A Provable-Security Analysis of Intel's Secure Key RNG

Thomas Shrimpton and R. Seth Terashima

Dept. of Computer Science, Portland State University
{teshrim,seth}@cs.pdx.edu

Abstract. We provide the first provable-security analysis of the Intel Secure Key hardware RNG (ISK-RNG), versions of which have appeared in Intel processors since late 2011. To model the ISK-RNG, we generalize the PRNG-with-inputs primitive, introduced by Dodis et al. at CCS'13 for their /dev/[u]random analysis. The concrete security bounds we uncover tell a mixed story. We find that ISK-RNG lacks backward-security altogether, and that the forward-security bound for the "truly random" bits fetched by the RDSEED instruction is potentially worrisome. On the other hand, we are able to prove stronger forward-security bounds for the pseudorandom bits fetched by the RDRAND instruction. En route to these results, our main technical efforts focus on the way in which ISK-RNG employs CBCMAC as an entropy extractor.

Keywords: random number generator, entropy extraction, provable security

1 Introduction

In late 2011, Intel began production of Ivy Bridge processors, which introduced a new pseudorandom number generator (PRNG), fully implemented in hardware. Access to this PRNG is through the RDRAND instruction (pronounced "read rand"), and benchmarks demonstrate a throughput of over 500 MB/s on a quad-core Ivy Bridge processor [10]. The forthcoming Broadwell architecture will support an additional instruction, RDSEED ("read seed"), which delivers true random bits, as opposed to cryptographically pseudorandom ones. Both RDRAND and RDSEED fall under the Intel Secure Key umbrella, so we will refer to the new hardware as the ISK-RNG [11].

The ISK-RNG has received a third-party lab evaluation [8], commissioned by Intel, but has yet to receive an academic, provable-security treatment along the lines of that given the /dev/[u]random software RNGs by a line of papers [7,1,5,13]. We provide such a treatment.

Our abstract model for the ISK-RNG is that of a PRNG-with-input (PWI), established by Barak and Halevi [1] and extended by Dodis et

al. [5]. To better capture important design features of the ISK-RNG we make several improvements to the PWI abstraction, which have significant knock-on effects for the associated security notions. Our results establish the security of the ISK-RNG relative to these notions. Our findings are mixed, suggesting that in some cases `RDSEED` may not be as secure as one might hope, but with stronger results for `RDRAND`.

The ISK-RNG architecture. A detailed description of the ISK-RNG can be found in Section 3, but we'll provide a short sketch here. At a high-level, the ISK-RNG consists of four main components, as shown in Figure 1. At the heart is the hardware *entropy source*, which uses thermal noise to generate random bits and then writes them into a 256-bit raw-sample buffer. This buffer is subjected to a battery of heuristic *health tests*, which try to determine if the buffer contents are sufficiently random. The raw entropy bits are not assumed to be *uniformly* random — they may be biased or correlated. So a *conditioner* (i.e. an entropy extractor), repeatedly reads from this buffer, combining multiple 256-bit samples and compressing them into a single 128-bit string, hopefully one that is close to uniformly random.

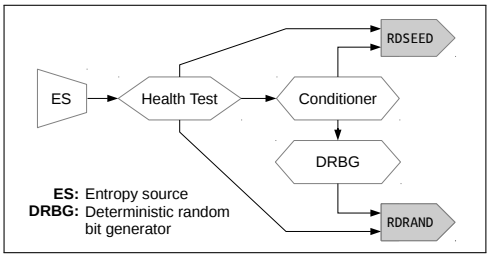

Fig. 1: Overview of the ISK-RNG.

These uniform bit strings then periodically reseed a deterministic PRNG (based on CTR-AES), providing a high-speed source of pseudo-random bits. Calls to the `RDRAND` instruction read from these bits, whereas calls to `RDSEED` will read directly from the conditioner output.

1.1 Security findings for the ISK-RNG

We consider security of the ISK-RNG relative to four PWI-security notions, adopted (with modifications) from Dodis, Pointcheval, Ruhault, Vergniaud and Wichs [5] (hereafter DPRVW): *resilience*, the apparent randomness of `RDRAND` and `RDSEED` outputs; *forward security*, the apparent randomness of previous `RDRAND` and `RDSEED` outputs once the PWI state is revealed; *backward security*, the apparent randomness of future `RDRAND` and `RDSEED` outputs from a corrupted PWI state; and *robustness*,

the apparent randomness of RDRAND and RDSEED outputs when state observation and corruption may happen at arbitrary times.

Using estimates for the quality of the entropy source derived from the findings of [8], we are able to show the following results (in a random permutation model):

1. As far as the resilience of RDRAND and RDSEED is concerned, RDRAND delivers pseudorandom bits with a comfortable security margin. On the other hand, RDSEED delivers truly random bits but with a security margin that becomes worrisome if an adversary can see a large number of outputs from either interface. If he controls an unprivileged process on the same physical machine, this could happen very quickly.
2. For forward security, RDRAND and RDSEED also provide these respective security margins, as long as one is willing to make some reasonable assumptions about the adversary's limitations.
3. The ISK-RNG does *not* provide backward security because the hardware indefinitely retains stale state when the ISK-RNG is not in active use. However, we are able to quantify the lifespan of this information when the ISK-RNG *is* in active use, thus proving backwards security and a read-only form of robustness against a class of "slow" adversaries.

Interpretation. In this context, forward security, backward security, and robustness are only relevant to those concerned about attackers who (1) are able to obtain physical access to the machine and (2) sophisticated enough to read or tamper with registers directly (the registers in question are not accessible through software, even by the operating system). Moreover, the window of opportunity for an attacker trying to compromise forward security (i.e., trying to reconstruct past random values given current access to the machine) is under a millisecond, barring pathological failures of the entropy source. Hence we suspect most practitioners will be concerned only with resilience.

As far as resilience, then, we prove RDRAND to be secure under a reasonable set of assumptions regarding the quality of the entropy source and a reasonable but heuristic assumption regarding AES-128: namely that it can be modeled as a random permutation when used with a specific fixed, publicly known key. We provide concrete, quantitative analysis in Section 7.3; the results are encouraging.

The situation with RDSEED is more complicated, because the security bounds become quantitatively quite weak in this context. We believe, but cannot prove, that this weakness does not correspond to a practical at-

tack. Our suspicion is that an actual attack would require the adversary to have a precise physical model of the entropy source (the exact parameters of which appear to change from chip to chip [8]), and compute, by brute force, the distribution induced by processing streams from this entropy source using CBC-MAC under the previously mentioned AES key. Such an attack would clearly be computationally infeasible as long as the number of possible streams is large, but the relevant portion of the security bound is for computationally unbounded adversaries. (Recall that `RDSEED` is designed to provide truly random bits, rather than "merely" cryptographically pseudorandom ones.)

The stronger `RDRAND` results hold even if an attacker can access both interfaces.

Analyzing the ISK-RNG entropy extractor. The core technical results of the paper are concerned with analyzing the ISK-RNG entropy extractor, which employs CBC-MAC over AES-128, using the fixed string $\mathsf{AES}_{\mathbf{0}}(1)$ as the AES key. Although Intel documents [17] appeal to a CRYPTO'02 paper by Dodis, Gennaro, Håstad and Krawcyzk [4] for support, this direct appeal is not well founded. There are significant technical obstacles to overcome before these CBC-MAC results can be applied. For example, because extractor-dependent state is maintained across extractions (including state revealed to the adversary by `RDSEED`), a crucial "seed independence" assumption is violated. The CRI report [8], on the other hand, ignores the issue entirely by making an implicit assumption that applying CBC-MAC-AES to an arbitrary input with 128 bits of min-entropy will produce an output close to a uniformly random 128-bit string, an assumption known to be false with respect to *any* entropy extractor (not just CBC-MAC) [15]. We discuss and resolve these issues in Section 4.

1.2 Improvements to the PWI model

For our abstract model, we take the *pseudorandom number generator with input* (PWI) primitive, formalized by DPRVW as a model for `/dev/[u]random`. At a high level, a PWI surfaces three algorithms: one to initialize the internal state of the primitive, one that produces an output for use by calling applications (updating the state in the process), and one that updates the state as a function of an *externally* provided input. Exposing an external input captures the practical situation in which PRNG outputs may depend upon external sources of (assumed) entropy.

One contribution of this paper is to generalize the PWI abstraction in ways that better capture not only the ISK-RNG, but also, we hope,

other real-world PWIs. These include allowances for: non-uniform state, as is common in real-world PRNGs; realistic modeling of state setup procedures such as those in ISK-RNG[1]; multiple external interfaces to the underlying state (e.g. `RDRAND` and `RDSEED`, as well as `/dev/[u]random`); and blocking behaviors.

To deal with non-uniform state, we introduce an analytical tool called a *masking function*. Loosely speaking, a masking function M is a tool for specifying what the "ideal" version $M(S)$ of any given PWI state S would be. This allows us to give general results about PWI security (e.g. what can be achieved when the state is ideal), yet admits per-scheme specification of what "ideal" means. We define masking functions, and incorporate them into the DPRVW's security notions in such a way that their results can be quickly lifted to our setting. Masking functions also allow us to frame an appropriate definition for secure initialization: i.e.e does the setup procedure produce a state S that is indistinguishable from $M(S)$?

2 Preliminaries

Notation. We denote the set of all n-bit strings as $\{0,1\}^n$, and the set of all (finite) binary strings as $\{0,1\}^*$. Given $x, y \in \{0,1\}^*$, both xy and $x \parallel y$ denote their concatenation, and $|x|$ is the length of x. If $|x| = |y|$, $x \oplus y$ is the bitwise XOR of x and y. The symbol ε denotes the empty string. The set $\mathrm{Perm}\,(n)$ denotes the set of permutations on $\{0,1\}^n$.

When S is a finite set, we assume that it is equipped with the uniform distribution unless otherwise specified. For any distribution $\mathcal{S}$, the notation $X \xleftarrow{\$} \mathcal{S}$ indicates X is a random variable sampled from $\mathcal{S}$. Similarly, if F is a randomized algorithm, $X \xleftarrow{\$} \mathsf{F}(x_1, \ldots, x_n)$ means that X is sampled from the distribution induced by providing F with the indicated arguments. An adversary A is a randomized algorithm, and we adopt the shorthand $A \Rightarrow y$ to mean that when its execution halts, it outputs y. When an algorithm P is provided oracle (black-box, unit-time) access to an algorithm Q, we write P^Q.

Entropy and Sources. If X and X' are random variables, their statistical distance is $\Delta(X, X') = \frac{1}{2} \sum_x |\Pr[\, X = x \,] - \Pr[\, X' = x \,]|$, where the sum is over the union of the supports of X and X'. The min-entropy of X is $\mathbf{H}_\infty(X) = -\max_x (\log \Pr[\, X = x \,])$, and the worst-case min-entropy of X given X' is $\mathbf{H}_\infty(X \mid X') = -\log\left(\max_{x,x'} \Pr[\, X = x \mid X' = x' \,]\right)$.

[1] See [9,6] for examples of what can go wrong when state initialization is weak.

When X is a random variable and $\mathcal{E}$ is some event, we denote by $X|_{\mathcal{E}}$ the random variable X conditioned on $\mathcal{E}$; i.e., for any x in the support of X, $\Pr[\, X|_{\mathcal{E}} = x \,] = \Pr[\, X = x \mid \mathcal{E} \,]$.

An *entropy source* $\mathcal{D}$ is a randomized algorithm that, given a state string $\sigma \in \{0,1\}^*$, samples a tuple $(\sigma', I, \gamma, z) \in \{0,1\}^* \times \{0,1\}^p \times \mathbb{R}_{\geq 0} \times \{0,1\}^*$. Let $(\sigma_i, I_i, \gamma_i, z_i) \xleftarrow{\$} \mathcal{D}(\sigma_{i-1})$ be a sequence of samples, where $\sigma_0 = \varepsilon$, and $i = 1, \ldots, q_{\mathcal{D}}$ for some integer $q_{\mathcal{D}}$. We say that entropy source $\mathcal{D}$ is *legitimate* if $H_\infty(I_j \mid (I_i, z_i, \gamma_i)_{i \neq j}) \geq \gamma_j$. In this paper, we assume all entropy sources are legitimate.

In this definition, $\sigma, \sigma' \in \{0,1\}^*$ represent the current and new states for $\mathcal{D}$, respectively. The string $I \in \{0,1\}^p$ is what will be to be fed as input to the PWI, and should provide fresh entropy. The quantity $\gamma \in \mathbb{R}_{\geq 0}$ is an estimate for the amount of entropy contained in I. We note that γ is strictly a convenient book-keeping device in the PWI model, and is not intended to reflect an actual output of the entropy source being modeled. Our security notions will formalize attacker capabilities of interest, but we also allow for side-information (about I) that an attacker might obtain through means not otherwise explicit in the model (e.g. timing or power side-channels). This side information will be encoded in the string z.

Cryptographic building blocks. A blockcipher is a function $E : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ such that for each key $K \in \{0,1\}^\kappa$, $E(K, \cdot)$, written $E_K(\cdot)$, is a permutation on $\{0,1\}^n$. Given $\mathsf{IV} \in \{0,1\}^n$, $K \in \{0,1\}^\kappa$, and $X_i \in \{0,1\}^n$ for $i \in [0..\nu]$, define

$$\mathsf{CTR}_K^{\mathsf{IV}}(X_0 \cdots X_\nu) = (X_0 \oplus E_K(\mathsf{IV})) \,\|\, \cdots \,\|\, (X_\nu \oplus E_K(\mathsf{IV} + \nu)).$$

(We define the + operator on $\{0,1\}^n$ as addition modulo 2^n on the unsigned integers encoded by the operands.) Further define

$$\mathsf{CBCMAC}_K^{\mathsf{IV}}(X_0 \cdots X_\nu) = \mathsf{CBCMAC}_K^{E_K(\mathsf{IV} \oplus X_0)}(X_1 \cdots X_\nu),$$

and $\mathsf{CBCMAC}_K^{\mathsf{IV}}(\varepsilon) = \mathsf{IV}$. Describing the standard CBCMAC algorithm in this manner simplifies descriptions of programs that compute CBCMAC online. We omit an explicit IV from the notation when $\mathsf{IV} = 0^n$. In this paper, the implicit blockcipher E will always be AES-128 ($\kappa = n = 128$).

The pseudorandom-permutation (PRP) advantage of an adversary A attacking a blockcipher $E : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ is defined as $\mathbf{Adv}_E^{\mathrm{prp}}(A) = \Pr\left[\, A^{E_K} \Rightarrow 1 \,\right] - \Pr[\, A^\pi \Rightarrow 1 \,]$, with probabilities over the coins of A and the random variables $K \xleftarrow{\$} \{0,1\}^\kappa$ and $\pi \xleftarrow{\$} \mathrm{Perm}(n)$.

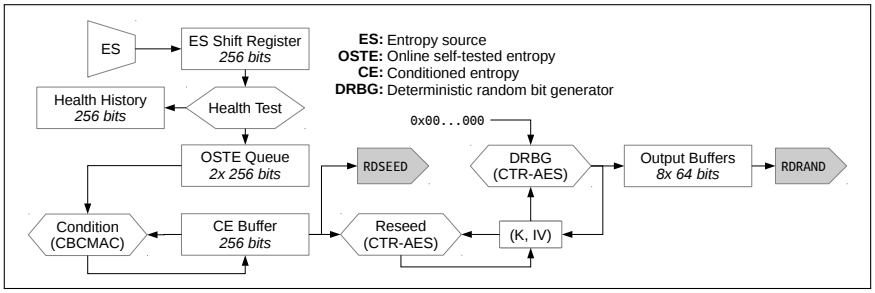

Fig. 2: Block diagram for Intel's `RDRAND` implementation. The CBCMAC computation uses AES-128 with the fixed key $K' = \mathsf{AES}_{\mathbf{0}}(1)$. The DRBG runs AES-128 in counter mode to produce $\{0,1\}^{128 \cdot 3}$ bits of output; the first 256 bits are used to update the key K and IV; the final 128 bits are sent to the output buffer, which is read by the `RDRAND` instruction.

3 The ISK-RNG architecture

This section describes the design of the ISK-RNG. Unless otherwise noted, this information comes from the CRI report [8]. The design can be divided roughly into three phases: entropy generation, entropy extraction, and expansion. Raw bits from the generation phase are fed into an entropy extractor, which is tasked with turning biased or correlated bits into uniform random strings. The expansion step uses these strings to seed a deterministic PRNG, which can produce cryptographically pseudorandom outputs at high speeds.

The design is shown in Figure 2. In this figure, rectangular boxes indicate values we consider part of the ISK-RNG state, hexagons indicate procedures that read and modify the state, and the shaded arrows indicate assembly instructions that allow (unprivileged) processes to read from the indicated buffer.

Entropy generation, Health Tests, and "Swellness". The hardware entropy source (labeled ES) is a dual differential jamb latch with feedback; thermal noise resolves a latch formed by two cross-coupled inverters, generating a random bit before the system is reset. Bits from the entropy source are written into a 256-bit shift register.

Every 256 writes, the contents of the register are subjected to a series of health tests. These count how many times certain specified bit strings appear, and verify that the results are within normal limits. For example, the substring `010` may occur between 9 and 57 times, inclusive. These substrings and the corresponding numbers of allowable occurrences are intended to catch pathologically bad failures while keeping the false-positive rate low. (For reference, a uniformly random 256-bit string would

be flagged as unhealthy approximately 1% of the time.) If the current ES register fails one of the tests, that 256-bit source-sample is flagged as *unhealthy*. We refer interested readers to the CRI report [8] for a more detailed description; for our purposes, it suffices to say there is some fixed set $\mathcal{H} \subseteq \{0,1\}^{256}$ of strings that pass the health tests. The health-history register tracks how many of the last 256 samples passed the health test. This is a first-in first-out buffer, where a 1-bit means that a sample was deemed healthy, and a 0-bit mean that a sample was deemed unhealthy. The global health of the ISK-RNG is captured by a property call *swellness*.

Definition 1 (Swell ISK-RNG). *The ISK-RNG is said to be* swell *if at least 128 of the last 256 samples were healthy, i.e. if the health-history register contains at least 128 1s.* □

Whether or not the current sample passes the health test, it is appended to the Online Self-Tested Entropy (OSTE) queue, and it is the OSTE queue that provides input to the extraction phase.

Extraction. Strings in the OSTE queue are not assumed to be uniformly random. Instead, each 256-bit entry is assumed to have a certain amount of min-entropy. The CBCMAC construction, over AES with key $K' = \mathsf{AES}_{\mathbf{0}}(1)$ [12], is employed as an entropy extractor, in order to turn strings in the OSTE queue into two 128-bit *conditioned entropy* (CE) strings. These are held in the CE buffer, which is initially all zeros, and are used to service `RDSEED` instructions and to reseed the DRBG. An important property of the CE buffer is its *availability*.

Definition 2 (CE buffer availability). *The CE buffer is* available *if (1) the ISK-RNG is swell, and (2) both 128-bit halves of the CE buffer (CE_0 and CE_1) have been updated using m healthy OSTE values since the most recent* `RDSEED` *call and the most recent DRBG reseeding. For Ivy Bridge chips, $m = 2$; for Broadwell chips, $m = 3$ [12].* □

When the CE buffer is not available, the hardware will replenish the OSTE buffers with fresh entropy and feed them into a running CBCMAC calculation until a sufficient number of healthy samples have been conditioned. So if at some point $\mathsf{CE}_0 = X$ and then the CE buffer is used to service a `RDSEED` instruction (making the CE buffer unavailable), the hardware will collect entropy strings $I_1, I_2, I_3, \ldots \in \{0,1\}^{256}$ and reassign $\mathsf{CE}_0 \leftarrow \mathsf{CBCMAC}_{\mathbf{0}}(XI_1I_2I_3\cdots)$ online until there exist $i_1 < i_2 < \cdots < i_m$

such that $I_{i_j} \in \mathcal{H}$ for $j \in [1..m]$ *and* the ISK-RNG is swell. Then the processes will repeat for CE_1.

The particulars of the way CBCMAC is used in the ISK-RNG extractor, along with the notions of swellness and availability, play a large role in Section 4.

Expansion. To reseed the DRBG, the contents of CE_0 and CE_1 are used to generate a key and IV (respectively) for counter mode encryption over AES. This reseeding process only happens when the CE buffer is available. It takes the current key and IV, (K, IV), and updates them by computing $K \,\|\, \mathsf{IV} \leftarrow \mathsf{CTR}_K^{\mathsf{IV}}(\mathsf{CE}_1 \,\|\, \mathsf{CE}_2)$. Initially, $K = \mathsf{IV} = 0^{128}$. However, using CTR with this non-random key is not a problem as long as the CE buffer is (close to) uniformly random: since the CE buffer is XORed into the CTR keystream, it can act as a one-time pad.

A pseudorandom value R is generated by computing $R \,\|\, K \,\|\, \mathsf{IV} \leftarrow \mathsf{CTR}_K^{\mathsf{IV}}(0^{3 \cdot 128})$. (Note that this process also irreversibly updates K and IV, which helps provide forward security.) The ISK-RNG writes R to an output buffer, which is read by `RDRAND`. This FIFO output buffer [10] can contain up to eight 64-bit values. ISK-RNG allows a maximum of 511 64-bit values to be generated between reseeding operations; after this, it will only return 0s and will clear the carry bit to signal an error.

Setup. When the ISK-RNG powers on, the ISK-RNG performs a series of known-answer, built-in self-tests. Then the conditioned entropy (CE) buffer is cleared and the deterministic random bit generator (DRBG) is reseeded four times [12]. Each reseeding operation requires reconditioning the CE buffer until it is available. Finally, the system populates the eight output buffers using the DRBG.

Standards compliance. Intel states [14] that ISK-RNG is compliant with NIST's SP800-90B & C draft standards. Whereas `RDRAND` can provide bit strings with "only" a 128-bit security level (since it uses AES-128 in CTR mode), `RDSEED` has no such limitation.

4 Analysis of the ISK-RNG extractor

As we will see, some of the PWI-security results for the ISK-RNG are not as strong as one might hope. Much of this is due to weak concrete bounds on its CBCMAC entropy extractor, which is tasked with turning the presumably biased and correlated bits from the entropy source into uniformly distributed strings. Let us explain.

Previous CBC-MAC results are not (directly) helpful. A paper by Dodis, Gennaro, Håstad, Krawcyzk and Rabin [4] analyzes the security of CBC-MAC as an entropy extractor, and their results are cited by Intel documents [17] to support the ISK-RNG design. Because generic PRFs-as-entropy-extractors results [3] are too weak to be useful, the analysis of [4] takes place in the random permutation model. That is, instead of considering CBC-MAC over a blockcipher with a random key, they consider CBC-MAC over a random permutation. This model is a heuristic: even, say, AES equipped with a random key would not be a random permutation. In fact, CBC-MAC within the ISK-RNG uses AES with the fixed key $K' = \mathsf{AES}_0(1)$ (on all chips). This fact may strike one as alarming. But we believe that a "nothing-up-my-sleeve" value for the extractor seed is a reasonable choice. (Generating the seed from the entropy source would be highly suspect from a theoretical perspective, because one requires that the extractor seed be "independent" of the entropy distribution.)

Anyway, our primary goal here is to identify what we *can* say about ISK, even if we're forced to use a heuristic model. Dodis et al.[4] provide the following theorem:

Theorem 1 (CBCMAC entropy extractor [4]). *Fix positive integers k and L. Let $I \in \{0,1\}^{Lk}$ be a random variable, $R \xleftarrow{\$} \{0,1\}^k$ be a uniform random string, and let $\pi \xleftarrow{\$} Perm(k)$ be a random permutation. Then* $\Delta((\pi, R), (\pi, \mathsf{CBCMAC}_\pi(I)) \leq \frac{1}{2}\sqrt{2^{k-\mathbf{H}_\infty(I)} + \frac{\mathcal{O}(L^2)}{2^k}}$.

Unfortunately, one cannot simply apply this theorem to the CBC-MAC-based extractor used in ISK-RNG, without attending to the following two significant obstacles:

(1) As we noted in Section 3, the CBCMAC-based extractor uses its own previous output as the first block of its next input. Consequently, the CBCMAC inputs are not independent of the seed. This pushes leftover-hash-lemma style results like Theorem 1 out of scope, and furthermore prevents us from employing a black-box hybrid argument to lift the results to the multiple-query setting.

(2) The $\mathcal{O}(L^2)$ term is problematic, contributing a $\mathcal{O}(L/2^{k/2})$ term to bound.[2] We note that this is significantly worse than the familiar $\mathcal{O}(L^2/2^k)$ "birthday bound" — although the two both become vacuous

[2] A set of slides published by Intel [17] claims a much stronger result based on Theorem 1. However, in addition to failing to account for point (1) above, the difference appears to stem from a mistake in translating notation. Specifically, the above theorem from [4] writes the second term under the radical as $K \cdot \epsilon(L, K)$, where $\epsilon(L, K) = \mathcal{O}(L^2/K^2)$ and in our notation $K = 2^k$. The Intel slides, however, ap-

when $L \approx 2^{k/2}$, the former violates a desired security level $\epsilon \ll 1$ much sooner (hidden constants being equal). The weak bound is exacerbated by the fact that L may grow very quickly in the ISK-RNG during periods of time when the CE buffer is not available.

Analyzing the CBC-MAC extractor. In this section we present results that allow us to overcome these hurdles, bringing Theorem 1 into scope. In particular, the main technical result of this section is the following theorem. Loosely, it says that we can still obtain a hybrid-like bound, even though a black-box hybrid argument isn't possible. Moreover, we can avoid the problem of "runaway" input strings (resulting in large L) by, in effect, only counting a fixed-length prefix of such strings.

Theorem 2. *Fix positive integers L, k, q and n with $q \leq n$. For $i \in [1..n]$, let $I_i \in \{0,1\}^*$ be random variables with lengths divisible by k, and sample $R_i \xleftarrow{\$} \{0,1\}^k$. Fix $\pi \xleftarrow{\$} Perm(k)$. Define I_i^{L} and I_i^{R} to be the unique strings such that $\left|I_i^{\mathsf{L}}\right| = \min\{|I_i|, Lk\}$ and $I_i = I_i^{\mathsf{L}} I_i^{\mathsf{R}}$. Let $C_i = \mathsf{CBCMAC}_\pi(C_{i-1} \parallel I_i)$, where C_0 is a random variable independent of π and each I_i and R_i. Then $\Delta((\pi, C_1, \ldots, C_q, I_{>q}), (\pi, R_1, \ldots, R_q, I_{>q})) \leq \frac{1}{2}\sum_{i=1}^{q} \sqrt{2^{k - \mathbf{H}_\infty(I_i^{\mathsf{L}} \mid I_{>i}, I_i^{\mathsf{R}})} + \frac{\mathcal{O}((L+1)^2)}{2^k}}$, where $I_{>m} = (I_{m+1}, \ldots, I_n)$ for integer m.*

The proof is available in the full version of this document [16]

It remains to show that, with high probability, the (potentially) truncated extractor input contains sufficient min-entropy. Note that making reasonable min-entropy assumptions regarding the entropy source is not sufficient; for example, the approximate 1% false-positive rate of the health tests on uniformly random 256-bit strings implies that there are at least 2^{249} *unhealthy* strings. Therefore the entropy source could produce *only* unhealthy samples, resulting in unbounded L, and still have high min-entropy. In order to avoid such pathological behavior, we will later (in Section 7.2) need to introduce additional assumptions regarding the rate at which the entropy source produces healthy samples. Ultimately, we will choose L such that we have a high probability of never needing more that $L/2$ samples, but such that $L/2^{k/2}$ is small, as this term will dominate our security bounds.

---

pear to have mistranscribed this term as $L \cdot \epsilon(L, K)$ (in their notation, $L = b$ and $K = 2^n$). Since $L \ll K$ for values of interest, Intel's claim significantly underestimates the concrete security bound.

5 Modeling the ISK-RNG as a PWI

Building upon DPRVW, here we define the syntax of a PWI. We give the syntax first, and then discuss what it captures, pointing out where our definition differs from DPRVW.

5.1 The PWI model

Definition 3 (PWI). *Let p, and ℓ be non-negative integers, and let* $\mathrm{IFace}, \mathrm{Seed}, \mathrm{State}$ *be non-empty sets. A* PRNG with input *(PWI) with interface set* IFace*, seed space* Seed*, and state space* State *is a tuple of deterministic algorithms* $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next}, \mathsf{tick})$*, where*

- setup *takes no input, and generates an initial PWI state* $S_0 \in \mathrm{State}$*. Although* setup *itself is deterministic, it may be provided oracle access to an entropy source* $\mathcal{D}$*, in which case its output* S_0 *will be a random variable determined by the coins of* $\mathcal{D}$*.*
- $\mathsf{refresh} : \mathrm{Seed} \times \mathrm{State} \times \{0,1\}^p \to \mathrm{State}$ *takes a seed* $\mathsf{seed} \in \mathrm{Seed}$*, the current PWI state* $S \in \mathrm{State}$*, and string* $I \in \{0,1\}^p$ *as input, and a returns new state.*
- $\mathsf{next} : \mathrm{Seed} \times \mathrm{IFace} \times \mathrm{State} \to \mathrm{State} \times (\{0,1\}^\ell \cup \{\perp\})$ *takes a seed, the current state, and an interface label* $m \in \mathrm{IFace}$*, and returns a new state, and either ℓ-bit output value or a distinguished, non-string symbol* $\perp$*.*
- $\mathsf{tick} : \mathrm{Seed} \times \mathrm{State} \to \mathrm{State}$ *takes a seed and the current state as input, and returns a new state.* □

We will typically omit explicit mention of the the seed argument to $\mathsf{refresh}$, next and tick, unless it is needed for clarity

The setup algorithm captures the initialization of the PWI, in particular its internal state. Unlike DPRVW, whose syntax requires setup to generate the PWI seed, we view the seed as something generated externally and provided to the PWI. Permitting an explicit setup procedure is necessary to correctly model ISK-RNG and, more generally, allows us to formulate an appropriate security definition for PWI initialization.

The $\mathsf{refresh}$ algorithm captures the incorporation of new entropy into the PWI state. Like DPRVW, we treat the entropy source as external. This provides a clean and general way to model the source as untrusted to provide consistent, high-entropy outputs.

Our next algorithm captures the interface exposed to (potentially adversarial) parties that request PWI outputs. By embellishing the DPRVW

syntax for next with the interface set interface, we model APIs that expose multiple functionalities that access PWI state. This is certainly the case for the ISK-RNG, via the `RDRAND` and `RDSEED` instructions, as well as `/dev/[u]random`. We also model blocking by letting next return $\perp$.

The tick algorithm is entirely new, and requires some explanation. In the security notions formalized by DPRVW, the passage of "time" is implicitly driven by adversarial queries. (This is typical for security notions, in general.) But real PRNGs like the ISK-RNG may have behaviors that update the state in ways that are not cleanly captured by an execution model that is driven by entropy-input events (refresh calls), or output-request events (next calls). The tick algorithm handles this, while allowing our upcoming security notions to retain the tradition of being driven by adversarial queries: the adversary will be allowed to "query" the tick oracle, causing one unit of time to pass and state changes to occur.

5.2 Mapping ISK-RNG into the PWI model

We now turn our attention to mapping the ISK-RNG specification into the PWI model. Figure 3 summarizes the state that our model tracks. Figure 4 provides our model for the PWI setup, refresh, next, and tick oracles. Two additional procedures, DRBG and reseed, are used internally.

Variable	Bits	Description
ESSR	256	Entropy source shift register
window	8	Counts new bits in the ESSR
OSTE_1	256	Online self-tested entropy buffers
OSTE_2	256	
CE_0	128	Conditioned entropy buffers
CE_1	128	
ptr	1	Tracks CE buffer to condition next
health	256	Tracks health of last 256 ES samples
K	128	DRBG key (For AES-CTR)
IV	128	DRBG IV (For AES-CTR)
$\mathsf{out}_{1,\ldots,8}$	512	Eight 64-bit output buffers
outcount	≥ 4	Counts number of full output buffers
count	≥ 9	Counts DRBG calls since reseeding
CEfull	1	Set if CE buffers are available
block	1	Set if reseed has priority over `RDSEED`

Fig. 3: State variables of the ISK-RNG

Oracle setup(ES):

```
01  for i = 1,2,3,4 do
02    S.CE_0 ← CBCMAC_{K'}(S.CE_0)
03    while S.ptr = 0 do
04      I ←$ ES
05      S ← refresh(S, I)
06    S.CE_1 ← CBCMAC_{K'}(S.CE_1)
07    while S.ptr = 1 do
08      I ←$ ES
09      S ← refresh(S, I)
10    S ← reseed(S)
11  for i = 1,3,5,7 do
12    (S, R) ← DRBG(S)
13    S.out_i ‖ S.out_{i+1} ← R
14  S.outcount ← 8
15  return S
```

Oracle DRBG(S):

```
16  S.IV ← S.IV + 1
17  R ← CTR_K^V(0^128)
18  if S.CEfull then
19    S ← reseed(S)
20  else if S.count < 512
21    S.K ‖ S.V ← CTR_{S.K}^{S.V+1}(0^256)
22    S.count ← S.count + 1
23  else
24    return (S, ⊥)
25  return (S, R)
```

Oracle tick(S):

```
26  if S.CEfull and S.count > 0 then
27    S ← reseed(S)
28    return S
29  if S.count < 512 then
30    if S.outcount < 8 then
31      S.outcount ← S.outcount + 1
32      (S, R) = DRBG(S)
33      S.out_outcount ← R
34      return S
35  return S
```

Oracle refresh(S, I):

```
36  shift(S.ESSR, I)
37  S.window ← S.window + 1 mod 256
38  if S.window = 0 then
39    shift(S.health, isHealthy(S.ESSR))
40    S.OSTE_2 ← S.OSTE_1
41    S.OSTE_1 ← S.ESSR
42    i ← S.ptr
43    I_j^i ← I_j^i ‖ S.OSTE_2 // Record-keeping
44    S.CE_i ← CBCMAC_{K'}^{S.CE_i}(OSTE_2)
45    if sum(S.health) ≥ 128 then
46      if isHealthy(OSTE_2) then
47        S.samples ← S.samples + 1
48    if S.samples = m then
49      S.samples ← 0
50      if S.ptr = 0 then
51        S.ptr ← 1
52      else
53        S.ptr ← 0; S.CEfull ← 1
54        C_j^0 ‖ C_j^1 ← S.CE // Record-keeping
55        j ← j + 1; // Record-keeping
56  return S
```

Oracle reseed(S):

```
57  S.K ‖ S.V ← CTR_K^{V+1}(S.CE)
58  S.CE_0 ← CBCMAC_{K'}(S.CE_0)
59  S.CE_1 ← CBCMAC_{K'}(S.CE_1)
60  S.count ← 0; S.CEfull ← 0
61  S.ptr ← 0; S.block ← 0
62  return S
```

Oracle next(interface, S):

```
63  if interface = RDRAND then
64    if S.outcount = 0 then return (S, ⊥)
65    R ← LSB_64(S.out_1)
66    for i = 1,...,7 do
67      S.out_i ← S.out_{i+1}
68    S.outcount ← S.outcount − 1
69    return (S, R)
70  else if interface = RDSEED
71    if S.CEfull = 0 then
72      return (S, ⊥)
73    if S.block = 1, S.count > 0 then
74      return (S, ⊥)
75    R ← S.CE_0 ‖ S.CE_1
76    S.CEfull ← 0; S.ptr ← 0
77    S.CE_0 ← CBCMAC_{K'}(S.CE_0)
78    S.CE_1 ← CBCMAC_{K'}(S.CE_1)
79    S.block ← 1
80    return (S, R)
```

Fig. 4: The above oracles describe the behavior of ISK-RNG from within the PWI model. See Table 3 for a description of the state variables $S.*$. All bits are initially zero. For Ivy Bridge chips, $m = 2$, and for Broadwell chips $m = 3$. The key $K' = \mathsf{AES}_0(1)$ is fixed across all chips. The function $\mathsf{shift}(x, y)$ sets value of x to the right-most $|x|$ bits of $x \,\|\, y$. Lines marked with a "Record-keeping" comment are there to aid in proofs and exposition.

6 PWI Security

Having defined the syntax for PWIs, we can now introduce corresponding security notions. The basic notions are those of DPRVW, with a few notable alterations. To handle issues of non-uniform state and (more) realistic initialization procedures, we introduce a new technical tool, masking functions, that allows us to cleanly address these issues.

6.1 Basic notions

Here we define four PWI-security notions, in the game-playing framework [2]. In each there is a (potentially adversarial) entropy source $\mathcal{D}$, and an adversary A. The latter is provided access to the oracles detailed in Figure 5 (top), and what distinguishes the four notions are restrictions applied to the queries of the adversary A. In particular, we consider the following games:

Robustness (ROB): no restrictions on queries.

Forward security (FWD): no queries to set-state are allowed; and a single query to get-state is allowed, and this must be the final query.

Backward security (BWD): no queries to get-state are allowed; a single query to set-state is allowed, and this must be the first query.

Resilience (RES): no queries to get-state or set-state are allowed.

See DPRVW for additional discussion. We note that all games share common initialize and finalize procedures, shown in Figure 5 (bottom). Thus, the robustness-advantage of A in attacking $\mathcal{G}$ is defined to be $\mathbf{Adv}^{\mathrm{rob}}_{\mathcal{G},\mathcal{D}}(A) = 2\Pr[\,\mathsf{ROB}_{\mathcal{G},\mathcal{D}}(A) = 1\,] - 1$. The forward security, backward security, and resilience advantages $\mathbf{Adv}^{\mathrm{fwd}}_{\mathcal{G},\mathcal{D}}(A)$, $\mathbf{Adv}^{\mathrm{bwd}}_{\mathcal{G},\mathcal{D}}(A)$, and $\mathbf{Adv}^{\mathrm{res}}_{\mathcal{G},\mathcal{D}}(A)$ are similarly defined. It is clear that robustness implies forwards and backwards security, and both of these independently imply resilience.

We note that, because the PRNG cannot reasonably be expected to produce random-looking outputs without sufficient entropy or with a known or corrupted state, the various security experiments track (1) a boolean variable corrupt and (2) a value γ measuring the total entropy that has been fed into the PRNG since corrupt was last set. These serve as book-keeping devices to prevent trivial wins. The corrupt flag is cleared whenever γ exceeds some specified threshold γ^*.

Oracle $\mathcal{D}$-refresh:

$(\sigma, I, \gamma, z) \xleftarrow{\$} \mathcal{D}(\sigma)$
$S \leftarrow \mathsf{refresh}(S, I)$
$c \leftarrow c + \gamma$
if $c \geq \gamma^*$ **then**
  $\mathsf{corrupt} \leftarrow \mathsf{false}$
return (γ, z)

Oracle next-ror(m):

if corrupt **then**
  return $\perp$
$(S, R_0) \leftarrow \mathsf{next}(m, S)$
if $R_0 = \perp$ **then**
  $R_1 \leftarrow \perp$
else
  $R_1 \xleftarrow{\$} \{0,1\}^\ell$
return R_b

Oracle get-next(m):

$(S, R) \leftarrow \mathsf{next}(m, S)$
if corrupt **then**
  $c \leftarrow 0$
return R

Oracle wait:

$S \leftarrow \mathsf{tick}(S)$
return ε

Oracle get-state:

$c \leftarrow 0$
$\mathsf{corrupt} \leftarrow \mathsf{true}$
return S

Oracle set-state(S^*):

$c \leftarrow 0$
$\mathsf{corrupt} \leftarrow \mathsf{true}$
$S \leftarrow S^*$

Procedure initialize:

$\sigma \leftarrow 0$; $\mathsf{seed} \xleftarrow{\$} \text{Seed}$; $i \leftarrow 0$
$S \leftarrow \mathsf{setup}^{\mathsf{ES}}$
$c \leftarrow n$; $\mathsf{corrupt} \leftarrow \mathsf{false}$
$b \xleftarrow{\$} \{0,1\}$
return $(\mathsf{seed}, (\gamma_j, z_j)_{j=1}^{i})$

Oracle ES:

$i \leftarrow i + 1$
$(\sigma, I, \gamma_i, z_i) \xleftarrow{\$} \mathcal{D}(\sigma)$
return I

Procedure finalize(b):

if $b = b^*$ **then**
  return 1
else
  return 0

Fig. 5: Top: Oracles for the PWI security games. **Bottom:** the shared initialize and finalize procedures for the PWI security games. Recall that the output of initialize is provided to adversary A as input, and the output of finalize is the output of the game.

6.2 Masking functions and updated security notions

As noted earlier, the DPRVW security definitions assume the PWI state is initially uniformly random. However, this does not realistically model the behavior of real-world PWIs, notably ISK-RNG, which do not attempt to reach a pseudorandom state; for example, they may maintain counters. (Indeed one can construct PWIs that would be deemed secure when starting from a uniformly random state, but that would not be secure in actuality; the reverse is also true. See the full version of this paper [16] for examples.) Yet, clearly, some portion of the PWI state must be unpredictable to an attacker, as otherwise one cannot expect PWI outputs to look random.

To better capture real-world characteristics of PWI state, we introduce the idea of a *masking function*. A masking function M over state space State is a randomized algorithm from State to itself. As an example, if states consist of a counter c, a fixed identifier id, and a buffer B of (supposedly) entropic bits, then $M(c, \mathrm{id}, B)$ might be defined to return (c, id, B') where B' is sampled by M from some distribution.

A masked state is meant to capture whatever characterizes a "good" state of a PWI, i.e. after it has accumulated a sufficient amount of externally provided entropy. Informally, for any state S, we want that (1)

a PWI with state $M(S)$ should produce pseudorandom outputs, and (2) after the PWI has gathered sufficient entropy, its state S should be indistinguishable from $M(S)$.

To the second point, the initial PWI state S_0 is of particular importance. In the following definition, we characterize masking functions M such that the initial S_0 and $M(S_0)$ are indistinguishable.

Definition 4 (Honest-initialization masking functions). Let $\mathcal{D}$ be an entropy source, $\mathcal{G} = (\mathsf{setup}, \mathsf{refresh}, \mathsf{next})$ be a PWI with state space State, A be an adversary, and $M : \text{State} \to \text{State}$ be a masking function. Let (seed, Z) be the random variable returned by running the $\mathsf{initialize}()$ (Figure 5) using $\mathcal{G}$ and $\mathcal{D}$, and let S_0 be the state produced by this procedure. Set $\mathbf{Adv}^{\text{init}}_{\mathcal{G},\mathcal{D},M}(A) = \Pr[\, A(S_0, \mathsf{seed}, Z) \Rightarrow 1\,] - \Pr[\, A(M(S_0), \mathsf{seed}, Z) \Rightarrow 1\,]$. If $\mathbf{Adv}^{\text{init}}_{\mathcal{G},\mathcal{D},M}(A) \leq \epsilon$ for any adversary A running in time t, then M is a $(\mathcal{G}, \mathcal{D}, t, \epsilon)$-*honest-initialization* masking function. □

Note that the above definition is made with respect to a specific $\mathcal{D}$. The assumptions required of $\mathcal{D}$ (e.g., that it will provide a certain amount of entropy within a specified number of queries) will depend on the PWI in question, but should be as weak as possible.

We now define "bootstrapped" versions of the PWI security goals, which always begin from a masked state. This will allow us to reason about security when the PWI starts from an "ideal" state, i.e. what we expect after an secure initialization of the system.

Definition 5 (Bootstrapped security). *Let $\mathcal{G}$ be a PWI and M be a masking function. For $x \in \{\text{fwd}, \text{bwd}, \text{res}, \text{rob}\}$, let $\mathbf{Adv}^{x/M}_{\mathcal{G},\mathcal{D}}(A)$ be defined as $\mathbf{Adv}^{x}_{\mathcal{G},\mathcal{D}}(A)$, except with line 02 of the* $\mathsf{initialize}$ *procedure (Fig. 5) changed, to execute instead $S' \xleftarrow{\$} \mathsf{setup}^{\mathsf{ES}}; S \xleftarrow{\$} M(S')$.* □

6.3 PWI-Security Theorems

Bootstrapped security notions are useful, because they allow the analysis to begin with an idealized state. However, this comes at a cost: we need to ensure that the masking function is honest in the sense that it accurately reflects the result of running the setup procedure. The following theorem states the intuitive result that if the masking function is secure (and honest), then security when the PWI begins in a masked state $M(S)$ implies security when the PWI begins in state S. We omit the simple proof, which follows from a standard reduction argument.

Theorem 3. *Let $\mathcal{G}$ be a PWI, $\mathcal{D}$ be an entropy source, and M be a masking function. Suppose M is a $(\mathcal{G}, \mathcal{D}, t, \epsilon)$-honest initialization mask. Then for any $x \in \{\text{fwd}, \text{bwd}, \text{res}, \text{rob}\}$ there exists some adversary $B(\cdot)$ such that for any adversary A, $\mathbf{Adv}^{x}_{\mathcal{G},\mathcal{D}}(A) \leq \mathbf{Adv}^{x/M}_{\mathcal{G},\mathcal{D}}(B(A)) + \epsilon$. Further, if it takes time t' to compute M, and A makes q queries and runs in time t, then $B(A)$ makes q queries and runs in time $\mathcal{O}(t) + t'$.*

For a second general result, we revisit a nice theorem by DPRVW and adapt it to our model. The theorem states that if a PWI possesses two weaker security properties — roughly, the ability to randomize a corrupted state after harvesting sufficient entropy and the ability to keep its state pseudorandom in the presence of adversarial entropy — then it is robust. These definitions, however, again assume that a state "should" appear uniformly random. We present modified definitions that instead use masking functions, and prove an analogous theorem. While the transition involves a couple subtleties — in particular, we require an idempotence property of the masking function — the proof is essential identical to the one in [5]; therefore we make an informal statement here and defer the formal treatment to the full version [16].

Theorem 4 (Informal). *Let $\mathcal{G}$ be a PWI. Suppose there exists a mask M such that: (1) When starting from an arbitrary initial state S of the adversary's choosing, the final PWI state S' is indistinguishable from $M(S')$ provided the PWI obtains sufficient entropy; (2) When starting from an initial state $M(S)$ (for adversarially chosen S), the final PWI state S' is indistinguishable from $M(S')$, even if the adversary controls the intervening entropy input strings; (3) $\mathcal{G}$ produces pseudorandom outputs when in a masked state. Then $\mathcal{G}$ is robust.*

7 Security of the ISK-RNG as a PWI

We are now positioned to analyze the security of ISK-RNG. To begin, we demonstrate some simple attacks that violate both forwards and backwards security (hence robustness, too). Next, we show that by placing a few additional restrictions on adversaries — restrictions that are well-motivated by the hardware — we can recover forward security. As we said in our introduction, the concrete security bounds we prove are not as strong as one might hope, due to some limitations of CBCMAC's effectiveness as an entropy extractor in the ISK-RNG. However, we are able to prove somewhat better results when legitimate parties use only the `RDRAND` interface, even when attackers also have access to `RDSEED`. This

means that, e.g., a hostile process can't use its access to RDSEED to learn information about RDRAND return values used by a would-be victim; the result also implies stronger results for Ivy Bridge chips, where RDSEED is not available.

For the remainder of Section 7, we fix the following constants: $p = 1$ is the length of each entropy input; $k = 128$ is the length of each CBCMAC input block (since ISK-RNG uses AES); IFace = $\{\texttt{RDSEED}, \texttt{RDRAND}\}$ are the ISK-RNG interfaces; $m = 2, 3$ is the number of healthy samples required by Ivy Bridge and Broadwell, respectively, before the CE buffer is available; and $\ell = 64$ is the length of the PWI outputs. Although RDRAND also allows programs to request 16 or 32 bits, this is implemented by fetching then truncating a 64-bit output, and similarly with RDSEED [12]. Therefore we assume without loss of generality that the adversary only requests the full 64 bits.

Recall that in the PWI model, the entropy source leaks information γ about each input string. We assume that every 256th such string (each one a single bit, $p = 1$) leaks the health of the corresponding 256 bit string (as determined by the online health test). Hence the adversary will always know the health of the OSTE buffers and the value of the health buffer. This is not simply a convenience: because the CE buffer is not available until it has been reconditioned with m healthy samples, RDSEED may leak health information through a timing side channel.

When the CE buffer is available, it can be used to reseed the DRBG or to service a RDSEED instruction. Priority is given to whichever was not last used [12]. However, because the PWI model cannot describe pending RDSEED instructions, the adversary must explicitly use its wait oracle to yield when it has priority: a wait invocation uses the CE to reseed, while a RDSEED invocation returns its contents.

The adversary's wait oracle also allows us to account for the fact that updating the eight 64-bit output buffers is not an atomic operation. By using the tick function (invoked by wait) to only fill one at a time, we conservatively allow the adversary to control if a reseeding operation intervenes. Note that tick will reseed rather than fill an output buffer if reseeding is desired ($S.\mathsf{count} > 0$) and possible ($S.\mathsf{CEfull} = 1$). This reflects the priorities of the hardware [12].

In order to save power, the entropy source goes to sleep if all the output buffers are full, the CE buffer is available, and no RDRAND instructions have been processed since the last reseed [12]. The PWI model, however, requires that we continue to provide $\mathcal{D}$-refresh access to the adversary. Our decision to leak health information to the adversary allows us to

avoid any problems here: the adversary knows when the entropy source sleeps, so we can restrict the adversary to not make $\mathcal{D}$-refresh calls when it does.

To make this power-saving hardware constraint "work" with the PWI model, we assume that each healthy 256-bit block produced by the entropy source contains at least γ bits of min-entropy. Formally, define $(\sigma_i, b_i, \gamma_i, z_i) = \mathcal{D}(\sigma_{i-1})$ for $i \geq 1$ (where $\sigma_0 = \varepsilon$), and let $I_i = b_{256i}b_{256i+1} \cdots b_{256i+255}$. We assume $\mathbf{H}_\infty\left(I_i \mid (\sigma_j, I_j, \gamma_j, z_j)_{j\neq i}, I_i \in \mathcal{H}\right) \geq \gamma$, for some $\gamma > 0$, and require that $\sum_{j=256i}^{256i+255} \gamma_i \geq \gamma$ whenever $I_i \in \mathcal{H}$. We set $\gamma^* = m\gamma$ to demand, in effect, that ISK-RNG delivers on its implicit promise that m healthy entropy samples are sufficient. At the end of this section, we will draw from the CRI report's analysis to find reasonable estimates for γ and discuss the implications.

7.1 Negative Results

We begin with some quick negative results, showing that the ISK-RNG achieves neither forward nor backwards security. This immediately rules out robustness, too. We again emphasize that these negative results will be followed by positive results for realistic classes of restricted adversaries; we present them primarily to motivate the coming restrictions.

Theorem 5 (ISK-RNG lacks forward security). *There exists an adversary A making one* next-ror *query and one* get-state *query such that for any entropy source $\mathcal{D}$, $\mathbf{Adv}^{\mathrm{fwd}}_{\mathsf{ISK},\mathcal{D}}(A) = 1 - 2^{-128}$.*

Theorem 6 (ISK-RNG lacks backward security). *There exists an adversary A making one* next-ror *query and one* set-state *query such that for any entropy source $\mathcal{D}$, $\mathbf{Adv}^{\mathrm{bwd}}_{\mathsf{ISK},\mathcal{D}}(A) = 1 - 2^{-128}$.*

In the case of backwards security, the adversary sets some initial state S with $S.\mathsf{samples} = 0$, makes a sequence of $\mathcal{D}$-refresh calls to clear the corrupt flag (which, by our previously state assumptions, will happen as soon as the CE buffer becomes available), and finally assigns $X \leftarrow$ next-ror(RDRAND). The adversary then checks if $X = S.\mathsf{out}_1$, and outputs 0 if this is the case and 1 otherwise. For forward security, the adversary assigns $X \leftarrow$ next-ror(RDSEED), then learns the resulting state S using get-state(). If $X = \mathsf{AES}_{\mathbf{0}}^{-1}(S.\mathsf{CE}_0) \,\|\, \mathsf{AES}_{\mathbf{0}}^{-1}(S.\mathsf{CE}_1)$, the adversary outputs 0; otherwise, the it outputs 1. (Here, $\mathbf{0} = 0^{128}$.)

However, these results are very conservative. In the case of forward security, the hardware will quickly recondition the CE buffer and refill the

output buffers, effectively erasing all state that could be used to compute previous outputs. Backwards security is more complicated because not only do future outputs persist in the output buffer indefinitely, but future DRBG keys are leaked via the ESSR, OSTE, and CE buffers. Once the output buffers are flushed, though, these other buffers will quickly be overwritten with fresh entropy.

7.2 Positive results

We now turn our attention to restricted, but still conservative, classes of adversary in order to produce positive results.

Additional assumptions. We further assume that in the forward-security game, adversaries do not make their get-state query until they have allowed the output buffers to be refilled. This assumption is motivated by the speed with which the hardware will automatically accomplish this: at the reported `RDRAND` throughput of 500 MB/s, all eight 64-bit buffers can be refilled around 8 million times per second. Formally:

Definition 6 (Delayed adversaries). *An adversary A attacking ISK-RNG in the forward-security game is* delayed *if after making its last* get-next *and* next-ror *queries, A calls* $\mathcal{D}$-refresh *until the CE buffer is available, then calls* wait *nine times before making its* get-state *query.* □

This will trigger a reseed and then refill any empty output buffers.

Moreover, we will assume there is some positive probability β such that each 256-bit block of bits from the entropy source is healthy with probability at least β. Formally (recall that $\mathcal{H} \subseteq \{0,1\}^{256}$ is the set of strings deemed healthy by ISK-RNG's online health tests):

Definition 7 (β-healthy). *Let $\mathcal{D}$ be an entropy source and fix $\beta > 0$. Let $\mathcal{H} \subseteq \{0,1\}^{256}$ be the set of strings deemed healthy by the ISK-RNG. For $i = 1, 2, 3, \ldots$ define $(\sigma_i, b_i, \gamma_i, z_i) = \mathcal{D}(\sigma_{i-1})$ (where $\sigma_0 = \varepsilon$), and for $j = 0, 1, 2, \ldots$, define $B_j = b_{256j} \,\|\, b_{256j+1} \,\|\, \cdots \,\|\, b_{256j+255}$. Let $H_j = 1$ if $B_j \in \mathcal{H}$, and set $H_j = 0$, otherwise. Then $\mathcal{D}$ is* β-healthy *if for all such j and all $H \in \{0,1\}^{j-1}$, $\Pr[\, B_j \in \mathcal{H} \mid (H_\ell)_{\ell<j} = H \,] \geq \beta$.* □

So for any positive integers ℓ and L_m, we can upper bound the probability that the sequence $(B_i)_{i=\ell}^{\ell+L_m-1}$ contains fewer than m healthy values using: $\Pr[\, |\{j \,:\, B_j \in \mathcal{H}, \ell \leq j < \ell + L_m\}| < m \,] \leq \sum_{i=0}^{m-1} \binom{L_m}{i} \beta^i (1-\beta)^{L_m - i}$.

Remark 1. Our goal is to identify under what reasonable assumptions ISK-RNG could be deemed secure, and, as we argued at the end of Section 4, this requires making an assumption about the entropy source's

ability to produce "healthy" samples (a min-entropy assumption is too weak). We settled on the above β-healthy assumption because it is simple and fairly broad: we do not assume the probabilities of samples being healthy are constant or even independent, just that the conditional probabilities don't dip below the β threshold. Moreover, we later show that the "unhealthy sample rate" could easily be fifty times the ideal 1% false-positive base rate without significantly damaging our bounds. Finally, even the β-healthy assumption is more than we need. We require an upperbound on the probability on the left-hand side of the above equation, and the β-healthy assumption provides a natural, concrete way to think about this probability.

Rigorously testing the β-healthy assumptiton without access to the entropy source is problematic. That being said, barring such access, we doubt it would be possible to do significantly better.

With these assumptions, we are ready to continue on to our positive results. Our first step is to define an appropriate masking function that describes an "ideal" state, and then to prove that setup creates such a state. This lets later proofs simply assume we begin in an idealized state (see Theorem 3).

ISK-RNG masking function. Fix the masking function $M : \{0,1\}^n \to \{0,1\}^n$ that on input S, overwrites $S.\mathsf{CE}$, $S.K$, $S.\mathsf{IV}$, and $S.\mathsf{out}_{1,\ldots,8}$ with independent, uniformly random strings of the appropriate lengths, leaves all other portions of the state untouched, and returns the result (refer back to Fig. 3 for a listing of the components of the ISK-RNG state S). This is the ISK-RNG masking function.

Recall the results of Theorem 2. For convenience, we define $\epsilon(L_m) = \mathcal{O}(L_m+1)/2^{k/2}$ and $\hat{\epsilon}(L_m) = \sum_{i=0}^{m-1} \binom{L_m}{i} \beta^i (1-\beta)^{L_m-i}$, where $\epsilon(L_m)$ is from Theorem 2 and $\hat{\epsilon}(L_m)$ is the above bound on the probability of obtaining fewer than m healthy samples from a β-healthy entropy source within L_m trials. Our theorem statements refer to various previously defined values, summarized in Figure 6

The following lemma says that if AES is a secure PRP (against adversaries making three queries) and each healthy sample from the entropy source has sufficiently large min-entropy, then the ISK-RNG masking function is honest. That is, that the ISK-RNG setup procedure successfully places the hardware in a state where (we will show) it can begin producing pseudorandom outputs.

Lemma 1 (ISK-RNG masking function is honest). *Fix positive integers k and m, and fix $0 < \beta \leq 1$. Let L_m be a positive integer.*

k	CBCMAC blocksize (128 bits)
m	Number of "healthy" $2k$-bit strings that need to be conditioned before the CE buffer becomes available ($m = 2, 3$ for Ivy Bridge and Broadwell chips, respectively).
L_m	Parameter we can freely choose to keep both $\hat{\epsilon}(L_m)$ and $\epsilon(L_m)$ small.
γ	An assumed lower bound on the conditional min-entropy of healthy strings.

Fig. 6: Summary of values used for theorem statements.

Let M be the ISK-RNG masking function. Let $\mathcal{D}$ be a β-healthy entropy source. Then for any adversary A, there exists an adversary B running in the same time and making three queries such that $\mathbf{Adv}^{\text{init}}_{\mathsf{ISK},\mathcal{D},M}(A) \leq 2^{(k-m\gamma)/2+2} + 4\epsilon(L_m) + 8\hat{\epsilon}(L_m) + 5\left(\mathbf{Adv}^{\text{prp}}_{\mathsf{AES}}(B) + \frac{3}{2^k}\right)$.

The proof is deferred to the full version of this paper [16]. Using reasonable estimates for the big-$\mathcal{O}$ constant and γ (discussed in Section 7.3) provides us with an upper bound of roughly 2^{-60} for the first three terms of the security bound for both $m = 2, 3$.

Remark 2. The PRP term may be problematic if one takes the view that `RDSEED` should offer information-theoretic security. That is, Lemma 2 says that the ISK-RNG initialization procedure yields state — which includes the CE buffers — that is only computationally indistinguishable from "ideal". However, we observe that if one adjusts the masking function to leave the output buffers unchanged, and demands a post-setup reconditioning (which the hardware endeavors to provide, anyway), one could indeed use the result to prove information-theoretic `RDSEED` security. However, this would be at the expense of *not* being able to prove security of the `RDRAND` interface, a task which necessarily requires computational assumptions.

Forward security. Our exploration of forward security proceeds in two steps. To begin, we introduce a new game, M-`RDRAND`, which differs from M-FWD in that the next-ror oracle always returns the "real" value R_0 when queried on the `RDSEED`interface, but behaves normally during queries to the `RDRAND` interface. Define

$$\mathbf{Adv}^{\text{fwd}-\texttt{RDRAND}/M}_{\mathcal{G},\mathcal{D}}(A) = 2\Pr[\, M\text{-}\texttt{RDRAND}(A) \Rightarrow 1\,] - 1.$$

Proving the security of this game is not only a useful intermediate step in proving the security of M-FWD, but also can be interpreted as measuring the strength of `RDRAND` return values when an adversary also has access to the `RDSEED` instruction (which can be used to learn information about

the ISK-RNG state, but that we do not require to return pseudorandom values). This distinction is valuable, because the concrete bounds on the M-FWD experiment are not as strong as one would hope.

Theorem 7 (M-RDRAND). *Let A be a delayed adversary making q queries to* RDRAND *and running in time t. Then there exists an adversary B making three queries and running in time $\mathcal{O}(t)$ such that* $\mathbf{Adv}^{\mathrm{fwd-RDRAND}/M}_{\mathsf{ISK},\mathcal{D}}(A) \leq 2(q+4)\left(\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{AES}}(B) + \frac{3}{2^k}\right)$.

The proof appears in the full version [16]. Barring an efficient attack on AES (that only uses three queries!) this bound is quite strong. If q were to grow quite large, say on the order of $q \approx 2^{80}$, then the bound might begin to approach 2^{-40}, which seems a reasonable safety margin. However, even at the reported rate of around 500 MB/s, ISK-RNG would take over 70 years to reach this point. Moreover, the hybrid factor of q is likely a conservative artifact of the proof.

Note, however, that this bound applies to ISK-RNG when starting in an "ideal" masked state; one needs to add in the bound from Lemma 1 to account for initialization. As we mentioned earlier, reasonable estimates for the big-$\mathcal{O}$ constant and γ (see Section 7.3) place this term at roughly 2^{-60}.

We now proceed to the "full" forward-security result, where both the RDRAND and the RDSEED interfaces are required to produce indistinguishable-from-random outputs. Since RDSEED reads directly from the CE buffer, this bound relies more heavily on the entropy source and CBCMAC extractor (and less on the computational security of AES). Again, see the full version [16] for a proof.

Theorem 8 (ISK-RNG's masked forward security). *Fix a positive integers k and m, and fix $0 < \beta \leq 1$. Let L_m be a positive integer. Let A be a delayed adversary making a combined q queries to* get-next *and* next-ror*. Then if $\mathcal{D}$ is β-healthy, there exists some adversary B making three queries and running in the same time as A such that*

$$\mathbf{Adv}^{\mathrm{fwd}/M}_{\mathsf{ISK},\mathcal{D}}(A) \leq (q+1)\left(2^{(k-m\gamma)/2} + \epsilon(L_m) + 2\hat{\epsilon}(L_m)\right) + 2(q+4)\left(\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{AES}}(B) + \frac{3}{2^k}\right).$$

Corollary 1. *Let A be a delayed adversary making a combined q queries to its* get-next *and* next-ror *oracles. If $\mathcal{D}$ is β-healthy, then there exists*

and adversary B making three queries and running in the same time as A such that

$$\mathbf{Adv}^{\mathrm{fwd}}_{\mathsf{ISK},\mathcal{D}}(A) \leq (q+5)\left(2^{(k-m\gamma)/2} + \epsilon(L_m) + 2\hat{\epsilon}(L_m)\right) + (2q+13)\left(\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{AES}}(B) + \frac{3}{2^k}\right),$$

where the remaining quantities are defined as in Theorem 8.

The corollary follows from applying Theorem 3 to Theorem 8 and Lemma 1. We defer our discussion of this bound to Section 7.3. First, we briefly turn our attention to the questions of backwards security and robustness.

Backwards security and Robustness. The issue with obtaining backwards security (and hence robustness) is that future outputs can linger in the output buffers indefinitely: the hardware will shutdown the entropy source after all the buffers are full and the CE buffer is available. Hence, state remains compromised until fresh entropy filters through the $\mathsf{ESSR} \rightarrow \mathsf{OSTE}_1 \rightarrow \mathsf{OSTE}_2 \rightarrow \mathsf{CE}$ buffers and is used to reseed the DRBG, without first being siphoned off by `RDSEED`.

Consider the worst-case scenario for Ivy Bridge chips, where only the `RDRAND` interface is available. Following a state compromise, the next eight outputs are revealed by the output buffers, the next 511 may be computed using the compromised DRBG seed, the next 511 may be computed using a DRBG seed determined by the compromised CE buffer, and the next 511 may be computed using a DRBG key determined by the compromised OSTE and ESSR buffers. This amounts to slightly more than 12 KB of outputs that an adversary could potentially predict.

However, we show in the full version [16] that if one restricts the model to "read-only" adversaries (by denying adversaries access to set-state but permitting access to get-state) *and* one discounts wins based on the above attacks (by denying adversaries access to next-ror until after the "corrupted" values have already been replaced) then ISK-RNG is secure. The concrete bounds we obtain are essentially identical to those provided by Theorems 7 and 8, depending on whether or not one requires the `RDSEED` interface to be secure. See the appendix for further discussion of how these restrictions can be interpreted along with a formal theorem statement and proof.

7.3 Discussion of results

Let us examine the bound of Corollary 1 in detail. We specialize to the parameters used by Intel: $k = 128$ (a consequence of using AES), $m = 2$ for Ivy Bridge chips, and $m = 3$ for Broadwell chips.

To estimate γ, we turn to the CRI report [8]. Hamburg, Kocher, and Marson subjected raw entropy source bits (using data provided by Intel) to a battery of statistical tests. Using a Markov model with 12 bits of state, they estimate the entropy source produces approximately 0.65 bits of min-entropy per bit of output. However, this was an average (some states of the Markov model resulted in more predictable bits), and a 12-bit state, though perhaps necessary to collect enough samples for a meaningful empirical analysis, is not enough for our purposes. Therefore let us suppose a more conservative rate of 0.5, leading to $\gamma = 128$.

This sets the $(q+5)2^{(k-m\gamma)/2}$ term of our bound to $(q+5)2^{-64}$ for Ivy Bridge (where $m = 2$) and $(q+5)2^{-128}$ for Broadwell (where $m = 3$). The latter bound is quite strong, but, given how quickly q can grow, the former may be worrisome if one wishes to maintain strong security guarantees (e.g., one wishes to cap an adversary's advantage at 2^{-40}). But this is not the dominate term.

We next consider the term $(q + 5)(\epsilon(L_m) + 2\hat{\epsilon}(L_m))$. If we set the big-O constant of ϵ to c (so $\epsilon(L_m) = cL/2^{64}$) then we can choose L_m to optimize this expression. Taking $\beta = 1/2$, $c = \sqrt{10}$, which we believe to be conservative,[3] gives an upper bound of $(q + 5)2^{-56}$; a more generous $\beta = 0.99$, $c = 1$ improves the upper bound to about $(q+5)2^{-60}$. (These bounds are accurate for both $m = 2$ and $m = 3$, although the corresponding values for L_m differ considerably.)

At this point, limiting an adversary's advantage to 2^{-40} is difficult — an adversarial process gathering random bits at the benchmarked rate of 500 MB/s could issue the maximum allowable number of queries in under one millisecond. Or at least, this is the case if we demand that `RDSEED` produces uniform random outputs. On the other hand, if one only needs `RDRAND` to be secure, then Theorem 7 suggests that limiting an adversary's advantage to 2^{-40} is entirely reasonable; in this setting, we only pick up a single $4(\epsilon(L_m)+2\hat{\epsilon}(L_m))$ term even after moving to the unmasked forward-security setting, with no troublesome multiplicative factor of q.

[3] An author of [4] assures us that the asymptotic constant is "certainly less than 10" (and our c is the square root of this constant). A perfect entropy source would give $\beta = 0.99$ since the health tests have a 0.01 false-positive rate.

The remaining term, $(2q + 13)(\mathbf{Adv}^{\mathrm{prp}}_{\mathsf{AES}}(B) + 3/2^{128})$, is likely to be negligible (recall that B is permitted only three queries).

Our analysis does not point to any obvious, practical attacks (aside from the trivial ones that exploit the output buffers, though it seems a stretch to deem those practical). However, it exposes the CBCMAC extraction process as the likely weakest link, and quantifies the extent of that weakness. An actual attack would need to exploit how the specific output distribution of the entropy source interacts with CBCMAC under the fixed key K'.

7.4 Discussion of the attack model

The DPRVW syntax and security notions, which we take as our starting point, assume a strongly adversarial operating environment. They treat the entropy source as adversarial (although not pathologically bad), and allow attackers to observe, even corrupt, the full internal state of the PWI. One might argue that these choices are inappropriate in the case of ISK-RNG. After all, the entire RNG is implemented in 22-32nm hardware, so direct observation of the internal state should require the use of expensive and highly technical equipment, e.g. a state of the art scanning/tunnelling electron microscope.

We are sympathetic to this argument, but still find value in adopting the strong attack model. Even if the entropy source is beyond attacker influence, treating it as adversarial can bee seen as a mathematical tool for minimizing the assumptions we make regarding its behavior. Moreover, the model allows us to explore the limits of ISK-RNG's security, providing analysis of less pessimistic settings (i.e. resilience security) as a byproduct.

Acknowledgements

The authors wish to thank DJ Johnston and Jesse Walker of Intel for answering our questions regarding the design and implementation details of ISK-RNG. Both Terashima and Shrimpton were supported by NSF grants CNS-0845610 and CNS-1319061.

References

1. Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to `/dev/random`. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 203–212. ACM, 2005.

2. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology—EUROCRYPT 2006*, pages 409–426. Springer, 2006.
3. Olivier Chevassut, Pierre-Alain Fouque, Pierrick Gaudry, and David Pointcheval. The twist-AUgmented technique for key exchange. In *Public Key Cryptography*, pages 410–426. Springer, 2006.
4. Yevgeniy Dodis, Rosario Gennaro, Johan Håstad, Hugo Krawczyk, and Tal Rabin. Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In *Advances in Cryptology—CRYPTO 2004*, pages 494–510. Springer, 2004.
5. Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input: `/dev/random` is not robust. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 647–658. ACM, 2013.
6. Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, and Michael Swift. Not-so-random numbers in virtualized linux and the Whirlwind RNG. *IEEE Symposium on Security And Privacy*, 2014.
7. Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the Linux random number generator. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.
8. Mike Hamburg, Paul Kocher, and Mark E Marson. Analysis of Intel's Ivy Bridge digital random number generator. *Online: http://www. cryptography. com/public/pdf/Intel_TRN G_Report_20120312. pdf*, 2012.
9. Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, pages 205–220, 2012.
10. Gael Hofemeier. Intel Digital Random Number Generator (DRNG) software implementation guide. https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide, August 2012. Accessed May 2014.
11. Gael Hofemeier and Robert Chesebrough. Introduction to Intel AES-NI and Intel Secure Key instructions. https://software.intel.com/en-us/articles/introduction-to-intel-aes-ni-and-intel-secure-key-instructions, July 2012. Accessed May 2014.
12. JD Johnston (Intel). Personal communication, May 2014.
13. Patrick Lacharme, Andrea Röck, Vincent Strubel, and Marion Videau. The Linux pseudorandom number generator revisited. *IACR Cryptology ePrint Archive*, 2012:251, 2012.
14. John Mechalas. The difference between RDRAND and RDSEED. https://software.intel.com/en-us/blogs/2012/11/17/the-difference-between-rdrand-and-rdseed, November 2012. Accessed April 2014.
15. Jaikumar Radhakrishnan and Amnon Ta-Shma. Bounds for dispersers, extractors, and depth-two superconcentrators. *SIAM Journal on Discrete Mathematics*, 13(1):2–24, 2000.
16. Thomas Shrimpton and R. Seth Terashima. A provable security analysis of Intel's Secure Key RNG. Cryptology ePrint Archive, Report 2014/504, 2014. `http://eprint.iacr.org/`.
17. Jesse Walker. Conceptual foundations of the Ivy Bridge random number generator. http://www.ists.dartmouth.edu/docs/walker_ivy-bridge.pdf, November 2011.