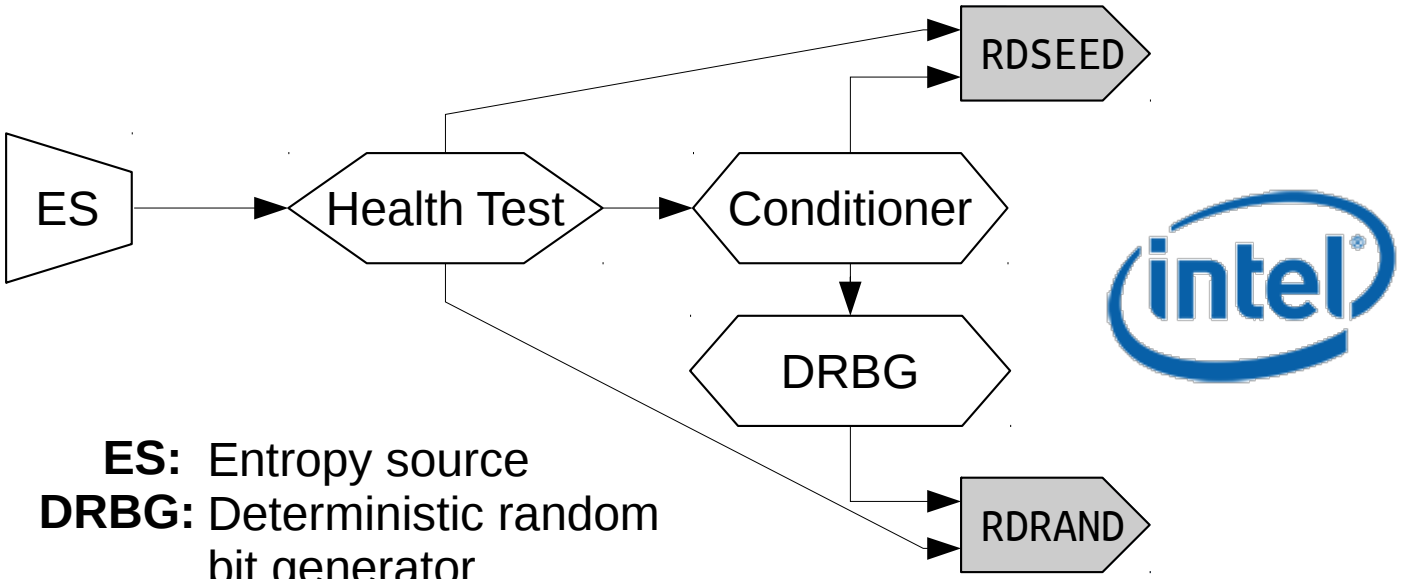
A Provable Security Analysis of Intel's Secure Key RNG

Thomas Shrimpton & **Seth Terashima**
Portland State University

The Intel RNG

- New hardware random-number generator on all recent Intel chips (Ivy Bridge +)
- Two new instructions:
  - **RDRAND:** Fetch pseudo-random bits
  - **RDSEED:** Fetch “truly random” bits (Broadwell +)

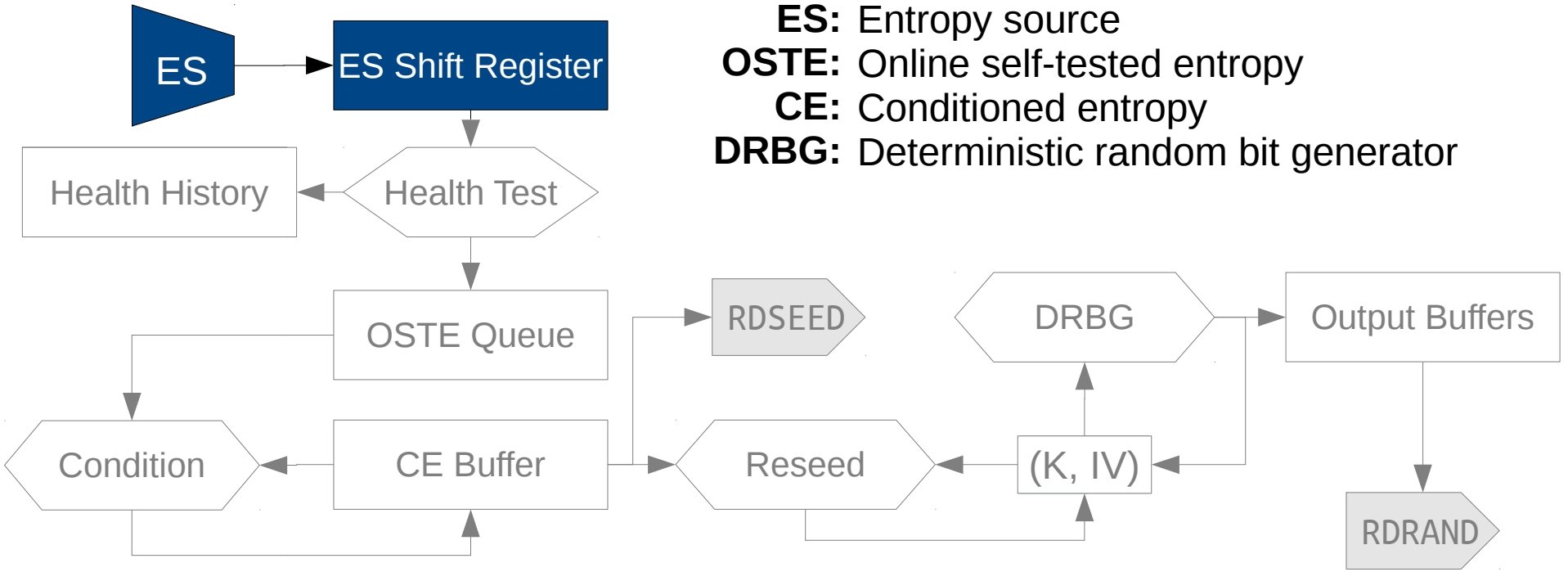


Agenda

- **Intel's Secure-Key RNG design**
- The Model : “PRNGs With Input” (PWIs)
- Analsyis

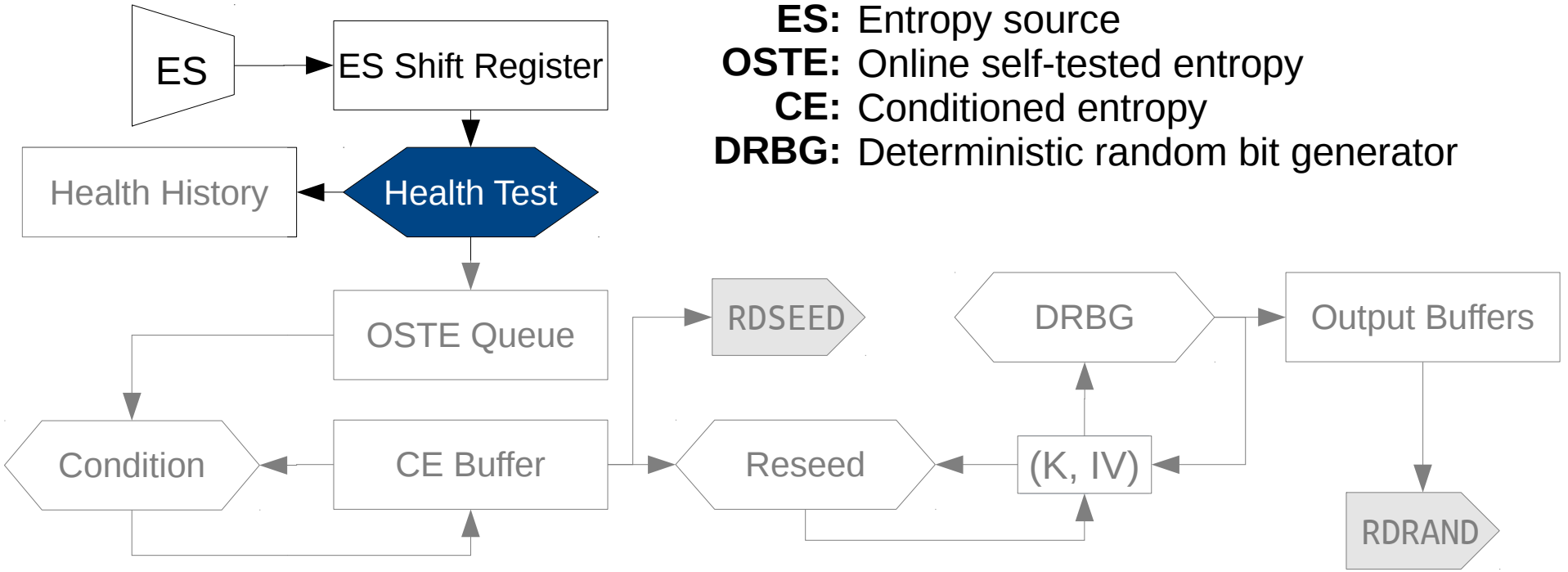
Not on the Agenda:

Entropy Source



- Uses thermal noise to generate random bits
- Analysis of empirical data by Cryptographic Research, Inc. (Hamburg, Kocher, Marson '12)
- 256-bit samples buffered in shift register

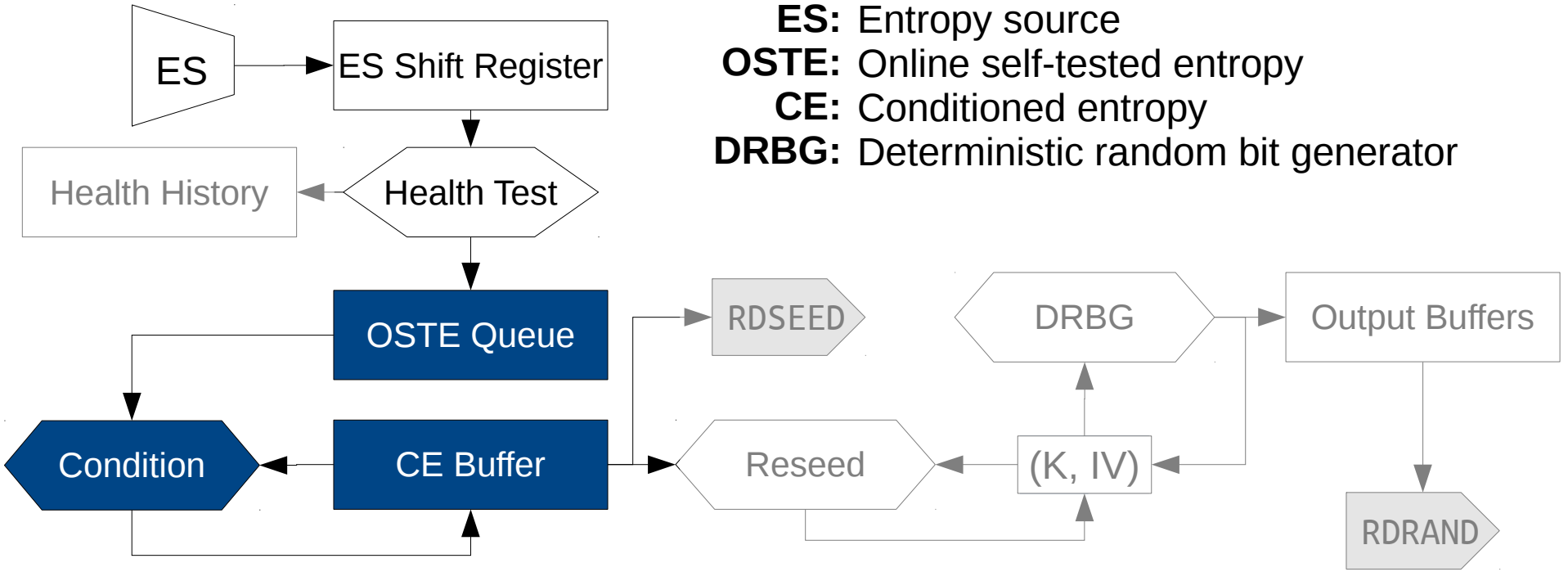
Health Tests



ES: Entropy source
OSTE: Online self-tested entropy
CE: Conditioned entropy
DRBG: Deterministic random bit generator

- Heuristic tests for catastrophic ES failure
- 1% false-positive rate on ideal random source

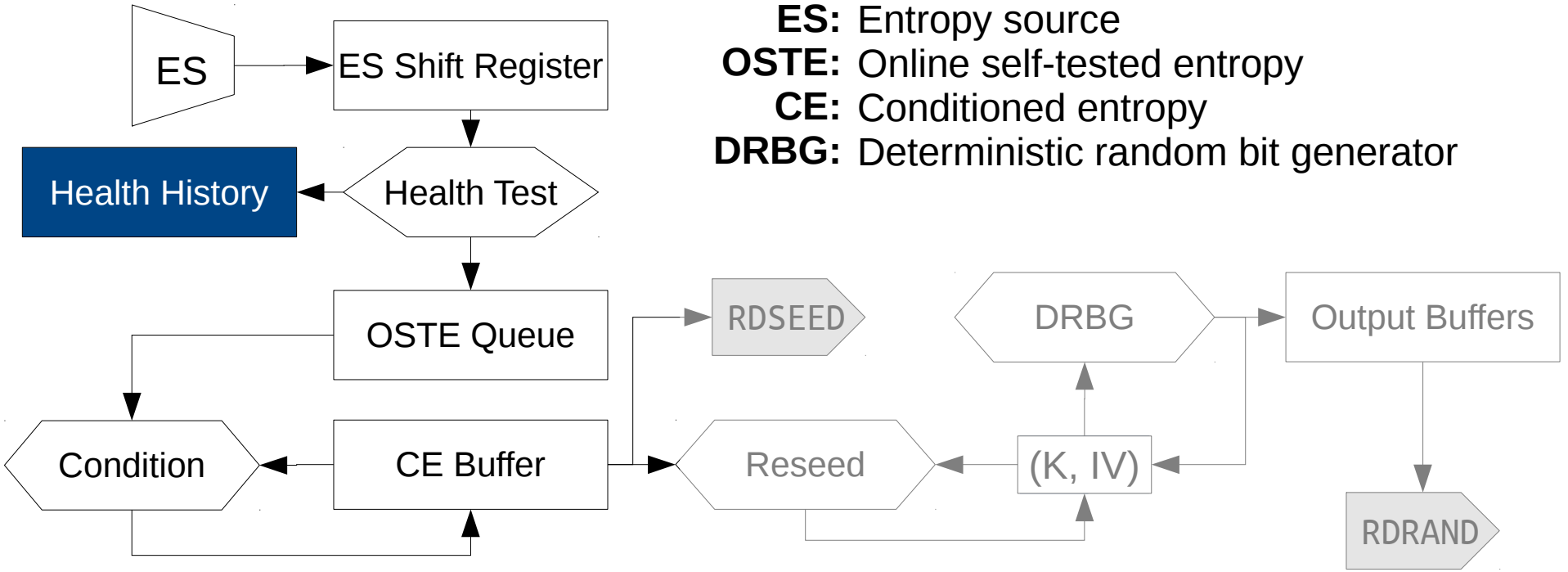
Conditioning



ES: Entropy source
OSTE: Online self-tested entropy
CE: Conditioned entropy
DRBG: Deterministic random bit generator

- ES bits are assumed to be biased, correlated
- Fed into streaming CBC-MAC computation to “condition” them into (hopefully) uniform random bits

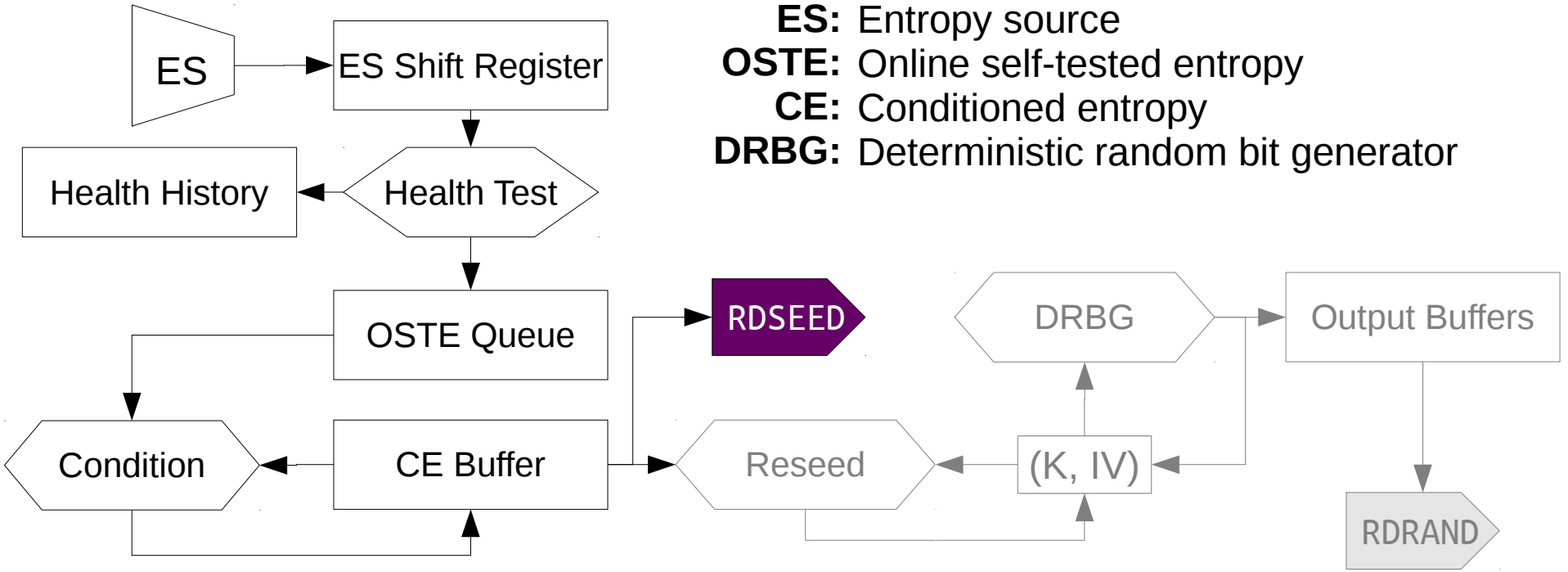
Health History



ES: Entropy source
OSTE: Online self-tested entropy
CE: Conditioned entropy
DRBG: Deterministic random bit generator

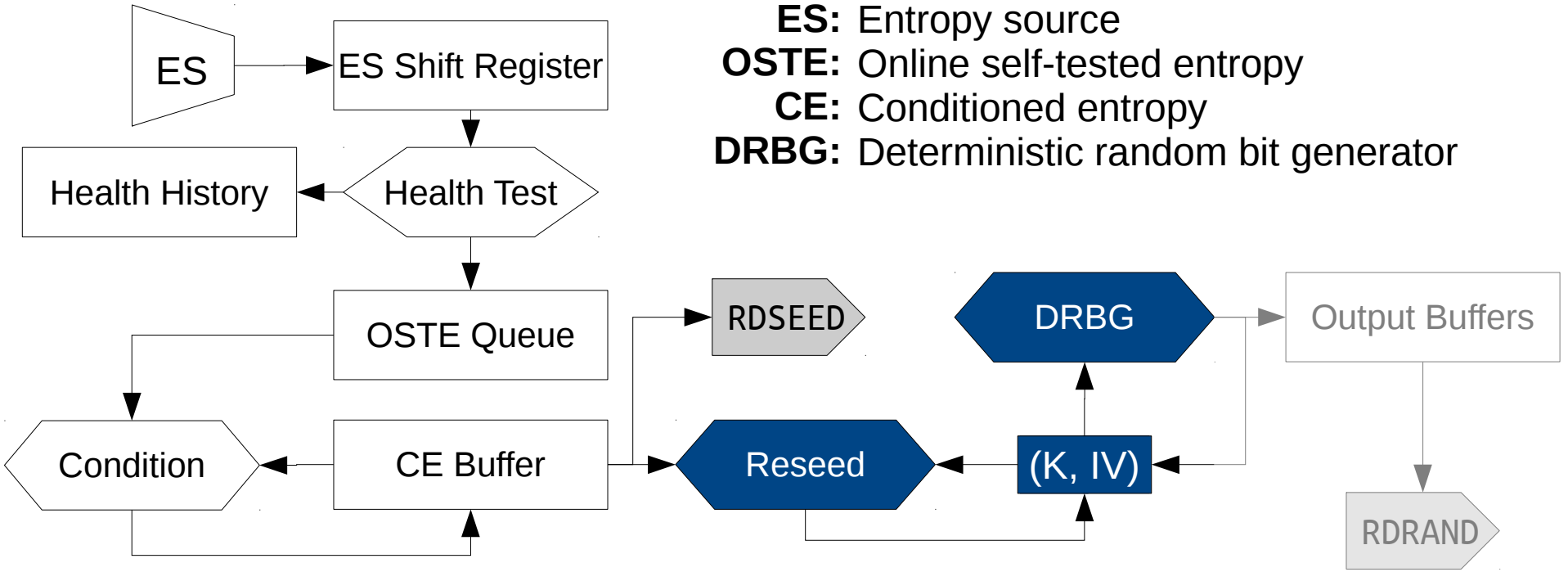
- At least two/three (Ivy Bridge/Broadwell) healthy samples needed before CE buffer is “available”
- But samples don't count unless at least half of the past 256 samples were healthy

RDSEED



- RDSEED instruction grabs bits from CE buffer
- Buffer not cleared, but flagged as “unavailable”
- Will be made available again after sufficient number of healthy samples generated, conditioned

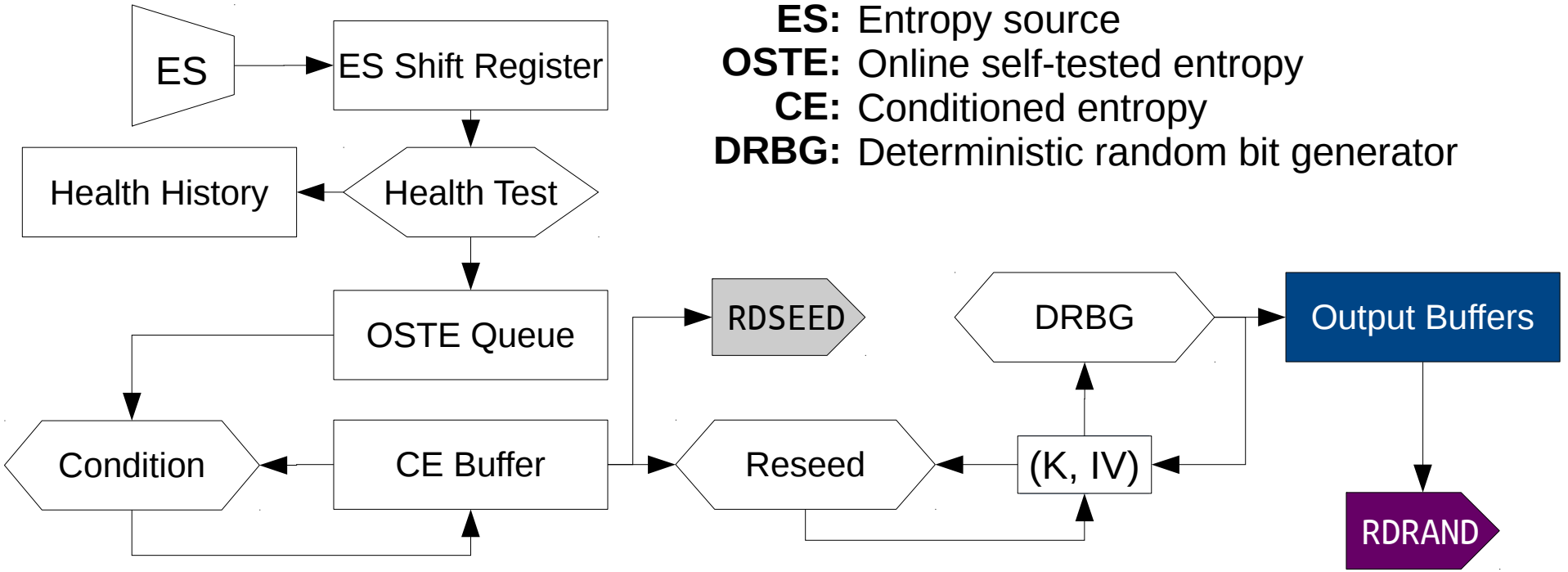
(Re)Seeding the DRBG



ES: Entropy source
OSTE: Online self-tested entropy
CE: Conditioned entropy
DRBG: Deterministic random bit generator

- CE buffer also used to reseed traditional deterministic PRNG (CTR-AES based)
- Reseeding makes CE buffer unavailable

(Re)Seeding the DRBG



ES: Entropy source
OSTE: Online self-tested entropy
CE: Conditioned entropy
DRBG: Deterministic random bit generator

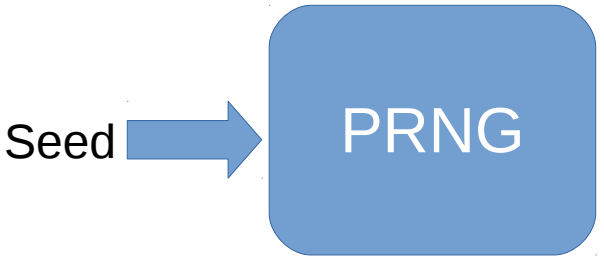
- PRNG outputs buffered
- Read by RDRAND instruction
- At most 64Kb generated between reseeds

Agenda

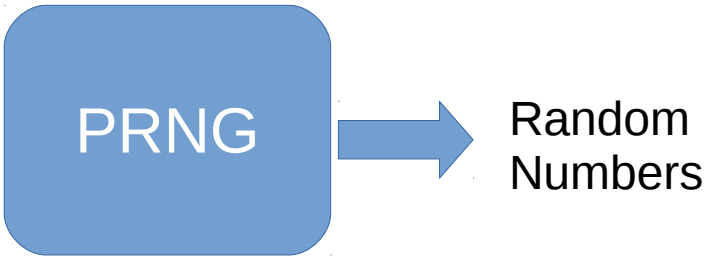
- Intel's Secure-Key RNG design
- **The Model : “PRNGs With Input” (PWIs)**
- Analsyis

PRNGs (the traditional view)

Step 1. Provide seed.



Step 2. Get random numbers.

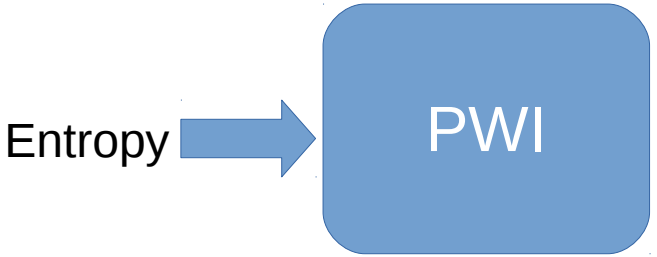


This isn't how

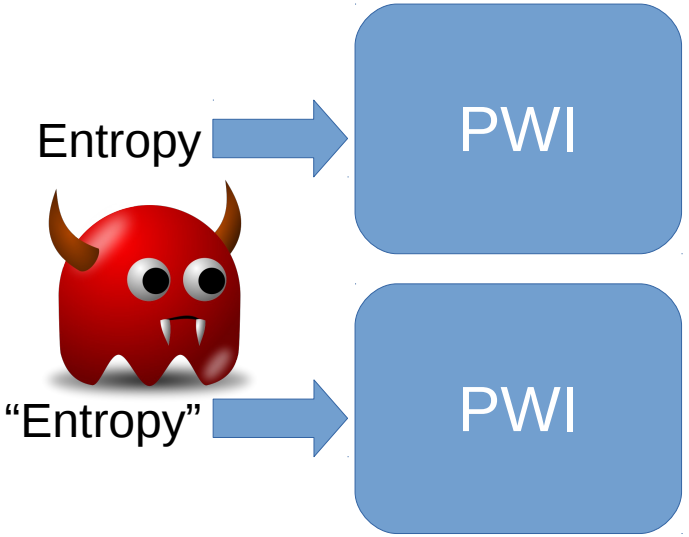
{`/dev/[u]random`, OpenSSL RNG, `RDRAND`}

work.

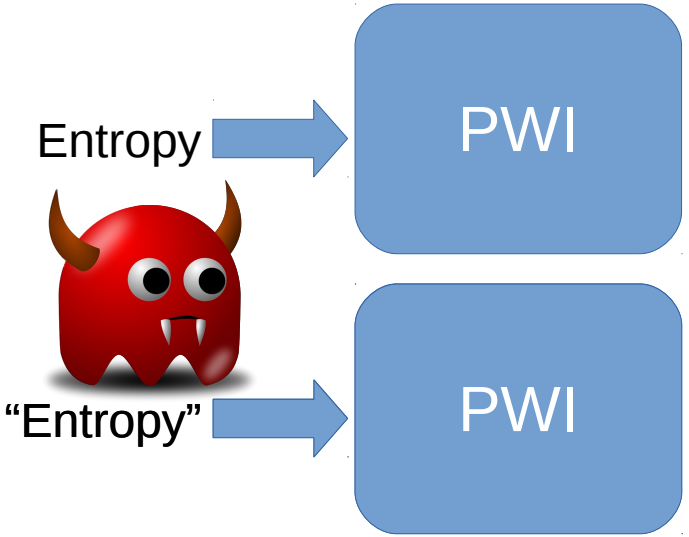
PRNGs with Input



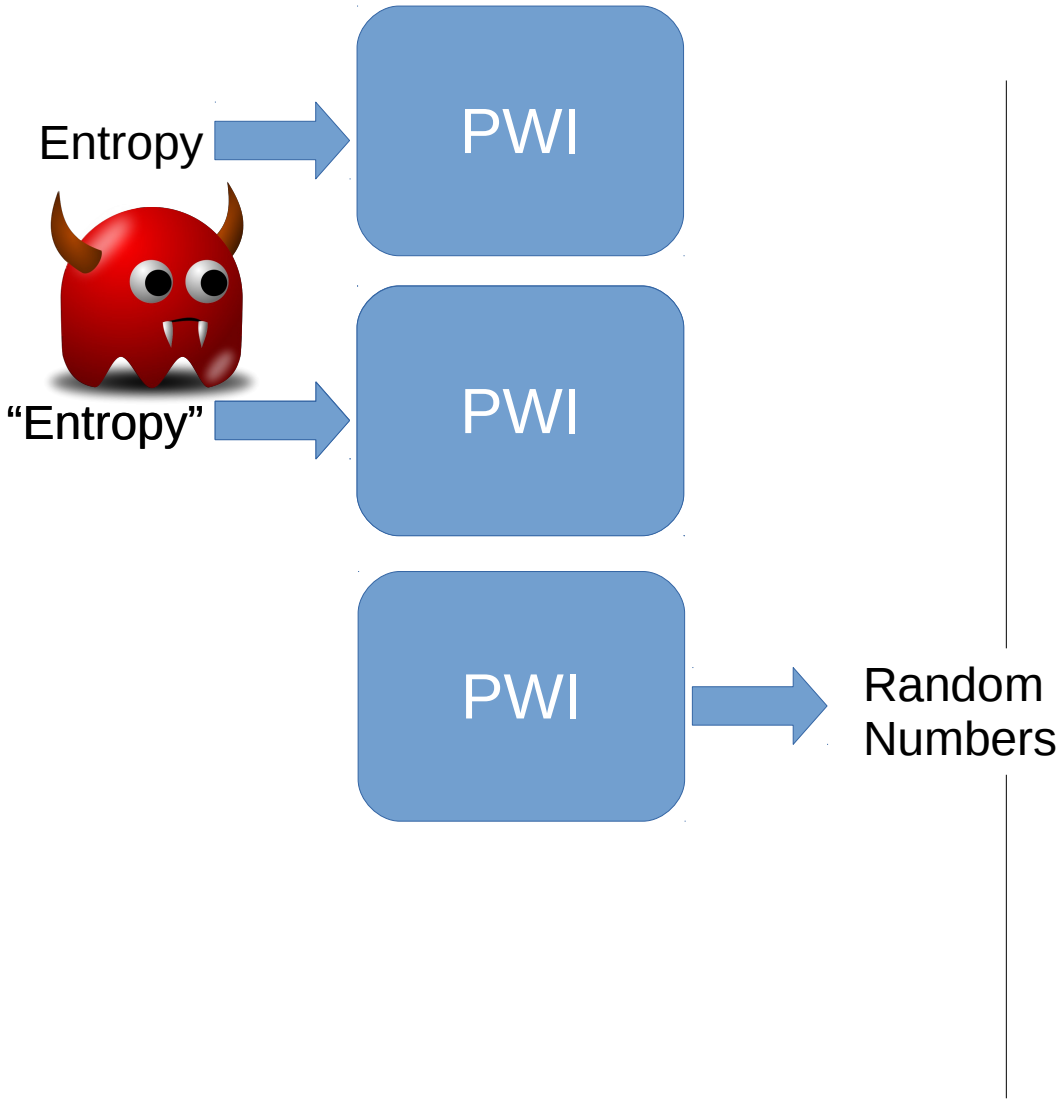
PRNGs with Input



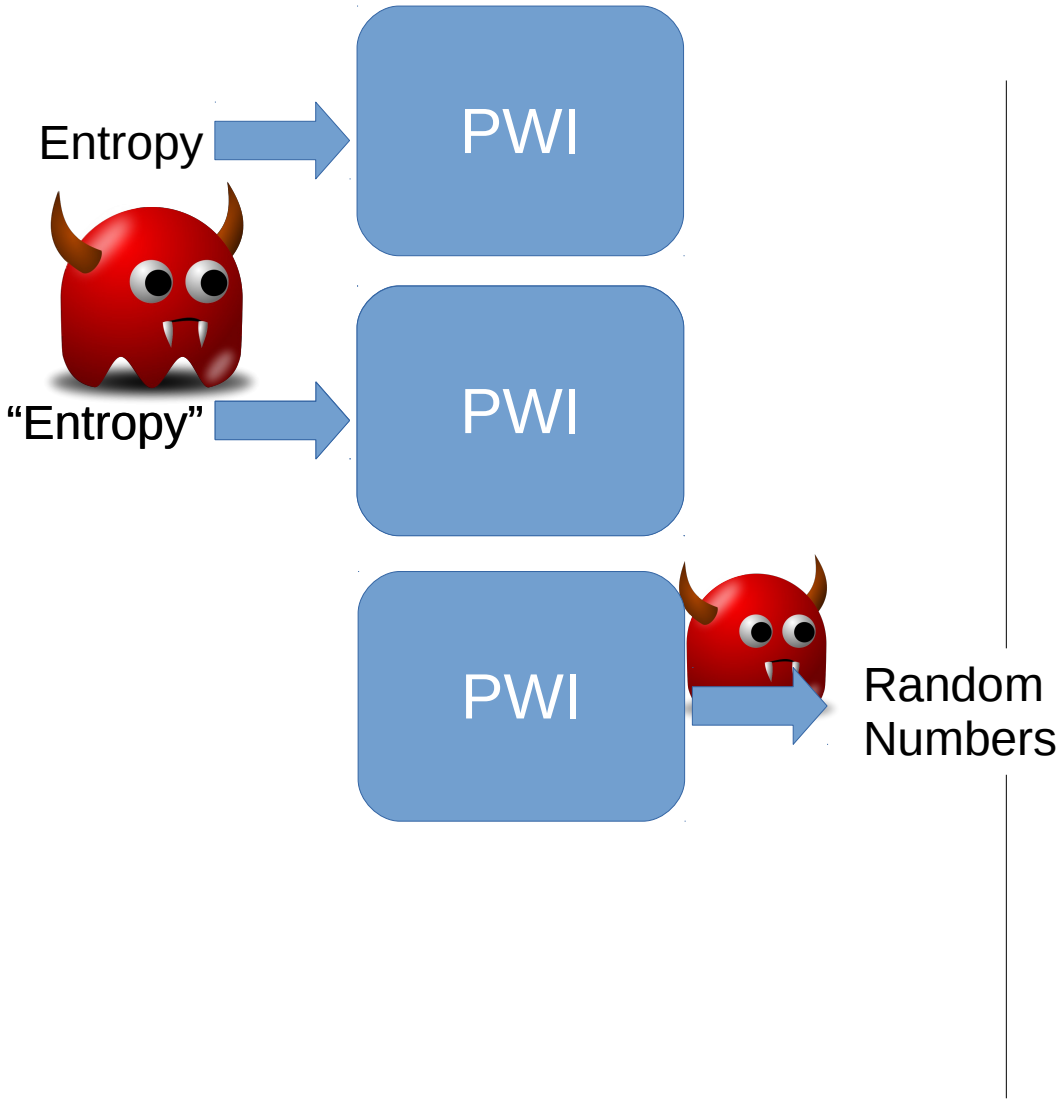
PRNGs with Input



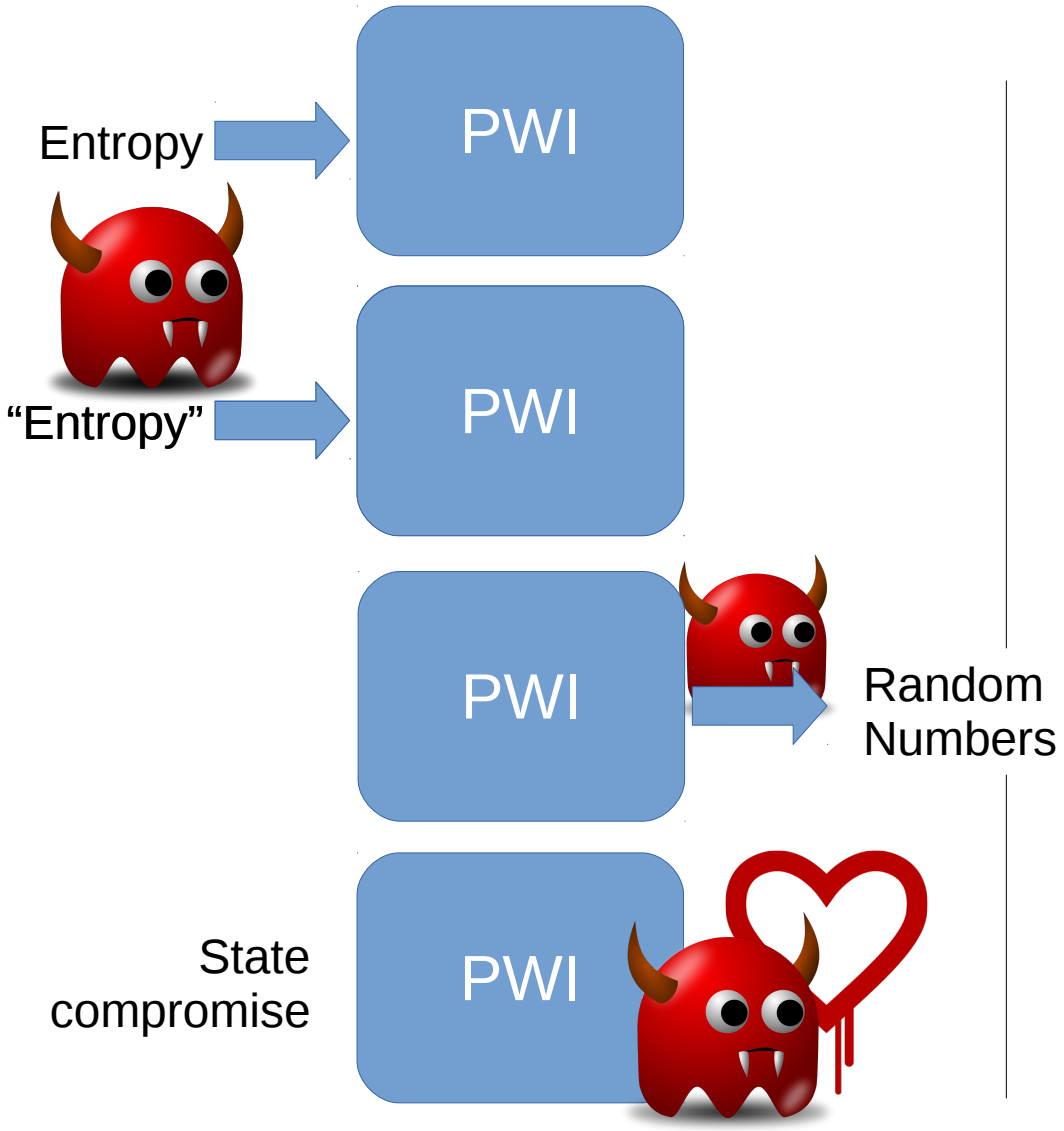
PRNGs with Input



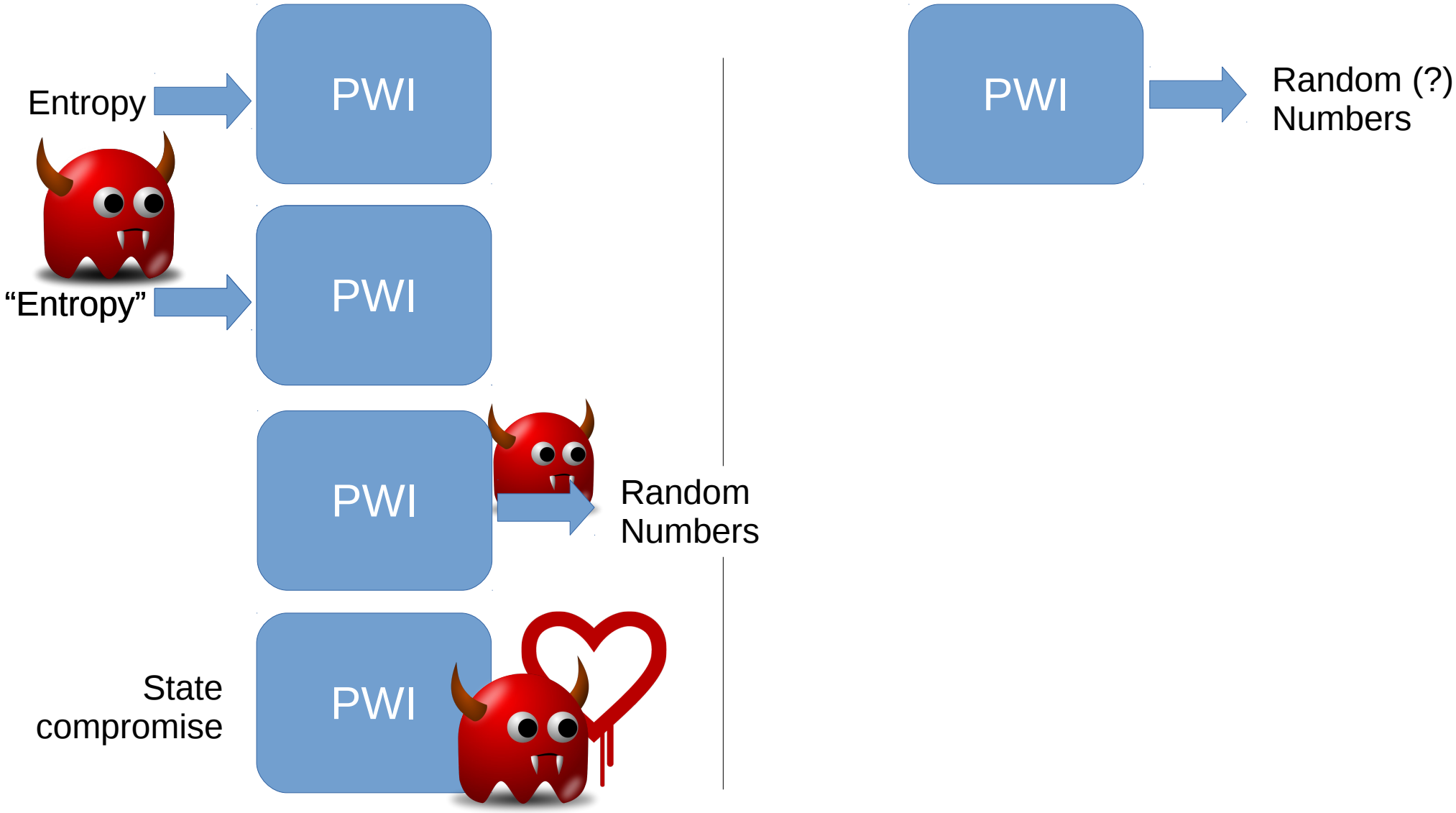
PRNGs with Input



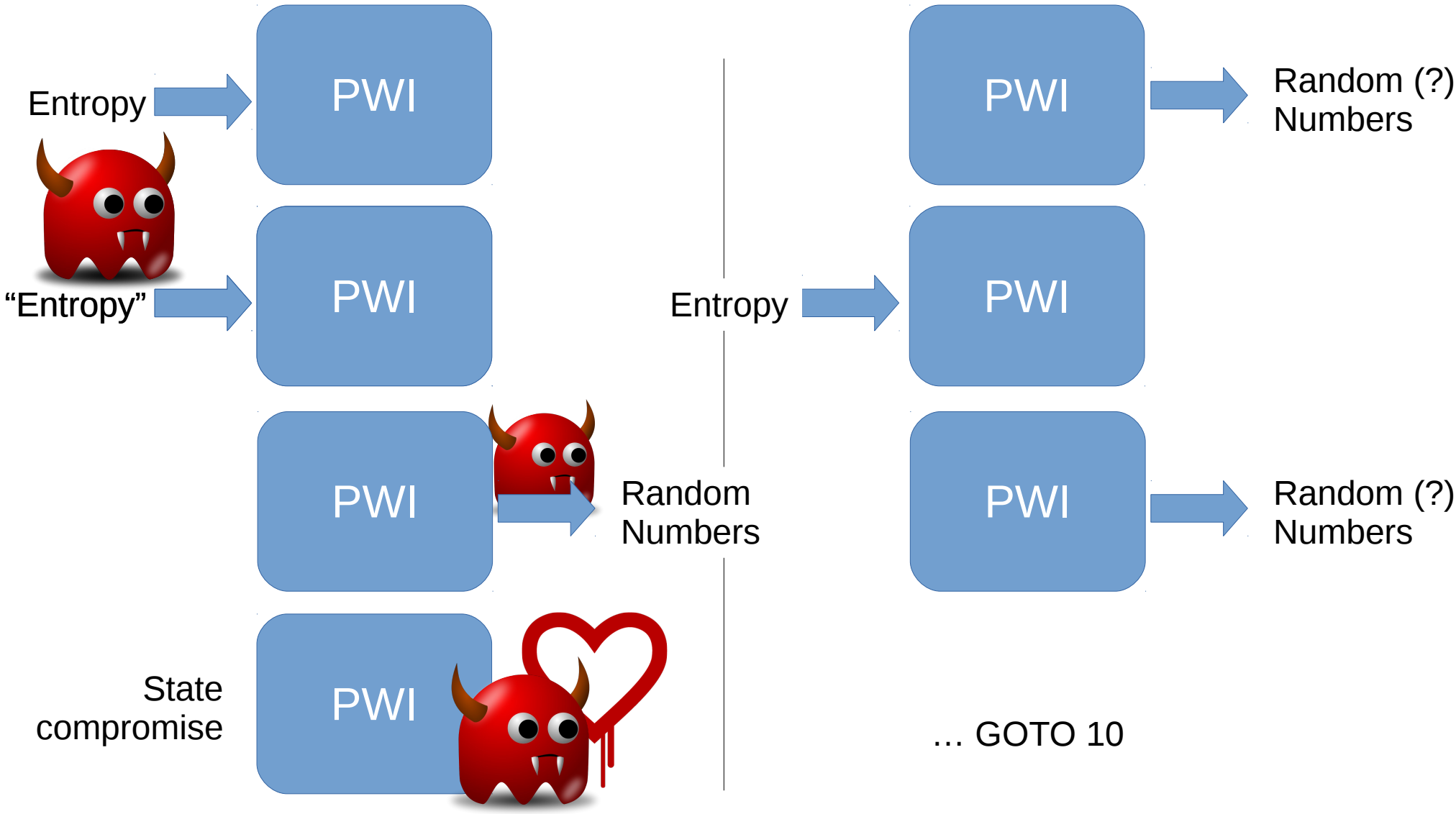
PRNGs with Input



PRNGs with Input



PRNGs with Input



Types of security

Resilience: Basic security. No state compromise, everything looks random

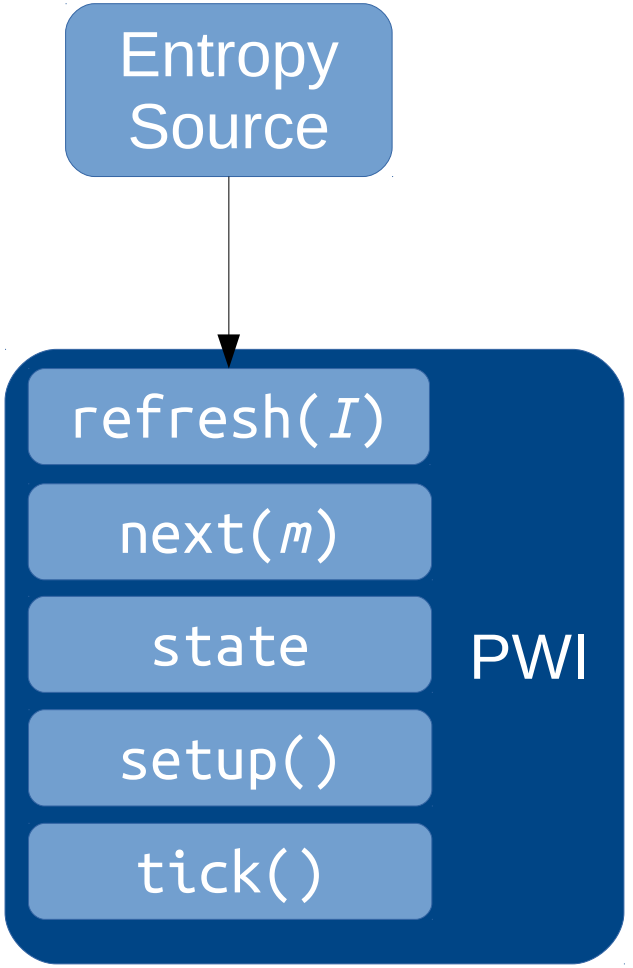
Forward security: Random values are safe even if PWI state is compromised in the future.

Backward security: Random values are safe even if PWI state was compromised in the past (as long as we've harvested enough entropy since then).

Robustness: Both forward and backward security, even if the adversary can *tamper* with state.

PWI Primitive

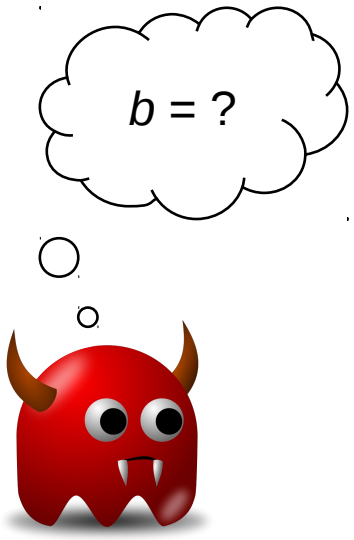
- Started with PWI model of [DPRVW'13]
- Some simple extensions:
  - Blocking
  - Multiple interfaces
  - Explicit setup
  - Asynchronous behavior



PWI Oracles

Challenger runs `setup()`, flips coin ***b***.

- Heads: API connects to real PWI
- Tails: API connects to idealized version of PWI



Entropy Source

ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

`refresh(I)`

`next(m)`

`state`

`setup()`

`tick()`

PWI

PWI Oracles

Grabs bits from entropy source, feeds them to PWI. Leaks some side-channel info to attacker.

Entropy Source

ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

refresh(*I*)

next(*m*)

state

setup()

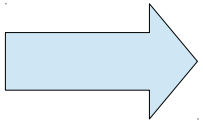
tick()

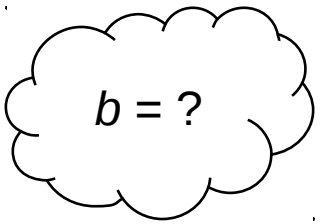
PWI

b = ?

PWI Oracles

Calls `next()`. Returns the result.





ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

Entropy Source

`refresh(I)`

`next(m)`

`state`

`setup()`

`tick()`

PWI

PWI Oracles

Calls `next()`.
- Returns result if $b = 0$.
- Returns random string if $b = 1$.

$b = ?$

ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

Entropy Source

PWI

`refresh(I)`

`next(m)`

`state`

`setup()`

`tick()`

PWI Oracles

Returns the PWI state.

ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

Entropy Source

PWI

refresh(*I*)

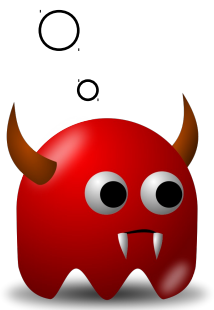
next(*m*)

state

setup()

tick()

b = ?



PWI Oracles

Sets the PWI state to a value specified by the Adversary.

Entropy Source

ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

refresh(*I*)

next(*m*)

state

setup()

tick()

PWI

b = ?

PWI Oracles

Prompts the PWI to perform next scheduled atomic task. No return value.

ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

Entropy Source

PWI

refresh(*I*)

next(*m*)

state

setup()

tick()

b = ?

PWI Oracles

The `get-state` and `set-state` oracles make the state “corrupt”.

ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

Entropy Source

PWI

`refresh(I)`

`next(m)`

`state`

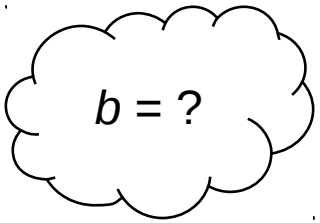
`setup()`

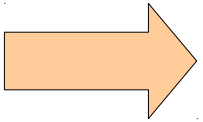
`tick()`

b = ?

PWI Oracles

While the state is corrupt, the Adversary is cut off from the `next-ror` oracle.





ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

Entropy Source

PWI

`refresh(I)`

`next(m)`

`state`

`setup()`

`tick()`

PWI Oracles

The state remains corrupt until the PWI harvests a specified amount of entropy.

ES-Refresh( )

get-next( )

next-ror( )

get-state( )

set-state( )

wait( )

Entropy Source

PWI

refresh(*I*)

next(*m*)

state

setup()

tick()

b = ?

Measuring security

Attacker's advantage is:

$$\mathbf{Adv}^{\mathrm{xxx}}_{\mathsf{PWI}}(A) = |\Pr[\, A \Rightarrow 1 \mid b = 1 \,] - \Pr[\, A \Rightarrow 1 \mid b = 0 \,]|$$

$$\mathbf{Adv}^{\mathsf{RES}}_{\mathsf{PWI}}(A), \mathbf{Adv}^{\mathsf{FWD}}_{\mathsf{PWI}}(A), \mathbf{Adv}^{\mathsf{BWD}}_{\mathsf{PWI}}(A), \mathbf{Adv}^{\mathsf{ROB}}_{\mathsf{PWI}}(A)$$

Secure `setup()`

- `setup()` should place the PWI in a “good” state
  - Some state is sensitive
  - Other state is not (counters, buffered entropy, etc.)
- Define a **masking function** *M* such that *M(S)* is a “good version” of *S*.

State produced by `setup()`

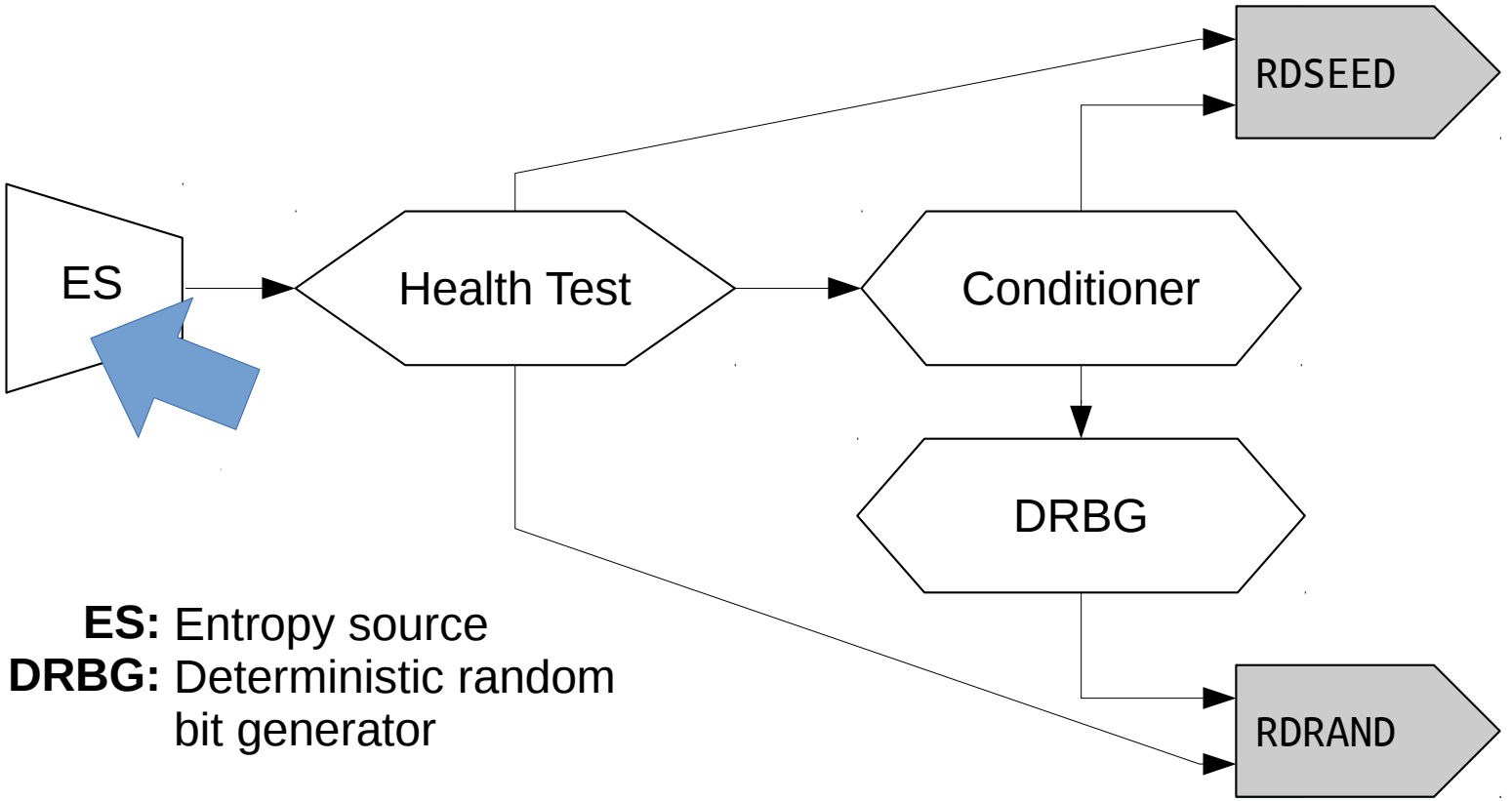
$$\mathbf{Adv}^{\mathsf{init}}_{M}(A) = |\Pr[\, A(S_0) \Rightarrow 1 \,] - \Pr[\, A(M(S_0)) \Rightarrow 1 \,]|$$

$$\mathbf{Adv}^{\mathsf{FWD}}_{\mathsf{PWI}}(A) \leq \mathbf{Adv}^{\mathsf{init}}_{M}(B) + \mathbf{Adv}^{\mathsf{FWD}/M}_{\mathsf{PWI}}(A)$$

Forward security when starting from masked state

Agenda

- Intel's Secure-Key RNG design
- The Model : “PRNGs With Input” (PWIs)
- **Analsyis**



ES: Entropy source
DRBG: Deterministic random bit generator

Entropy Source

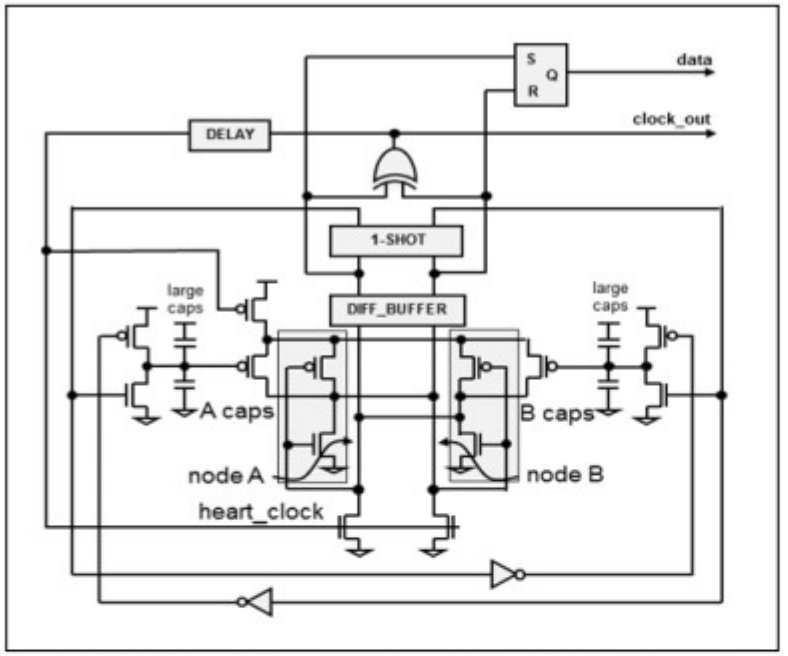


Figure 2: Entropy source for the Intel RNG (from [8])

The ES is a dual differential jamb latch with feedback. It is a latch formed by two cross-coupled inverters (nodes A and B). The circuit is self-clocking (heart_clock), and designed such that when the clock is running, the circuit enters a metastable state. The circuit then resolves to one of two possible states, determined randomly by thermal noise in the system. The settling of the circuit is biased by the differential in the charges on the capacitors (A caps and B caps). The state to which the latch resolves is the random bit of output of the ES.

The circuit is also designed with feedback to seek out its metastable region. Based on how the latch resolves, a fixed amount of charge is drained from one capacitor and added

From CRI Report (Hamburg, Kocher, Marson '12)

Entropy Source

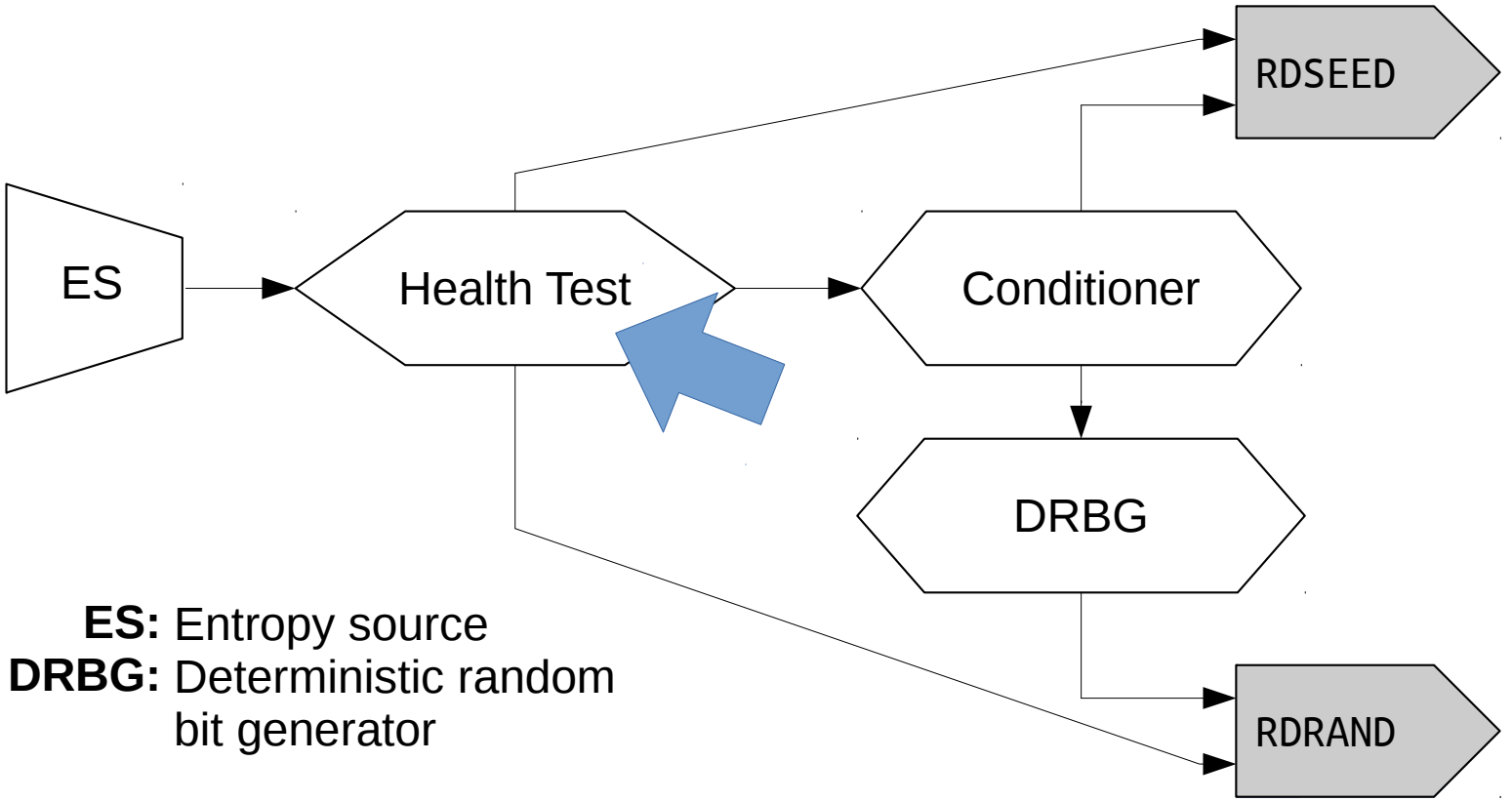
Figure 2: ~~Entropy source for the Intel RNG (from [8])~~

Magic

The ES is a magic black box that produces bits with some amount of min-entropy.

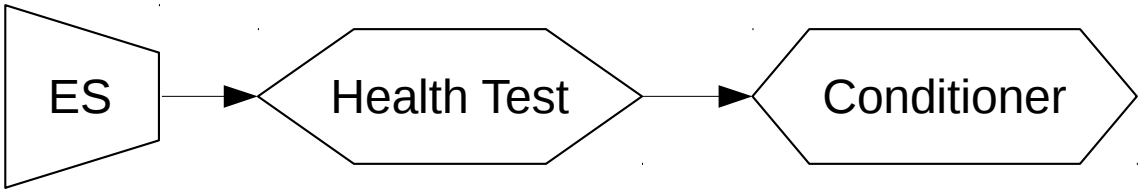
The ES is ~~a~~ dual differential jamb latch with feedback. It is a latch formed by two cross-coupled in[illegible]clock), and designed s[illegible]table state. The circuit ther[illegible]y by thermal noise in the syst[illegible] the charges on the capacitors [illegible] the random bit of output of the ES.

The circuit is also designed with feedback to seek out its metastable region. Based on how the latch resolves, a fixed amount of charge is drained from one capacitor and added

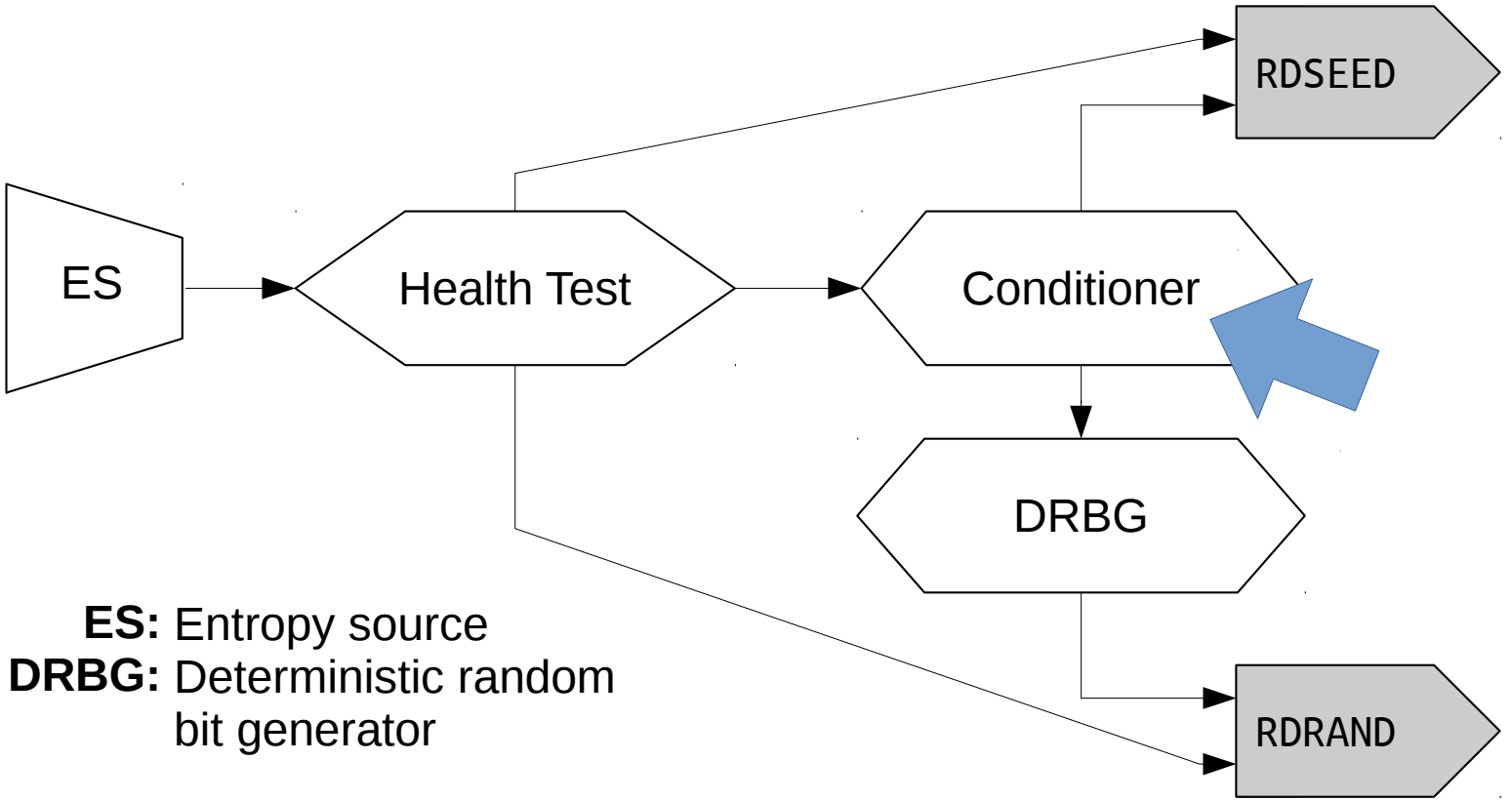


ES: Entropy source
DRBG: Deterministic random bit generator

Entropy source assumptions

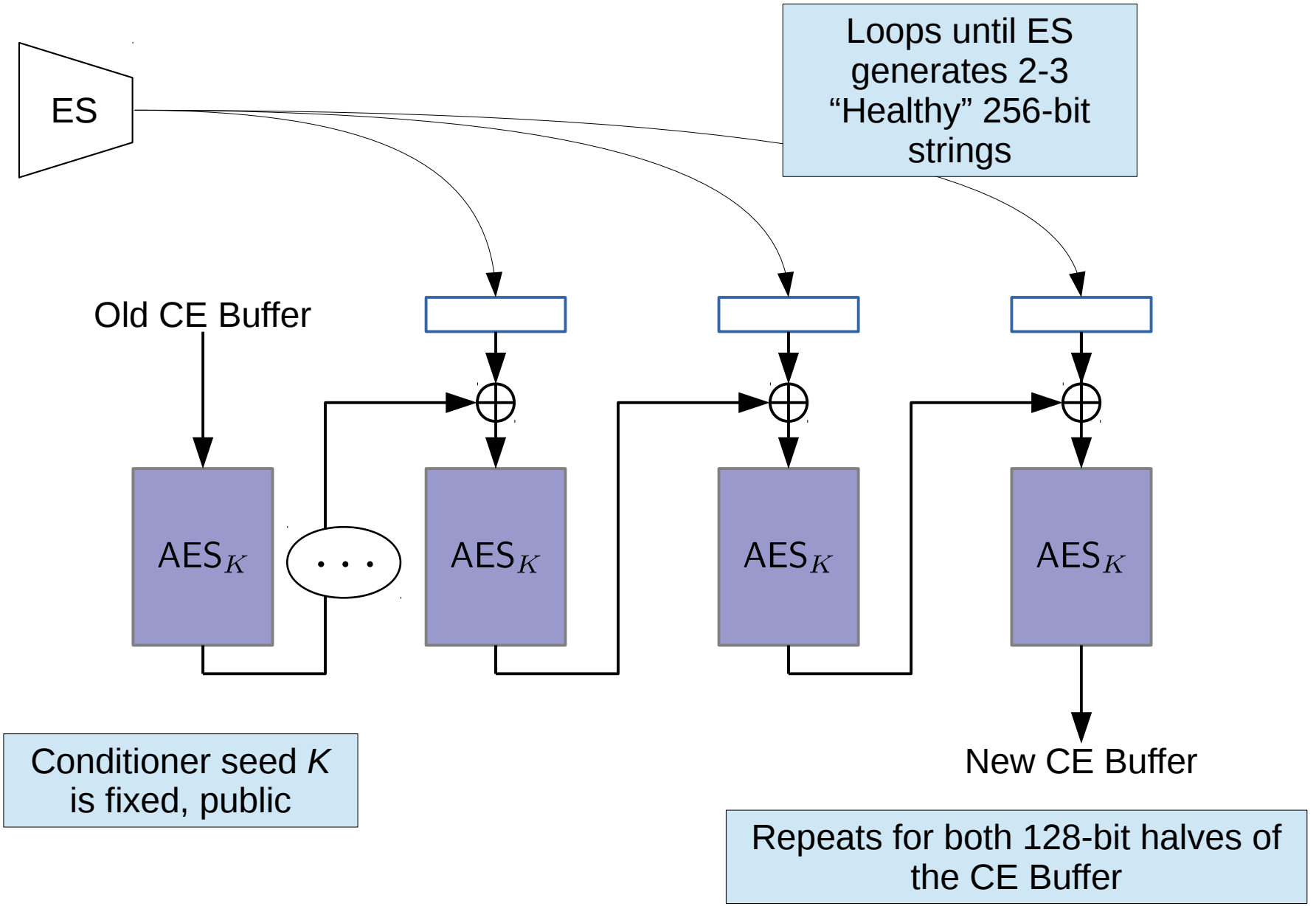


- Assume “healthy” samples have min-entropy γ
  - Will estimate value from CRI analysis
  - (Conservatively) assume no entropy from “unhealthy” samples
- Assume ES will eventually produce a healthy sample
  - Perfect entropy source = 1% of samples unhealthy
  - Say any 256-bit sample is healthy with probability $\geq \beta$



ES: Entropy source
DRBG: Deterministic random bit generator

Conditioner: CBC-MAC



Does this work?

Theorem from [DGHKR Crypto '04] says CBC-MAC (over a random permutation) works as an entropy extractor **but**

- Intel RNG recycles state
- Bound degrades quickly with input length

Does this work?

Theorem from [DGHKR Crypto '04] says CBC-MAC (over a random permutation) works as an entropy extractor **but**

- Intel RNG recycles state

Not too hard to patch

- Bound degrades quickly with input length

Does this work?

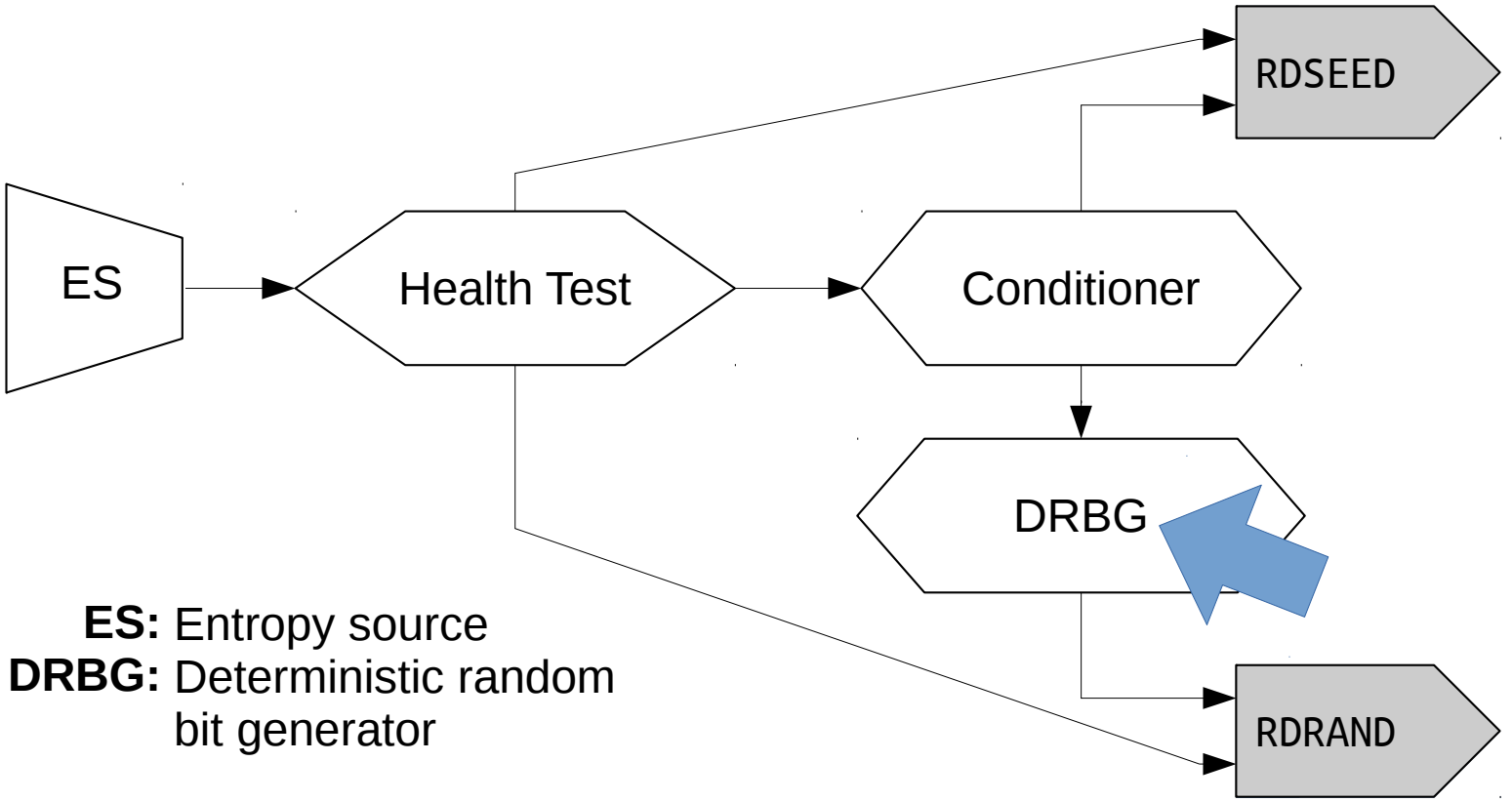
Theorem from [DGHKR Crypto '04] says CBC-MAC (over a random permutation) works as an entropy extractor **but**

- Intel RNG recycles state

Not too hard to patch

- Bound degrades quickly with input length

We can impose a fixed “cut-off” point --- don't count entropy or input length past this point.

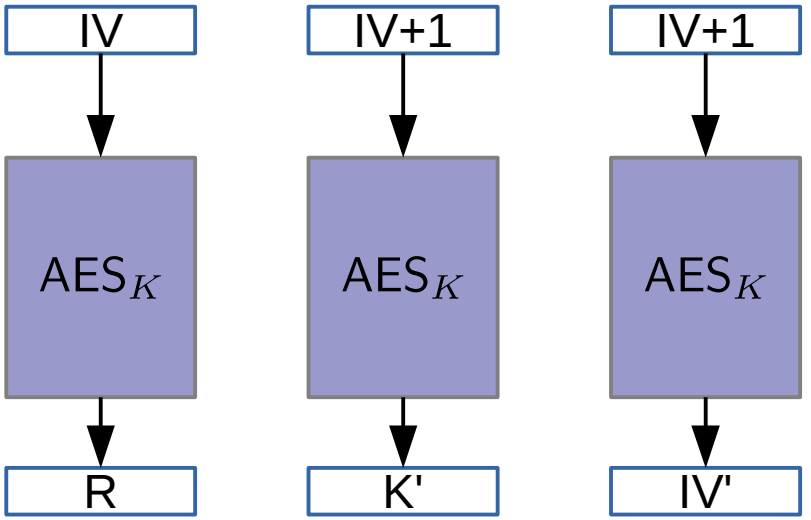


ES: Entropy source
DRBG: Deterministic random bit generator

DRBG: CTR-AES

- PWI state includes a CTR key and IV
- DRBG operation:
  - Compute R, K', IV' as below
  - Reassign (K, IV) ← (K', IV')
  - Return R as the DRBG output

Helps with forward security.



Results

$$\epsilon(L_m) = \mathcal{O}(L_m + 1)/2^{k/2}$$

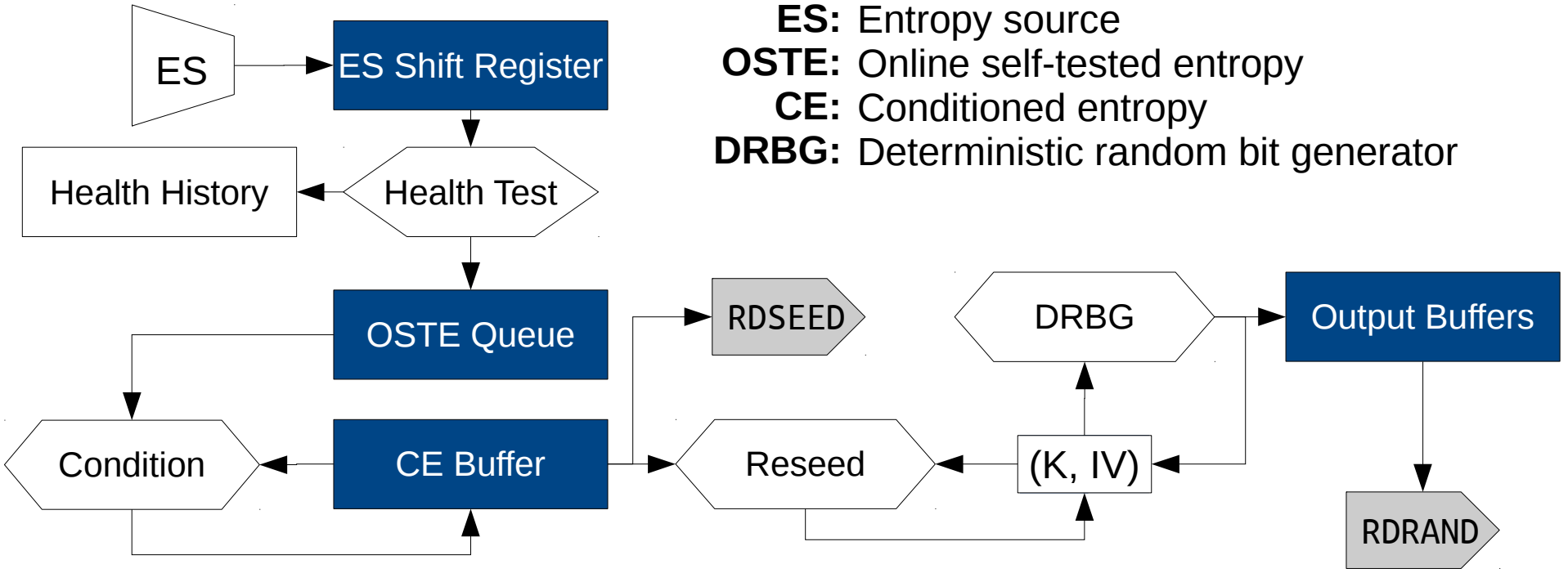
$$\hat{\epsilon}(L_m) = \sum_{i=0}^{m-1} \binom{L_m}{i} \beta^i (1-\beta)^{L_m - i}$$

$$\mathbf{Adv}^{\mathsf{INIT}}_{\mathsf{ISK\text{-}RNG},M}(A) \leq 2^{(k-m\gamma)/2+2} + 4\epsilon(L_m) + 8\hat{\epsilon}(L_m) + 5\left(\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{AES}}(B) + \frac{3}{2^k}\right)$$

$$\mathbf{Adv}^{\mathsf{FWD}/M}_{\mathsf{RDRAND}}(A) \leq 2(q+4)\left(\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{AES}}(B) + \frac{3}{2^k}\right) + \mathbf{Adv}^{\mathsf{backdoor}}_{\mathsf{RDRAND}}(\mathsf{NSA})$$

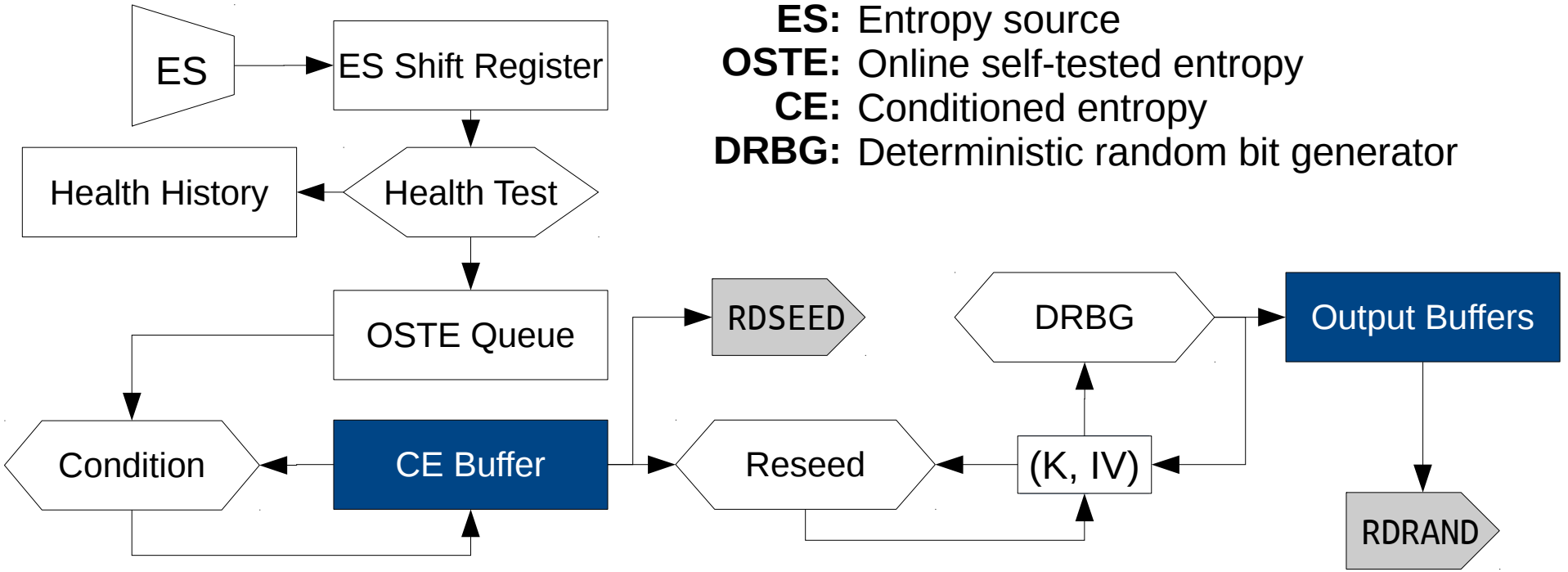
$$\mathbf{Adv}^{\mathsf{FWD}/M}_{\mathsf{ISK\text{-}RNG}}A \leq (q+1)\left(2^{(k-m\gamma)/2} + \epsilon(L_m) + 2(L_m)\right) + 2(q+4)\left(\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{AES}}(B) + \frac{3}{2^k}\right)$$

Limited Backwards Security



- Future outputs, DRBG seeds linger in buffers, no entropy collected when buffers full
- **Only a concern for hardware attacks**
- Security regained after this pipeline is flushed

Technically no Forward Security



- Old outputs linger in buffers until overwritten
  - Barring ES failure, this takes < 1 microsecond
- **Only a concern for hardware attacks**

Resilience / Limited Forward Security

Need to start plugging in numbers

- CRI report estimates each 256 bit sample has (0.65)*256 bits of entropy. Let's use 128 bits.
- No empirical data for β, but somewhere between 0.5 and 0.99 seems reasonable

Let's to find max number of `RDRAND`/`RDSEED` queries that limits advantage to 2^{-40}.

Resilience / Limited Forward Security

To keep Adversary advantage below 2^{-40}:

- **RDSEED:** Limit ~64MB (!)
- **RDRAND:** Hope Adversary can't bruteforce AES (dominant non-computational term is ~$6q/2^{128}$).

RDSEED bound is against **computationally unbounded** attackers, doesn't reflect a real weakness...

...we think.

Bottom line

- RDRAND design seems sound
- RDSEED security bounds problematic, but probably okay in practice