platforms have had a perennial problem of the absence of any high quality/high performance “entropy source”.

Older approaches were almost all based upon the premise that true “raw” entropy accumulation was a very slow process.

Entropy was slowly gathered in small quantities from sources of true entropy (some HW source) at slow rates - in the bits/sec (e.g., key strokes, mouse click timing, disk seek times) up to kilobits/sec (analog ring oscillator-based TRNGs)

As a scarce resource, entropy had to be accumulated (in an entropy pool) and used to seed/reseed a SW PRNG that could cryptographically spread that scarce entropy resource out over numerous requests with acceptable performance.

With little availability of quality entropy early in boot, OSes can have difficulty generating good boot time keys.

The Platform Entropy Problem

OS mediated access to HW entropy sources reduces application or networking performance further.

Many of these SW PRNGs did not met quality standards such as NIST SP 800-90 or were not FIPS 140-2 certified as such.

Such SW PRNGs also have a history of being error prone and easily monitorable/attackable.

As servers become headless (e.g., with no keyboard or mouse) and move to SSDs (instead of disks), platform sources of entropy are going away.

As one moves to virtualized environments, the virtualized OS that thinks it can get at the platform’s HW entropy probably can’t and will suffer further performance loss caused by hypervisor mediation.

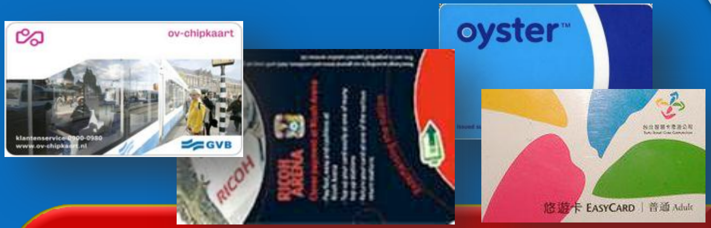
Examples of RNG-based Security Attacks

Debian/OpenSSL Fiasco

Debian has warned of a vulnerability in its cryptographic functions that could leave systems open to attack.

The use of a cryptographically flawed pseudo random number generator in Debian's implementation of OpenSSL meant that potentially predictable keys were generated…

The Register – May 13th, 2008



MiFare Classic Crypto-1

Stream cipher used in about 200 million RFID chips worldwide. 16-bit random numbers generated by LFSR-based RNG. Internal state can be unshifted, filter function can be inverted, limited size enables replay attacks.

BlackHat 2008

Jan 1996 - Mozilla SSL Browser RNG Failure

28th September 1999 - How We Learned to Cheat at Online Poker: A Study in Software Security

- *by Brad Arkin, Frank Hill, Scott Marks, Matt Schmid and Thomas John Walls*

August 2007 – NSA's Dual EC DRBG shown to have backdoor parameters

19th November 2007 – Microsoft Windows Insecure Random Number Generator

- CVE-2007-6043

13th May 2008 - Debian/OpenSSL Fiasco

4th November 2008 - MiFare Classic

29th March 2010 - Weak RNG in PHP session ID generation leads to session hijacking

December 2010 Sony Playstation 3 Jailbreak

RNG-based Attacks Have Happened More Often Than You Think

Serendipity Happens

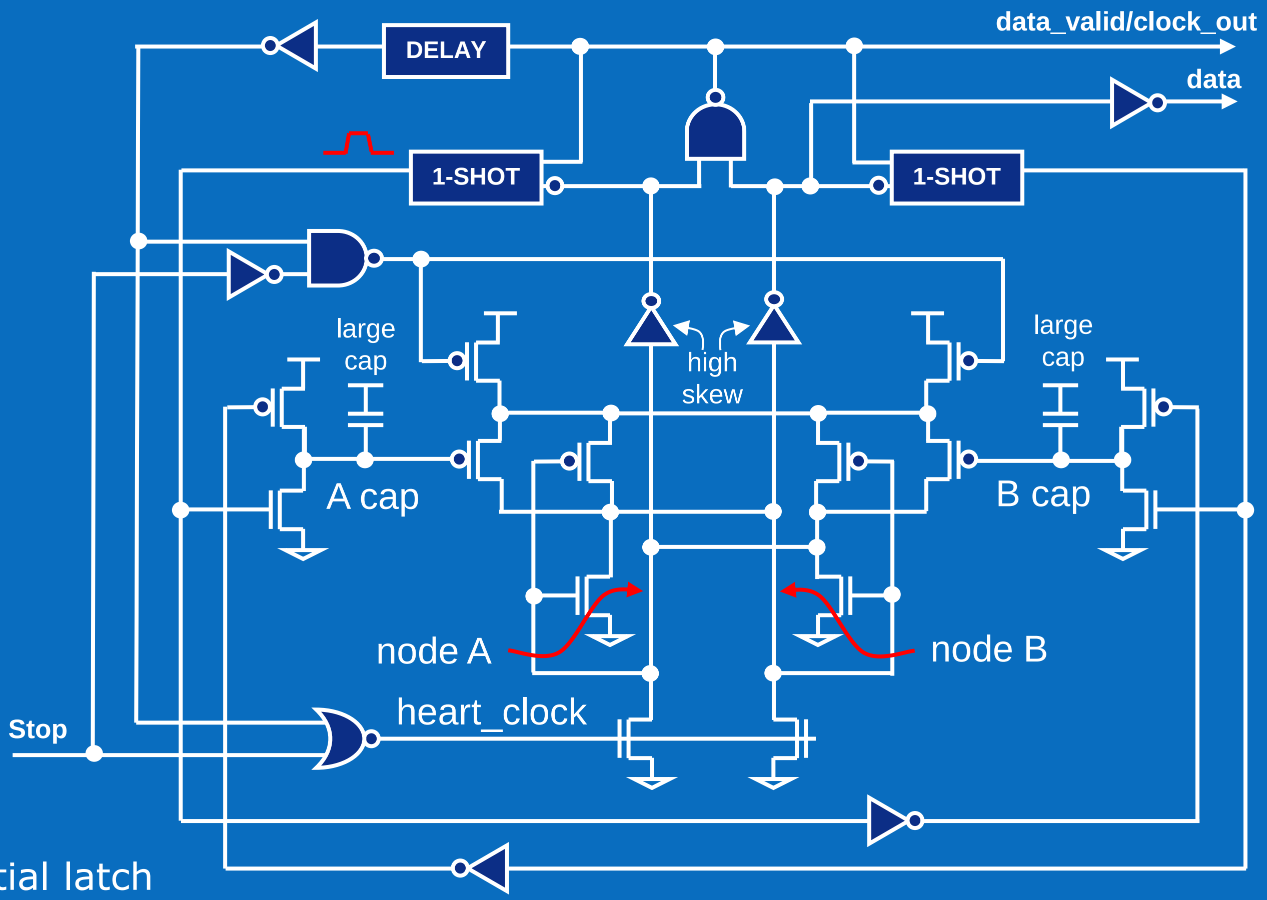
The need for high quality entropy had been established/recognized, but no acceptable solution was available – NO embedding of analog TRNG in Intel processors

Then came a small, but crucial, invention

- Charles Dike and his ***digital*** Entropy Source (or TRNG)
- Which delivers "raw" entropy @ 2-3 gigabits/sec

Given that opportunity, we jumped in to add the necessary downstream and wrapper functionality to support instruction level use

DRNG Entropy Source



Differential latch

Push into metastable state with resolution driven by thermal noise

Dynamic, bilateral, step-based feedback loop to deal with any circuit bias

Designed to be stable across process, temperature, and voltage/power variations

Our Solution

Quite simply, we are in the process of making this long standing security problem “just go away” by deploying a solution (code named Bull Mountain) consisting of two parts:

- A RdRand instruction making entropy ***directly*** available to ALL SW on Intel platforms in any operating state/mode and
- A HW Digital Random Number Generator (DRNG) producing NIST SP 800-90 compliant and FIPS 140-2 certifiable entropy that supplies random numbers to the RdRand instruction.

New Common Brand Promise of Great Entropy on ALL Intel platforms!

SW Interface – The RdRand Instruction

Access to the DRNG is provided to SW through the new RdRand instruction

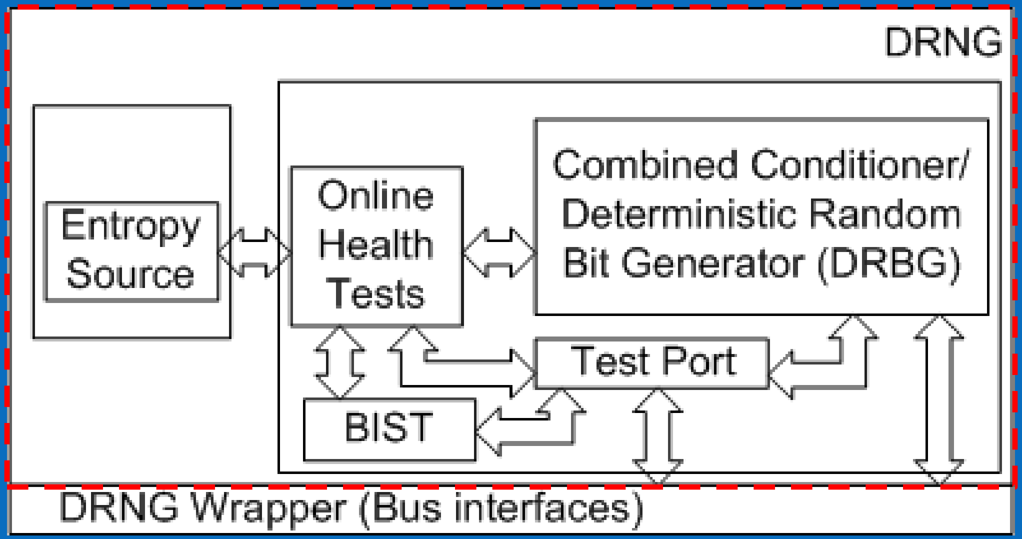
- Intel Advanced Vector Extensions Programming Reference, Chapter 8 (http://software.intel.com/en-us/avx/)) – pages 8-15 and 8-11;

The RdRand Software Implementation Guide (SIG) (http://software.intel.com/en-us/articles/download-the-latest-bull-mountain-software-implementation-guide/); and

RdRand retrieves a hardware generated random value from the DRNG and stores it in the destination register given as an argument to the instruction.

RdRand is available to any system or application software running on the platform. That is, there are no hardware ring requirements that restrict access based on process privilege level. As such, RdRand may be invoked as part of an operating system or hypervisor system library, a shared software library, or directly by an application.

The DRNG


DRNG
Entropy Source
Online Health Tests
Combined Conditioner/ Deterministic Random Bit Generator (DRBG)
Test Port
BIST
DRNG Wrapper (Bus interfaces)

A reusable IP module that provides each embedding with an autonomous/self contained, high quality/high performance, "complete" DRNG

Provides "common brand promise" of high quality, high performance entropy across ALL Intel products

Composed of

- An all-digital Entropy Source (NRBG), runtime entropy quality measurement via Online Health Test (OHT),
- Conditioning (via AES CBC-MAC mode) and DRBGing (via AES CTR mode) post processing and
- Built In Self Test (BIST) and Test Port

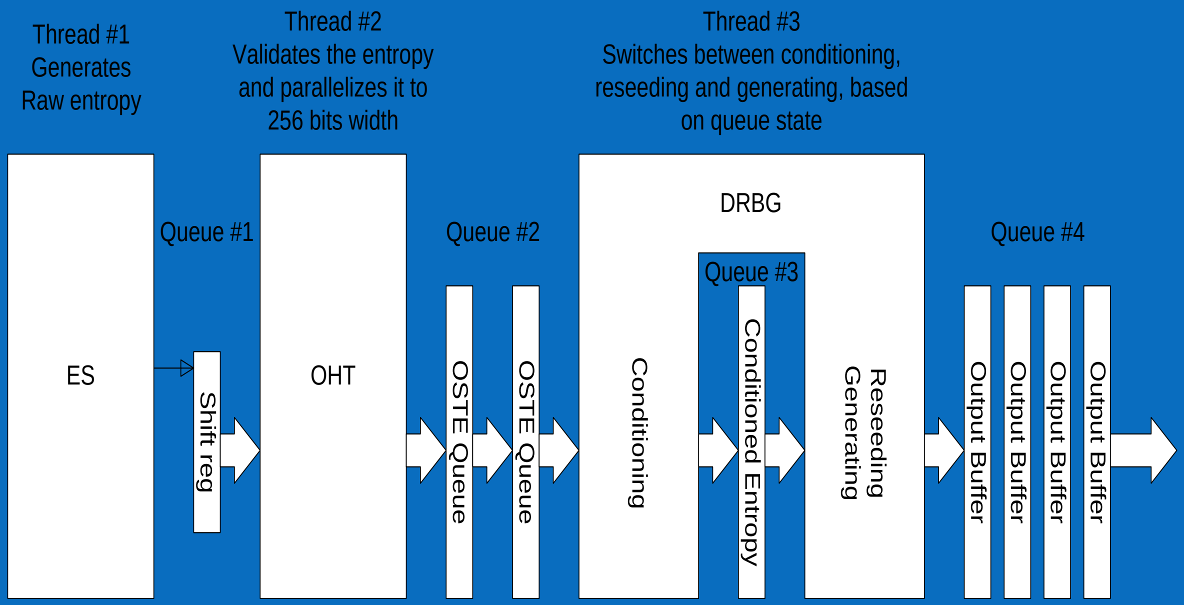
"Standards" compliant (NIST SP 800-90) and FIPS 140-2/3 Level 2 certifiable as such and

Designed for ease of testability, debug, and validation in HVM and in end user platforms

- Comprehensive Built In Self Test (BIST) and
- Test Port (and associated tools) for full pre/post-silicon debug flexibility

Red dotted line is the DRNG's FIPS boundary

Temporal Asynchrony Between Subunits



Our DRNG is logically a three stage/subunit asynchronous production pipeline (composed of the Entropy Source, Online Health Test, and Conditioner and DRBG)

For "flow control" purposes, these subunits each have what amounts to an "output queue" between them and their nearest neighbor in the DRNG production sequence

Depending on the subunit production rate and the next subunit consumption rate, unpredictable dynamic synchronization behaviors ensue

RdRand/DRNG Deployment

RdRand/DRNG incorporation into Ivy Bridge processors and subsequent reuse across ALL of Intel's:

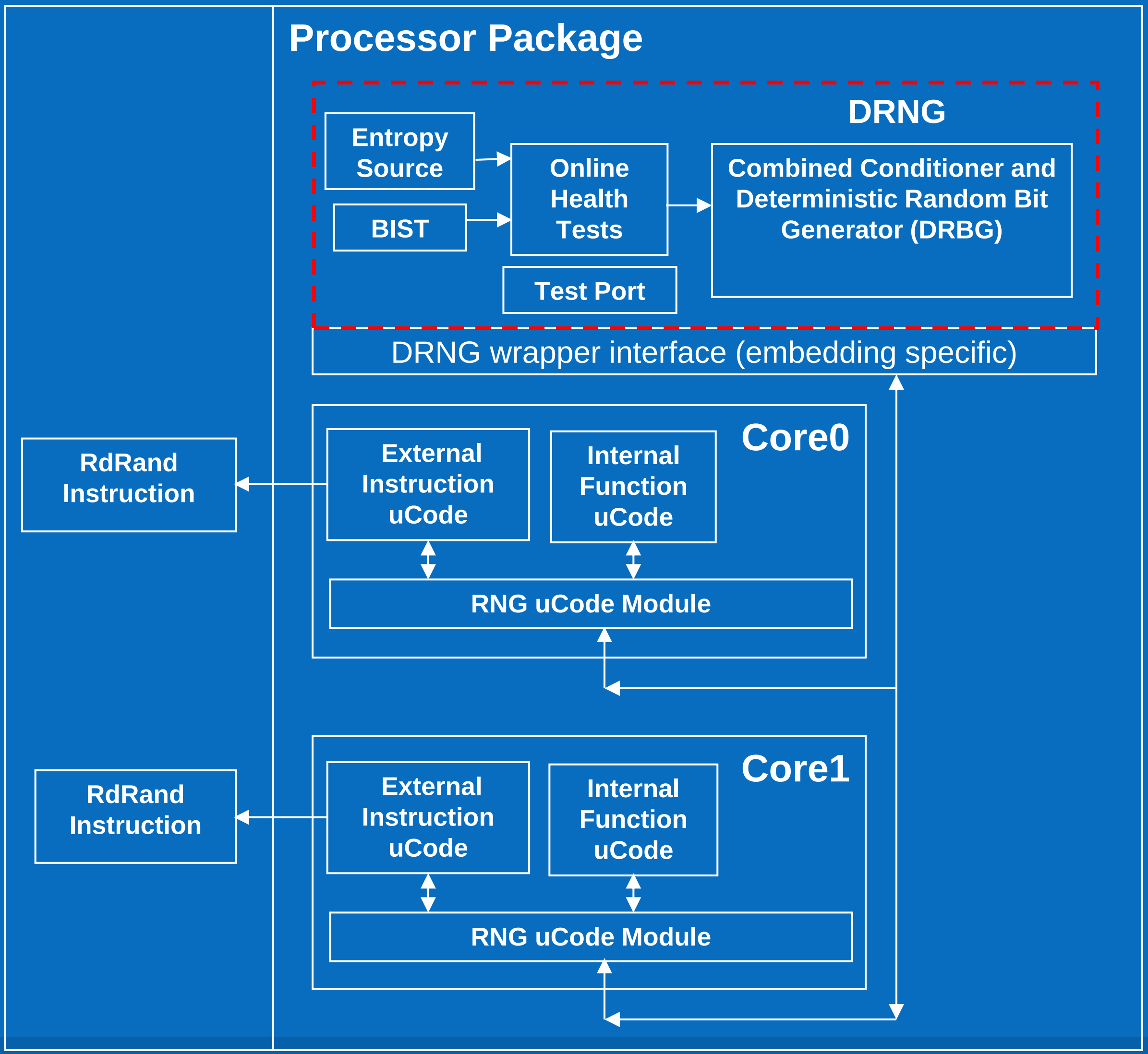
- Large core client (and server) processors
- Small core client (and server) processors (and associated SOCs);
- Client (and server) chipsets;
- Integrated graphics; and
- Throughput computing

summing to 31 product embeddings by EOY'2012

guarantees that we meet our "common brand promise" of consistently delivering high quality/high performance entropy, where needed, across ALL Intel products/platforms.

Ivy Bridge (large core client) is the first Intel product deployment of RdRand/DRNG.

Processor Embedding Example



Provides each processor package with a chipset independent, autonomous/self contained, high quality/high performance, "complete", shared, uncore resident DRNG

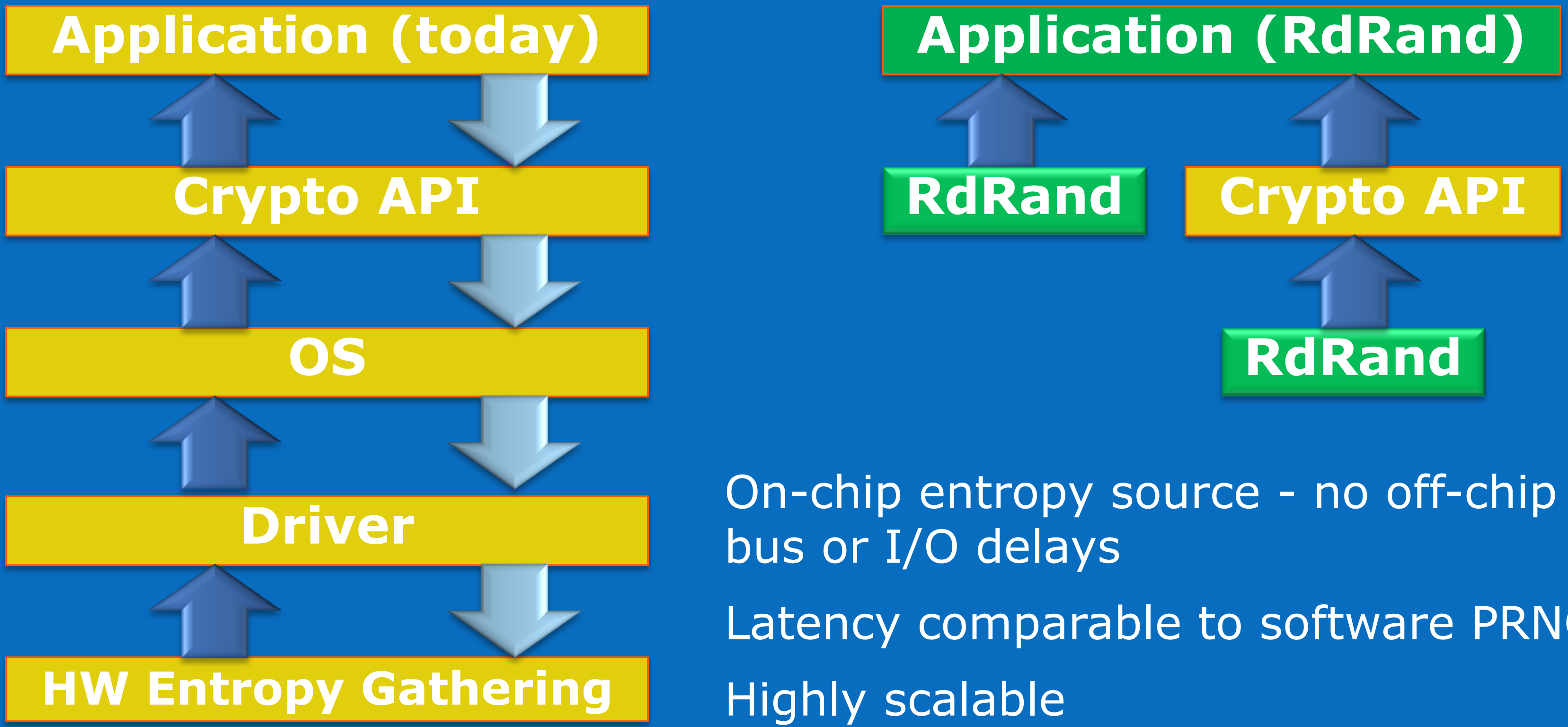
DRNG access by software via RdRand instruction

In any given embedding, the DRNG is shared by multiple users (e.g.,

- Processor ucode (across multiple threads/cores) to implement RdRand;
- PCU;
- GEN/PAVP; and/or
- Programmable elements in SOCs)

Performance

Direct access to random numbers through RdRand bypasses OS, driver, and associated overhead



On-chip entropy source - no off-chip bus or I/O delays

Latency comparable to software PRNGs

Highly scalable

Measured Throughput

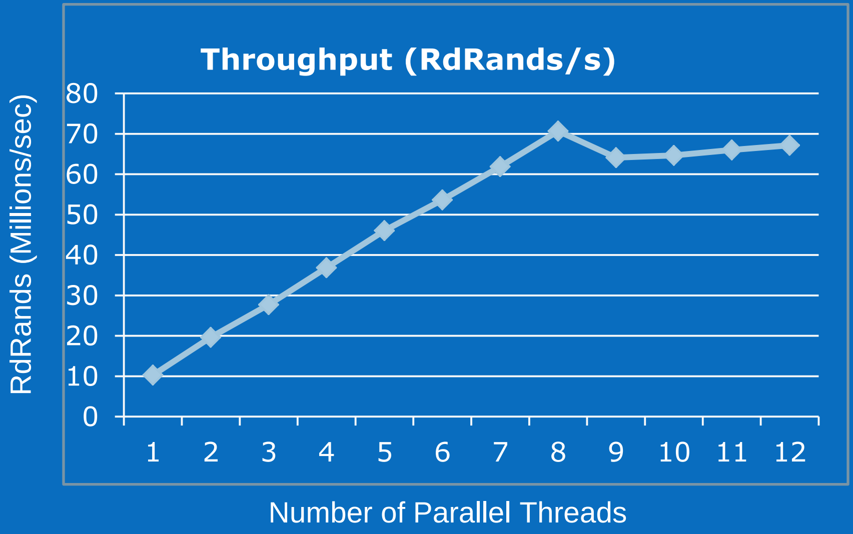
Preliminary data from pre-production Ivy Bridge sample[1]

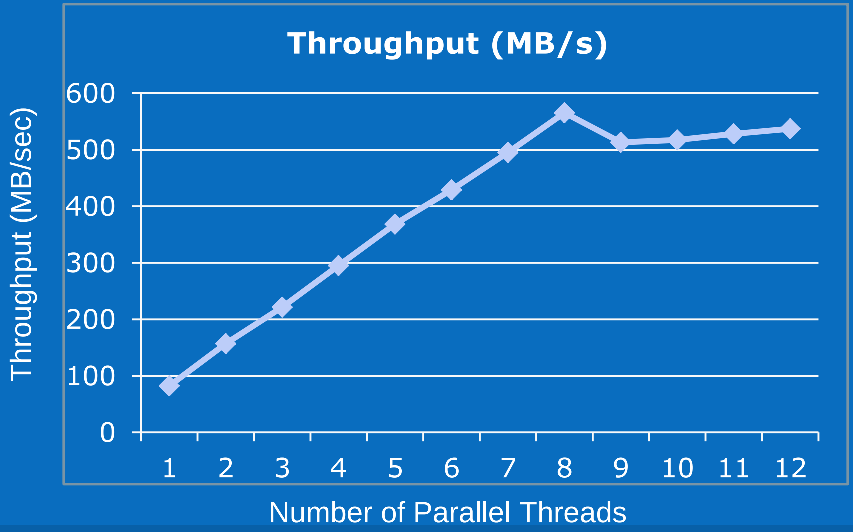
Up to 70 million RdRand invocations per second

500+ Million Bytes of random data per second

Throughput ceiling is insensitive to number of contending parallel threads

❑ Steady state maintained at peak performance





[1]Data taken from Intel® processor codename Ivy Bridge early engineering sample board. Quad core, 4 GB memory, hyper-threading enabled. Software: LINUX* Fedora 14, gcc version 4.6.0 (experimental) with RdRand support, test uses pthreads kernel API

Response Time and Reseeding Frequency

Preliminary data from pre-production Ivy Bridge sample[1]

RdRand Response Time

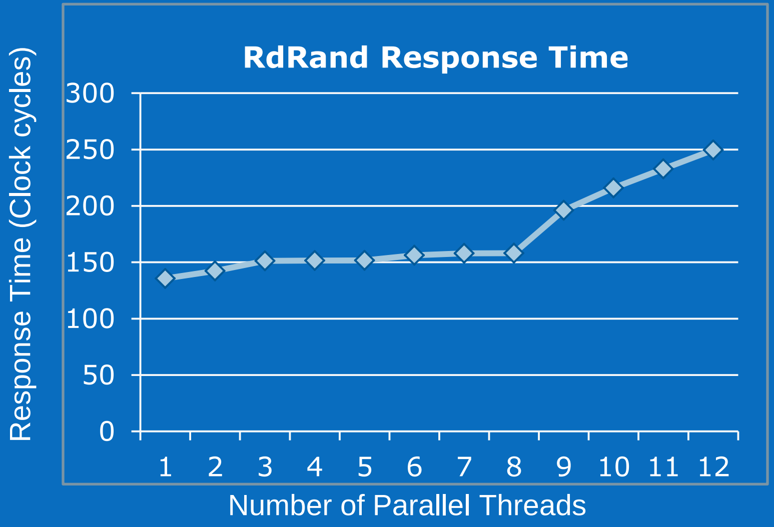
~150 clocks per invocation

(Note: Varies with CPU clock frequency since constraint is shared data path from DRNG to cores.)

Little contention until 8 threads

- (or 4 threads on 2 core chip)

Simple linear increase as additional threads are added



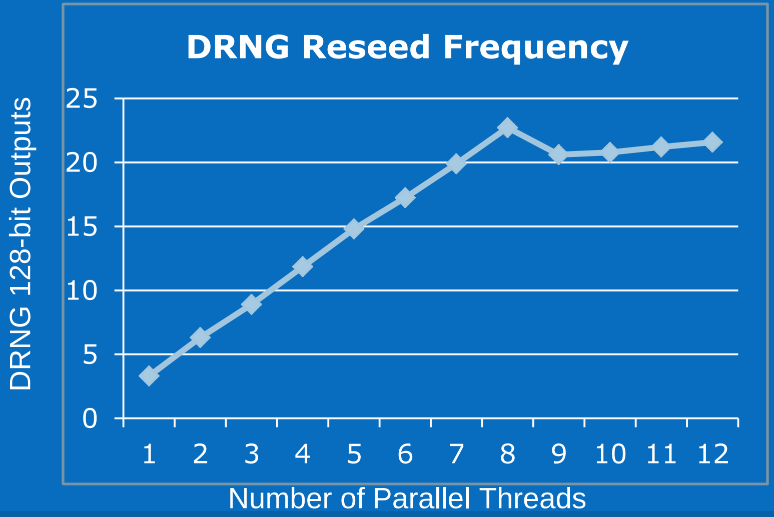
DRNG Reseed Frequency

Single thread worst case: Reseeds every 4 RdRand invocations

Multiple thread worst case: Reseeds every 23 RdRand invocations

At slower invocation rate, can expect reseed before every 2 RdRand calls

❑ NIST SP 800-90 recommends ≤ 2^{48}



[1]Data taken from Intel® processor codename Ivy Bridge early engineering sample board. Quad core, 4 GB memory, hyper-threading enabled. Software: LINUX* Fedora 14, gcc version 4.6.0 (experimental) with RdRand support, test uses pthreads kernel API

DRNG Performance Learnings

Initial processor-based embeddings used a “shared” DRNG module located in the processor uncore connected to the processor cores through a “shared” interconnect

The “shared” interconnects employed (i.e., Message Channel and IOSF) resulted in huge latency penalties (e.g. 2-400 uncore clocks) per reference even on unloaded configurations executing back-to-back RdRands

Ivy Bridge has implemented “a one off alternative” interconnect in order to cut this latency down to ~100 uncore clocks

This points to migrating the DRNG closer to is main/latency sensitive consumers:

- Closer to the processors on a wider/faster “shared” bus or
- Into the processor as the DRNG becomes ever smaller by:
  - Process related shrinkage and/or
  - A slower (but smaller) DRNG design that would deliver faster overall results by avoiding interconnect latency

DRNG Performance Learnings

We have to decide:

- What is “reasonable RdRand/RdSeed performance” and
- How can we place a platform’s DRNG(s) as to guarantee necessary and predictable performance.

Note that the basic DRNG itself can produce an 8 byte (64 bit) output every 8 clocks in whatever environment you embed it => lower bound on RdRand latency is 8 clocks.

The one accepted “selling justification” is our long stated goal of making RdRand ~as fast as alternative and comparable functionality SW PRNGs => then we can motivate SW PRNG users/developers to replace cryptographic SW PRNGs with RdRand use.

Thus, we are creating a “functionally equivalent” SW PRNG to run, under load, for comparison (e.g., a NIST SP800-90A AES-CTR DRBG SW PRNG using AES-NI) (e.g., OpenSSL)

We intend to have such “processor clocks per SW PRNG invocation” numbers shortly.

Summary

We described the “platform entropy problem” and showed how we are in the process of making this long standing security problem “just go away” - Randomness anywhere anytime!

We are succeeding in getting RdRand/DRNG ***deployed*** across ALL Intel HW products!

Collaborate with us on getting the resultant high quality/high performance entropy widely ***used***, wherever needed, in your SW products

Acknowledgements

The original RdRand/DRNG architecture team - Ernie Brickell, James Coke, George Cox, Charles Dike, Martin Dixon, Steve Fischer, Ed Gamsaragan, Shay Gueron, Howard Herbert, DJ Johnston, Greg Piper, Guna Thuraisingham, Jesse Walker

The DRNG design/implementation team - George Cox, Charles Dike, and DJ Johnston

The integration teams across 30 product embeddings

The product enabling teams working with OSVs/ISVs